Series HOMEPET

Introducción al C++

Primera Parte: un súper C

Ing. Simón Galliano Vidal, ret.

17 Temas
19 Ilustraciones
33 Tablas
6 Programas auxiliares
31 Programas resueltos
40 Ejercicios propuestos

Publicado online por El Archivo Histórico de la Ciudad de Manzanillo

URL: http://www.enciclopedia-manzanillo.cu

Fecha de publicación: 12 de noviembre de 2020

Versión: 3.5C.1020

Autor: S. GALLIANO - 2019

Publicista: Lic. Delio G. Orozco.

ISBN: 978-959-7179-67-2(1)

Páginas: 211

CONTENIDOS

ILUSTRACIONES

TABLAS

Primera Parte: un súper C

Programación Imperativa en C++

El proceso de preparar programas para una computadora digital es especialmente atractivo, no sólo porque puede ser económica y científicamente premiado, pero también porque puede ser una experiencia estética, casi como componer poesía o música.

Donald E. Knuth

Profesor Emérito de la Universidad de Stanford

Ing. Simón Galliano Vidal, ret. 2020

1 - SISTEMA DE DESARROLLO

Code::Blocks es un completo IDE (entorno integral de desarrollo o Integrated Development Environment en inglés) multiplataforma para el desarrollo de programas en los lenguaje ANSI C e ISO C++, creado en C++ y liberado bajo la Licencia pública general de GNU. (Wikipedia en español)

Para esta primera parte de la serie y en un sistema operativo (SO en lo adelante) Windows 10 Home x64, con un Celeron® J1800 @ 2.41 GHz y 8 Gb de memoria, se usa el IDE Code::Blocks, versión 20.03-r11983 de dominio libre porque:

- Es gratis y está en Internet (http://www.codeblocks.org)
- Está hecho totalmente sobre C++, siendo muy liviano al SO
- Se ejecuta en cualquier versión actual de Windows™, sea XP, Vista, Seven, 8 ó 10.
- Automáticamente busca en el SO un compilador cualquiera para C++ y lo integra. Acá se usó la suite para Windows tdm-gcc-4.6.1, edición estándar de 32 bits para C++ que es bastante conforme con ISO C++11[1].
  - ✓ C++11, es el nombre de la segunda y más implementada versión de la norma ISO/ANSI para C++, aprobada el 12 agosto del 2011, que fuera reemplazada por la C++14 y más recientemente por la C++17. Proviene de la tradición de nombrar las versiones de C++ por la fecha de la publicación de la especificación. La versión del compilador del IDE que aquí se usa es muy completa y bastante apegada a C++11
  - ✓ ISO: *International Standard Association*: asociación internacional de normas, en español.
  - ✓ ANSI: *American National Standard Institute*: instituto nacional norteamericano de normas, en español.
- Como IDE, preprocesa, enlaza y compila de modo automático.
- El manejo de su entorno laboral es relativamente rápido de aprender, y una vez aprendido es muy cómodo y eficiente para codificar, y muy flexible para adoptarlo a un estilo propio.
- Tiene opciones para crear programas de consola (*Console application*, en inglés), los usados aquí.

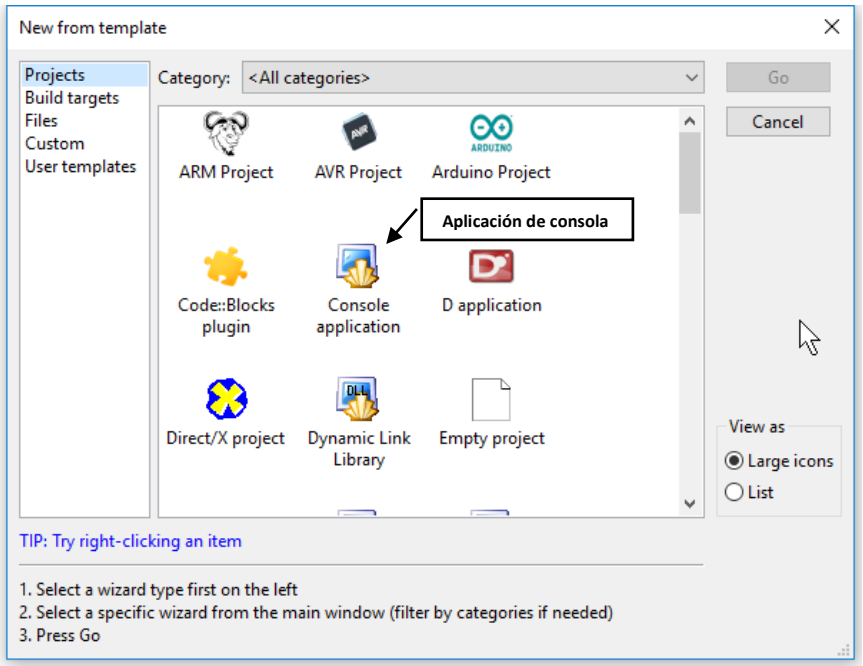


Ilustración 1. Opciones básicas de inicio

[1] ISO C++ se refiere específicamente a la versión normalizada de C++ y en el texto se referencian como la misma cosa. Algunos fabricantes de compiladores se apartan de la norma y pierden portabilidad. Eso no pasa aquí.

Todos los programas del texto toman como opción la aplicación de consola

Esto no es un respaldo oficial o a ultranza de dicho IDE. Allá afuera hay un buen puñado de ellos. Siéntase libre de estudiar usando uno de su gusto, aunque tendrá que ajustar lo que aquí se muestra, siempre con la esperanza de que las adaptaciones sean verdaderamente mínimas; en la inmensa mayoría de los casos, nulas, ¡el Santo Grial de la portabilidad!

Bajo este IDE, en el SO Microsoft Windows™ los programas correrán en una ventana separada que el usuario cerrará al concluir éstos, pero al correr aparte el ejecutable, para mantener la salida en consola basta con incluir **`<cstdlib>`** (*C++ standard library* en inglés), la biblioteca estándar de C++, y pasarle al SO la orden de pausar antes del retorno a Windows**:** **`system("pause");`** Esa salida se suprime en el texto por ser redundante.

Los programas gráficos son largos y complicados y dicho de un modo sencillo, *Windows™ no es un entorno apropiado para aprender a programar* por lo que en este texto introductorio el IDE fue utilizado en modo consola, tendencia generalizada en la bibliografía consultada. EODA[2], la programación visual, gráfica, manejada por eventos, se estudia de modo distinto, mejor y más fácilmente luego de operar con soltura en la programación tradicional.

Seguramente habrá alguien a quien gustaría aprender a programar visualmente desde un inicio. Un verdadero reto pedagógico. El autor de veras que lo siente, pero este no es el caso.

Objetivo del texto

C++ es uno de los lenguajes de computación más potentes jamás desarrollados. Es rápido, eficiente y empleado por la inmensa mayoría de las aplicaciones que se ven regularmente en el mercado actual. Sensitivo a los tamaños de letras, con código en formato libre, sus características básicas son el ser de nivel medio y propósito general, terso, estructurado, modular, multi-paradigmático e híbrido.

En un nivel introductorio, el objetivo único de este tomo es enseñar a codificar en C++ desde el paradigma de la programación imperativa, sin necesidad de precondición alguna. Programando como un súper C —definido por el autor como *cuando el programador usa un compilador C++ y sus características propias, pero sin aplicar a fondo las bondades de la STL o los fundamentos teóricos de la OOP*— se beneficia a un programa ANSI C con la mejor seguridad que brinda la compilación de C++ y sus facilidades de uso de las E/S, manejo de archivos, dominio de complejidades, etc.

El objetivo general de cada tema básico es enseñar un concepto más o menos simple, y de cada tópico en particular allí cubierto, un conjunto de conceptos que le son asociados de alguna suerte. Esto se consigue respaldándolo con un programa de ejemplo[3]. El autor concuerda con (Eckel, págs. xxvii-xxviii) en que se debe:

- Presentar el material paso a paso, de manera que el lector pueda asimilar cada nuevo concepto antes de proseguir.
- Usar ejemplos tan sencillos y cortos como sea posible y siempre que sea posible, porque cuando los estudiantes pueden entender cada detalle de un ejemplo, avanzan más y comprenden mejor. Además, hay ciertos límites subjetivos y variables, pero presentes en cada uno en particular, sobre la cantidad de código que pueda asimilar en un tema y el común denominador es tratar de mantener lo más bajo posible el nivel de dificultad.
- Mostrar un conjunto minimalista de tópicos que son básicos para comprender el lenguaje a un nivel introductorio, pero realmente útil, esto es, usable a partir de ahí.
- Mantener cada tema lo suficientemente enfocado para que el tiempo de estudio y práctica entre temas (que desde luego, aumentará según éstos se complejicen) en promedio sea moderado[4], forjando mayor confianza en el avance.

[2] EODA: En Opinión Del Autor…

[3] Al inicio la parte teórica será mayor que la práctica, pero a medida que se avanza, las cosas cambian…

[4] Moderado es un término subjetivo que representa algo distinto para cada individuo. EODA, el lector debe olvidarse de aprender formalmente algo serio, no ya en 24 horas, pero ni siquiera en 21 días. No es que esas series de libros no sean buenas: lo son, y

- Ofrecer al lector una base sólida, de manera que pueda comprender las cuestiones planteadas y aplicarlas en la solución de los problemas propuestos, o en su trabajo, o en sus estudios oficiales, para que se beneficie inmediatamente y al concluir el curso, pueda pasar a otros niveles más profundos y extensos, o sea, libros de texto especializados, cursos avanzados, etc.
- Introducir las técnicas básicas de programación y los buenos hábitos de escritura del código, empleando algunos elementos avanzados cuando se requieran para solucionar un problema, aunque tratando lo más posible de dejar las complejidades –que sin dudas vendrán– para los otros dos tomos[5], los cuales el autor tiene la esperanza que serán completados...tarde o temprano.

Desafortunadamente y con cierta frecuencia a medida que se avanza, se presentará algún tópico requiriendo de conocimientos que no han sido explicados todavía, producto de que la enseñanza en general —y la de la computación en particular— raramente avanza en línea recta, antes bien, progresa en espiral dialéctica y ascendente, yendo de lo conocido a lo desconocido y de lo simple a lo complejo. Pero en ese caso se ofrece una breve descripción aclaratoria, o se envía a una página donde su concepto esté más ampliado.

Sepa el amable lector que los programas resueltos, problemas propuestos y fragmentos de código aquí dados fueron extraídos de la práctica del autor, de la bibliografía consultada o concebidos para este tomo y todos fueron adaptados al objetivo pedagógico; y todos fueron hechos y comprobados por el autor. Dispense pues cualquier gazapo que aparezca.

¿Cómo escribir el código?

La elegancia en la escritura y la claridad en la lectura de un programa no es un problema de gustos: ¡es una necesidad del oficio! Ver más adelante el Zen de Python, que entre otras cosas dice que *la legibilidad cuenta.* El autor recomienda al curioso lector fijarse bien en todos los ejemplos dados en el texto, estudiarlos hasta comprenderlos y —aunque respetando su propio estilo de escritura— reproducirlos y ejecutarlos sin falta, lo más fielmente posible. En general, por ahora en el código se debe:

- Poner cada instrucción en línea aparte.
- Dejar al menos un espacio en blanco entre los bloques lógicos o cíclicos del código y el resto de las instrucciones.
  - ✓ Bloque es un conjunto de instrucciones puestas entre llaves. En un bloque usualmente se pueden declarar las variables necesarias al trabajo que de él se espera. Cuando la última instrucción del bloque se ejecuta, se le termina su tiempo de vida y desaparece de la memoria junto con todo lo que lleva adentro.
- Dejar al menos un espacio en blanco entre los componentes de una instrucción.
  - ✓ Instrucción es un conjunto de caracteres legales, insertados en una secuencia específica del lenguaje de programación dado, con una forma de terminación fijada y que tiene un significado sintácticamente correcto.
- Poner siempre las llaves de apertura y cierre de bloque en torno a las estructuras de decisión y ciclo, aun cuando la sintaxis tolere su ausencia, y además comentarles el cierre.

---

mucho. Solamente se señala que el tiempo utilizado en aprender, siempre sobrepasará con creces los cortos plazos desenfadadamente prometidos, como gancho y gestión de ventas.

[5] El tema, aún a nivel introductorio es tan extenso, que el autor decidió dividirlo en tres tomos: éste que está en sus manos; un segundo tomo dedicado a la introducción de la programación genérica de funciones y al uso elemental de la biblioteca estándar de plantillas; y un tercero dedicado al estudio introductorio de las clases, su creación y manejo básico.

Se aprovecha la facilidad de adaptación del IDE utilizado, aplicando al código manualmente hecho un formateo uniforme mediante el uso del **plugin**[6] **AStyle** (*Art Style* o estilo artístico, en inglés) configurado y readaptado para el estilo de escritura en Java™, que sigue muy de cerca al estilo clásico de Kernighan y Ritchie. EODA el resultado es un código muy elegante, claro y uniformemente escrito.

Sangría

Todos los códigos escritos en cualquiera de los múltiples lenguajes imperativos de programación existentes al día de hoy, están sangrados. Algunos IDE lo hacen automáticamente, otros a demanda del programador y en otros casos, el programador sangra usando espacios o tabulaciones. Los estilos difieren incluso para un mismo lenguaje, pero la sangría siempre está presente y no se puede ignorar.

- ✓ Indentación o sangría es la acción de poner cierta cantidad de espacios en blanco al inicio de los bloques, cuerpo de funciones y datos complejos dentro de estructuras, preservando los múltiples niveles de jerarquía, con la intención de facilitar la lectura humana del código.

En las primeras versiones del lenguaje C, las definiciones se escribían y sangraban así:

```
int strlen(str)
char *str;
{
   int n = 0;
   while ( str[n] != NULL )
        n++;
   return (n);
}
```

Con el tiempo surgieron otros estilos que dieron origen a guerras sin cuartel entre los gurúes. Sin embargo, los autores del C clásico usaron la llave de apertura en la misma línea que la llamada a función o la instrucción de control, y pusieron la llave de cierre alineada con la llamada o instrucción. Según (Pugh) es el llamado *"Estilo de Kernighan y Ritchie"*, y su versión actualizada y usada en el texto es:

```
int strlen(char *str) {
   int n = 0;

   while ( str[n] not_eq nullptr ) {
      ++n;
   } // while

   return n;
}
```

El lector ha de comparar ambos estilos y tratar de establecer sus diferencias. Es importante, ya que permite ver al vuelo si un texto es muy antiguo y evitar errores al convertir sus ejemplos, puesto que el código escrito en estilo antiguo, ahora es refutado por los compiladores modernos. Además, el texto usa los operadores de comparación explícitos, es decir, **and**, **or** y **not_eq**.

¿Cuál es la guía para escribir buen código?

Mientras que algún desacuerdo es inevitable [...] esperamos que el experto no lo condenará porque algunas reglas están en conflicto con sus opiniones personales [...] Si Ud. está en desacuerdo con una regla, lea su justificación. Si luego de

[6] En las ciencias de la computación se llama **plugin** a una pieza de código que se incluye en el entorno de trabajo, de modo similar a como se enchufa un efecto electrodoméstico a la corriente.

leerla sigue en desacuerdo, simplemente no la use. Las reglas deben ser aplicadas consistente, pero no ciegamente [...] La consistencia tiene mucha importancia. La obediencia ciega no. Ranade & Nash - The Elements of C Programming Style. McGraw-Hill, 1993.

Mucho se ha hablado de ello y varios textos especializados han sido escritos. Algunos son: Plum: *C Programming Guidelines*; Strunk & White: *The Elements of Style*; Kernighan & Plauger: *The Elements of Programming Style*, etc.

Características generales de un buen código

Siguiendo muy de cerca a (Gottfried, pág. 11), este autor opina que –independientemente del paradigma utilizado en su codificación– un programa moderno debería de cumplir con la gran mayoría de las doce características generales siguientes (mientras más, mejor) dadas en orden de importancia:

1. Integridad: se refiere a las respuestas obtenidas a los cálculos hechos sobre los datos suministrados. Si se obtienen resultados erróneos, sobra todo lo demás. Sin discusión, esta característica es la más importante de todas.
2. Precisión: tiene que ver con que los resultados obtenidos respondan cabalmente a las rutinas de cálculo usadas. Si no hay precisión las respuestas serán muy íntegras…y muy imprecisas. Le sidue en eimportancia.
3. Fiabilidad: se refiere a que el programa hace bien todo lo que debe hacer durante su ciclo de vida, tal y como se estableció en las fases de diseño previas a su desarrollo. Es la tercera en importancia e íntimamente ligada a las dos anteriores, porque un programa íntegro y preciso, pero apartado de los lineamientos de su diseño, dará respuestas posiblemente muy equivocadas o sin mucho sentido y desde luego, no será fiable.
4. Claridad: caracteriza al grado de legibilidad del código por parte de un humano, tanto de su sintaxis, como de su lógica de trabajo ya que el código se lee muchas veces, pero se escribe o rescribe pocas. La ausencia de esta característica ha sido fuente inagotable de recargo de costos y de ofuscación manifiesta, presente muchas más veces de lo debido en operaciones de mantenimiento de aplicaciones POD.
    - ✓ POD significa *Plain Old Data*, o dato viejo en español y en C++ se aplica fundamentalmente al código que fuera desarrollado desde C.
    - ✓ Ofuscación[7], opacidad u obscurecimiento es una característica presente en el código que ignoró la filosofía de Python o que, partiendo de un código claramente escrito, ha sido enrevesado a propósito para ocultar su funcionalidad, o para evitar la ingeniería inversa, o por el mismo codificador para satisfacer su ego, etc.
5. Modularidad: se refiere a repartir un proyecto en módulos (hacerlo modular), aplicando su estructuración en tantas funciones o clases como sea posible (bibliotecas propias y módulos lógicos), y agrupando sus partes lógicas en ficheros escritos aparte (módulos físicos) que van fuera del programa principal.
    - ✓ La unión lógica de las estructuras de datos y sus funciones auxiliares dio origen al concepto de tipo abstracto de dato o ADT (*Abstract Data Type*, en inglés.) La integración física de datos y funciones del ADT, a su vez dio origen al surgimiento de la Programación Orientada a Objetos u OOP en lo adelante (OOP: *Object Oriented Programming,* en inglés), con las clases y sus representaciones, los objetos, que son instanciados (definidos) en un tipo concreto de dato o CDT (CDT: *Concrete Data Type*, en inglés.)
6. Simplicidad: puntualiza la generación de un código lo más corto y sencillo posible dentro de los objetivos básicos del proyecto.

[7] Existen lenguajes, como Forth (ideado por C. Moore y E. Rather) o Joy (creado por M. von Thun), donde el código ofuscado es algo propio, natural. Por cierto, el ANSI C se presta muy bien para eso. Hace ya muchos años existieron concursos de código ofuscado hecho a propósito. Ver (Pugh), o (Prata).

7. Generalidad: se refiere al hecho de solucionar un proyecto serio, profesional, en base a lineamientos generales, propios del entorno del problema –lo que constituye la base filosófica de la OOP– y no en base a lineamientos particulares, definidos puntualmente por el problema. El diseño correcto de las clases realza esta cualidad aún más.

> Aunque la característica es elusiva y nada fácil de conseguir, la parametrización correcta de los módulos lógicos permite elevar considerablemente y con muy poco esfuerzo la generalidad en el proyecto. *(Deitel & Deitel)*

8. Robustez: se refiere a un programa resistente a los errores de forma que, si se comete uno cualquiera, es capaz de recuperarse sin abortar o mal funcionar.

9. Tipicidad (también alternativamente llamada normalización): se aplica al diseño que tipifica el uso de la aplicación, la toma de datos y la presentación de los resultados obtenidos.

10. Portabilidad: característica cualitativa que representa la posibilidad de poder ejecutarse en una plataforma combinada de hardware/software, diferente a aquella en la que se elaboró, sin tener que alterar el código: a menos alteraciones, mayor grado de portabilidad. El especialista decide si es buena, regular, etc.

> Esta es una característica cualitativa muy deseable pues, por ejemplo, permite a un programa que se ha gestado en sistemas GNU/Linux, ejecutarse también en la familia de sistemas operativos Windows™ o MacOs™, haciendo que los costos de desarrollo se abaraten, el precio del producto sea (relativamente) bajo, y la aplicación pueda llegar fácilmente a más usuarios.

11. Eficiencia: se refiere a un concepto ligado con la meta de alcanzar un desarrollo óptimo de cada proyecto serio, que es definido en base a los costos generados.

- ✓ Eficiencia: *concepto general cuantitativo* representado por un número calculado como la razón entre el costo real del consumo de recursos y el planificado para realizar un trabajo cualquiera. Mientras más bajo sea el número obtenido, mejor es la eficiencia del producto. Multiplicado por 100 se da en porciento[8].

12. Eficacia: se refiere a un concepto ligado con la meta de alcanzar un desarrollo óptimo de cada proyecto serio, que es definido en base al consumo de todo tipo de recursos.

- ✓ Eficacia: *concepto general cualitativo* dado como una estimación del alcance de dicha meta usando un mínimo de recursos. Generalmente se define por parte de los especialistas de cada ciencia como una categoría del tipo excelente, típica, buena, etc.

El lector novel tendrá presente todas estas características al programar profesionalmente

---

La Filosofía del Zen de Python

Según (Wikipedia en español), el código que sigue los lineamientos de legibilidad y transparencia del lenguaje de programación Python, se dice que es pitónico y garantiza un código claro y razonablemente libre de errores. El código que desconoce o ignora estos conceptos generalmente produce código ofuscado en mayor o menor medida.

Estos principios que seguidamente se muestran, algunos de los cuales serán referidos en el texto, fueron descritos por Tim Peters (uno de sus desarrolladores oficiales) como reglas para escribir un código claro, sencillo, legible y con muy buenas probabilidades de estar libre de errores desde un principio:

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.

---

[8] Muchas ciencias tienen su propia definición de eficiencia, por eso, ante la duda el lector consultará fuentes adecuadas.

- Plano es mejor que anidado.
- Disperso es mejor que denso.
- La legibilidad cuenta.
- Los casos especiales no son tan especiales como para quebrantar las reglas.
- Lo práctico gana a lo puro.
- Los errores nunca deberían dejarse pasar silenciosamente...
- ...a menos que hayan sido silenciados explícitamente.
- Frente a la ambigüedad, rechaza la tentación de adivinar.
- Debería haber una —y preferiblemente sólo una— manera obvia de hacerlo.
- Aunque esa manera puede no ser obvia al principio, a menos que usted sea holandés.
- Ahora es mejor que nunca.
- Aunque nunca es a menudo mejor que ahora mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede ser que sea una buena idea.
- Los espacios de nombres son una gran idea ¡Hagamos más de esas cosas!

Nota: holandés aquí hace referencia a Guido van Rossum, autor de dicho lenguaje, que es oriundo de ese país. De paso, también hace referencia a la gran cantidad de desarrolladores holandeses que existen en relación a otras nacionalidades.

El autor exhorta al lector que se apegue lo más posible a la Filosofía del Zen de Python

Mantenimiento

Cualquier trabajo realizado para cambiar una aplicación que está en operación es considerado mantenimiento. Su gran objetivo es preservar la fiabilidad de dicha aplicación en el tiempo. El mantenimiento puede abarcar tópicos como eliminación de errores, ampliación de la clientela, integración de requisitos adicionales, incremento de la facilidad de uso y de la eficacia y la añadidura de nuevas tecnologías. Mientras que el desarrollo de un nuevo producto puede abarcar un par de años, ¡el mantenimiento subsiguiente puede abarcar hasta unos veinte!

Debido a que el cambio es inevitable, las casas productoras desarrollan mecanismos para evaluar, controlar y hacer modificaciones al código. Esta actividad es muy amplia e incluye mayormente el uso de control de versiones, la corrección de errores lógicos (*bugs* en inglés); la mejora de capacidades de la aplicación, y la refactorización, actividad que también abarca la limpieza del código "muerto", o sea, código obsoleto, con partes inalcanzable, con ciclos sinfín, etc.

Algunos significados acerca de C++

El lenguaje de programación C++ puede ser descrito como de nivel medio, propósito general, terso, estructurado, modular, multiparadigmático e híbrido.

Decir que C++ es un lenguaje de programación:

- De nivel medio, significa establecer que la codificación en C++, aunque es fácil de escribir en alto nivel (con instrucciones que parecen redactadas en inglés), entrega muchísimo de la potencia, rapidez y compactación de un lenguaje de bajo nivel como son los ensambladores.
  - ✓ Un lenguaje es de alto nivel cuando sus instrucciones expresan los algoritmos de manera más adecuada a la comprensión del ser humano. Los lenguajes de programación de alto nivel en general deben ser independientes de la arquitectura del equipo y de su SO, y si alguno de estos factores cambia, en teoría quizás sólo se cambie el compilador que le implementa, pero al menos no hay que aprender desde cero a trabajar con otro lenguaje.

- ✓ Un lenguaje de bajo nivel, también llamado lenguaje ensamblador es aquél cuyas instrucciones ejercen control directo sobre el hardware; se dice que van pegadas al microprocesador y son muy, muy eficientes y muy, muy difíciles de aplicar. Estando ligados a la arquitectura del equipo y a su SO, si alguno de estos dos elementos cambia, hay que cambiar el lenguaje ensamblador por otro que les "conozca" a ambos.

Mientras más alto sea el nivel, mayor es la portabilidad presente, y la rapidez y facilidad en su codificación...y mayor el agrandamiento del código, el uso de memoria y la pérdida de rapidez en su ejecución. Pero se aclara al lector que decir código fácil de escribir no implica que además sea fácil de concebir, crear o implementar. El problema a resolver puede estar fuera del alcance de los conocimientos del codificador y desde luego, él no podrá resolverlo por sí solo.

- De propósito general, es destacar que teóricamente se puede escribir cualquier aplicación que resuelva un problema dentro del ámbito de aquellos que son solubles mediante el uso de computadoras. Dicho ámbito abarca (pero no agota) las áreas de bases de datos, paquetes estadísticos, bibliotecas gráficas, sistemas informáticos, sistemas de inteligencia artificial, manejo de sistemas en tiempo real, etc. Para cada área bien delimitada, usualmente existe más de un lenguaje de programación muy adaptado al tipo del problema, que facilita mucho su solución, pero al menos en teoría, con C++ se puede resolver todo problema que pudiera presentarse allí.

  Muchos de los programas, aplicaciones, juegos, e incluso SO completos están hechos en C++, incluyendo todas las aplicaciones de Adobe, muchos navegadores de Internet, el SO macOS de Apple, el SO Windows 10 y la suite ofimática MS Office de Microsoft, el núcleo del buscador de Google, y es el lenguaje de oficio en la NASA, la empresa norteamericana privada de transporte aeroespacial SpaceX y la Organización Europea para la Investigación Nuclear, comúnmente conocida como el CERN. (Linux Tricks and Tips, págs. 40-41)

- Terso, es aclarar que su núcleo está compuesto por los elementos esenciales de programación y que otros de apoyo, tales como las operaciones de entrada y salida (E/S en lo adelante), tratamientos matemáticos, algorítmicos, gráficos y de ingenierías, estructuras complejas de datos, etc. son adicionados según demanda.
- Estructurado, es destacar que posee tanto las estructuras típicas de control del flujo del programa, basadas en decisiones y ciclos, como los tipos estándar de datos estructurados.
- Modular, es matizar que puede basarse enteramente en módulos para realizar su trabajo y en compilación separada para organizarlo.
- Multiparadigmático es indicar que partiendo de ser en su base un lenguaje imperativo, acepta los paradigmas de la OOP y de la programación genérica.
- Híbrido, es puntualizar que sus características especiales de manejo de la OOP y de la programación genérica no son de aplicación obligatoria. Están allí, pero se puede programar en C++ sin necesidad de ellas, como si fuera un súper C, objetivo pedagógico de esta primera parte de la serie. Es el programador, basado en su experiencia y conocimiento de esos temas, quien decide si aplica dichas características, cuándo y dónde.

Caracteres en español

En el sistema por donde se montó el curso, la salida de caracteres fuera del ámbito más estrecho del código ASCII (de los números 32 a 127) no es afortunada.

- ✓ ASCII: código estándar norteamericano para el intercambio de información (acrónimo de *American Standard Code for Information Interchange,* en inglés), fue creado en 1963 por el comité ANSI. En 1967, año en que se publicó como norma, se incluyeron las minúsculas y se redefinieron algunos códigos de control ¡Fue actualizado por última vez en 1986!

Por ejemplo, en este sistema, si da salida directa a una ñ, el monitor (consola en lo adelante) muestra el carácter ±, aunque cuando se envía una cadena a un archivo y luego se recupera, C++ la pone tal como se ve entre comillas. Cuando esos

caracteres están representados por tres números enteros, C++ los interpreta directamente en formato octal. La á tiene el código ASCII decimal 160, pero para sacarla a consola dentro se debe poner la secuencia de escape \240 octal.

Para solucionar este problema, el autor empleó los siguientes códigos embebidos en las cadenas de los programas, prefiriendo la legibilidad en la salida de los programas a consola antes que en el libro:

Tabla 1. Códigos de escape en formato octal para el idioma español

\240 es la á	**\265 es la Á**	**\202 es la é**	**\220 es la É**
\241 es la í	**\326 es la Í**	**\244 es la ñ**	**\245 es la Ñ**
\242 es la ó	**\340 es la Ó**	**\250 es el ¿**	**\255 es el ¡**
\243 es la ú	**\351 es la Ú**	**\201 es la ü**	**\232 es la Ü**

Una palabra sobre los ejercicios

Los ejercicios fueron cuidadosamente diseñados para que el lector se apoyara en conocimientos previos. Por ejemplo, para resolver cualquier ejercicio propuesto en el tercer bloque, solamente se requiere de lo previamente aprendido hasta ese punto y cuando no hay limitantes (casi nunca los hay) el lector aplicará lo que estime conveniente.

El bloque uno comprende los temas 3-El primer programa, 4-Entrando números y 5-Entrando cadenas. El segundo bloque abarca los temas 6-Valores límites, 7-Decisiones y ciclos, y 8-Ciclos y decisiones. El tercero engloba los temas 9-Funciones y 10-Compilación separada. El cuarto bloque incluye al tema 11-Tipos estructurados, el cinco contiene al tema 12-Punteros y el sexto bloque al tema 13-Manejo dinámico de la memoria. Todos son temas afines que el autor prefirió separarlos por su dificultad inherente.

De aquí en adelante, los bloques se dedican a puntualizar aplicaciones: el séptimo bloque se dedica al tema 14-Listas simplemente enlazadas; el octavo al tema 15-Ordenamiento y búsquedas y el noveno al tema 16-Manejo básico de archivos. Y el tema, 17-Interactuando con el DOS, se destina a introducir un conocimiento complementario, que el autor considera necesario, aunque ciertamente no imprescindible, por lo que no propone ejercicios.

2 - UN TEXTO INTRODUCTORIO

> *Aprender el C++ es una aventura en sí mismo. Conocido popularmente como La Bestia, el lenguaje acomoda tres paradigmas de programación: el imperativo (heredado del C), el de la OOP (su razón de existir) y el de la programación genérica, llegado a última hora, pero tan importante como los otros. Este texto describe la norma C++11, aunque ya salieron la C++14 y la C++17. Los ejemplos dados se ejecutarán con cualquier compilador. El Autor*

Basado en conceptos mayormente proporcionados por los dos libros de Prata, (C Primer Plus, p. xxvii) y (C++ Primer Plus, 2012, p. 2) este autor opina que un texto verdaderamente introductorio es un documento que presentará como mínimo las siguientes cinco características:

1. Los conceptos de programación son explicados junto con los detalles del lenguaje; el texto no asume que el lector es un profesional del ramo y es bastante fácil de entender, usar y seguir.
2. Presenta ejemplos (relativamente) cortos, factibles de reproducir, que ilustran varios conceptos nuevos, a la vez que se apoyan —siempre que sea posible— en los anteriormente mostrados.
3. Ilustraciones y tablas aclaran los conceptos que son difíciles de entender con palabras solamente.
4. Para grupos de tópicos asociados, propone algunos ejercicios conceptualmente más retadores que los ejemplos dados, puestos aquí y allá, con el objetivo de asentar sólidamente los conocimientos adquiridos.
5. El texto puede emplearse para el auto-estudio o como apoyo de las clases en el aula.

El autor pretende que este volumen sea un texto verdaderamente introductorio

¿Cómo se clasifica el C++?

Creado por el dinamarqués Bjarne Stroustrup, en términos muy generales C++ es la versión orientada a objetos de C e inicialmente fue construido sobre los cimientos de la norma ANSI C llamada C89, incluyendo sus enmiendas de 1995. El hoy llamado ISO C++ duplica con mucho el tamaño de ANSI C y no hace falta enfatizar que es uno de más potentes lenguajes de computación jamás establecido. Su clasificación depende del punto de vista de quien la emita. Por ejemplo:

- **Según su comerciabilidad**. C++ es un excelente lenguaje de programación de alto nivel que es usado en multitud de tecnologías. Todo, desde su aplicación móvil favorita, pasando por juegos para PC o consolas, hasta SO completos, lo lleva en su núcleo, ofreciendo simultáneamente un conjunto de IDE y bibliotecas comunitarias para facilitar los desarrollos. (Linux Tricks and Tips, pág. 39)
- **Según su funcionalidad**. Ampliamente usado para desarrollo de software comercial, es un lenguaje profesional aplicado en…sistemas complejos en el área de los negocios, la ciencia y la ingeniería…Para poder manejar las demandas del software comercial, en sí mismo C++ es complejo… (Halterman, 2018, pág. 6)
- **Buscando la comodidad en el trabajo**. C++ no es sólo un lenguaje orientado a objetos; es un lenguaje de múltiples paradigmas. Acepta muchas características de la orientación a objeto, pero no obliga a los programadores a usarlas. Se pueden codificar proyectos completos sin esa orientación, y muchos así lo hacen. (Sutter, Exceptional C++, 2001, p. Item 25)
- **Desde un punto de vista pragmático**. La meta de C++ es mejorar la productividad. Ésta viene por muchos caminos, pero el lenguaje está diseñado para ayudarle todo lo posible, y al mismo tiempo dificultarle lo menos posible con reglas arbitrarias o algún requisito que use un conjunto particular de características. C++ está diseñado para ser práctico; las decisiones de diseño del lenguaje C++ estaban basadas en proveer los beneficios máximos al programador... (Eckel, 2000, pág. 30)
- **Basándose en la OOP**. C++ es la respuesta al programador de C que trabaja con la programación orientada a objetos. Basado en los sólidos fundamentos del C, C++ añade el soporte para OOP sin perder la capacidad, el estilo y la flexibilidad de C. De hecho, muchos programadores ven C++ como un C mejorado, independientemente de que permita programación orientada a objetos. Si usted…simplemente desea construir de una forma sencilla programas estructurados, C++ simplifica la labor de programación. (Schildt, C++. Guía de Autoenseñanza, pág. 8)

Brevísima historia

Stroustrup comenzó a desarrollar un lenguaje que llamó C con Clases en los laboratorios estadounidenses de la Bell Telephone Co. para su diploma de doctorado especializado en investigación (PhD), durante un periodo que abarcó básicamente desde mediados de los años ’70 hasta principios de los ’80, pero que aún continúa.

Lo hizo al principio como una especie de envoltorio del lenguaje C, porque uno de sus tantos propósitos originales, en sus propias palabras, fue: *…que ni yo ni mis amigos tuviéramos que programar en ensamblador, C, o algunos de los modernos lenguajes de alto nivel. Su propósito principal fue hacer la escritura de buenos programas más fácil y placentera. (Bjarne Stroustrup, The C++ Programming Language, Third Edition. 1997).* Citado por (Prata, C++ Primer Plus, p. 14)

En 1983 se le dio el nombre definitivo de C++ a la primera versión operativa usada fuera de los laboratorios. Claramente y como ocurrió en su momento con FORTRAN, COBOL, Pascal y con el mismo C, salió un producto que cumplimentó muchísimo más allá de cualquiera de sus expectativas iniciales, trayendo si no fortuna, al menos fama a su autor.

Se obtiene una mejora incluso si Ud. viene del C y continúa escribiendo en él su código, porque C++ ha cerrado muchos agujeros en el lenguaje y ofrece mejor control de tipos y análisis en tiempo de compilación. Está obligado a declarar funciones de modo que el compilador pueda controlar su uso y la necesidad del preprocesador ha sido prácticamente eliminada para la sustitución de valores y macros, que aportan muchas dificultades para encontrar los errores lógicos.

C++ tiene una característica llamada referencias que permite un manejo más conveniente de direcciones para argumentos de funciones y retorno de valores. El manejo de nombres se mejora a través de una característica llamada sobrecarga de funciones, que le permite usar el mismo nombre para diferentes funciones. Una característica llamada espacios de nombres también mejora la seguridad respecto a C.

En C++ una declaración muestra el nombre y la sintaxis de una entidad: nombre, tipo de dato que retorna y parámetros de uso. Una definición puntualiza el tamaño del almacenaje, cuerpo y contenidos de una declaración, precisando cómo trabaja exactamente. Esto permite respetar las reglas de definición única u ODR (*One Definition Rules*, en inglés), que establecen:

1. En un fichero-fuente habrá al menos la definición de una entidad.
2. En un programa habrá exactamente una definición para cada objeto[9] usado, con la excepción de las funciones **`inline`**, que estarán idénticamente definidas en cada fichero que las use.
3. Un programa puede tener múltiples definiciones de una entidad, siempre que estén dadas en ficheros-fuente radicados en espacios de nombres diferentes.

3 - EL PRIMER PROGRAMA

> *El primer escollo es alguien tratando de aprender mucho, muy rápido. Por lo tanto, codifique en piezas pequeñas y tómese su tiempo. (Linux Tricks and Tips, pág. 64)*

Se comenzará con el habitual y muy sencillo primer programa de saludo al nuevo mundo que nos presenta el C++.

Programa 1: hola mundo

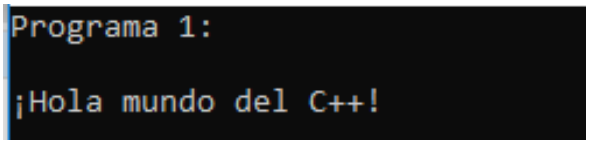
El programa de apertura que acá se brinda es una versión española del original en inglés, nombrado *Hello world*, que fuera traído a la luz por Brian Kernighan y Dennis Ritchie en su ya clásico texto *The C Programming Language* y a partir de allí, por fuerza de la costumbre, se hizo el primer programa en cualquier libro usado para aprender cualquier lenguaje de computación. Y no será éste la excepción.

Programa

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main() {
    cout << "Programa 1:\n\n\255Hola mundo del C++!\n\n";
    system ( "pause" );
    return EXIT_SUCCESS;
}
```

[9] Un objeto en este contexto no se refiere a la OOP, si no a un elemento propio del lenguaje C++

Salida



Notas:

- La primera línea la puso **`cout`**. La segunda la puso **`system("pause")`**.
- El sistema pone una nueva línea cada vez que el escape **`newline`** (nueva línea en español) se escribe dentro de un literal —texto entre comillas: **`"\n"`**— o como secuencia de escape —texto entre apóstrofes: **`'\n'`**. La secuencia de escape **`\255`** pone una apertura de exclamación (**`¡`**) en consola.
- La llamada del comando DOS **`system("pause")`** pone en consola la llamada de atención **`Presione una tecla para continuar . . .`** y detiene la ejecución hasta que el usuario pulse cualquier tecla. El mensaje puede ser en otro idioma, en dependencia de la versión de Windows en uso, y aun cuando está ahí, se suprime en la salida, por ser redundante.
- Con la instrucción **`return EXIT_SUCCESS`** (salida con éxito —también se suele poner **`return 0`**) se le dice al SO que la ejecución terminó bien. Si no es el caso, usualmente se escribe **`return EXIT_FAILURE`** (salida con fallo —también se suele poner **`return -1`**) y aunque estas instrucciones de retorno se pueden obviar, no se recomienda hacerlo.
- Los caracteres constantes simples (también se consideran literales) se ponen entre apóstrofes (por ejemplo, **`'a'`**), pero algunos de ellos (principalmente el retorno —*return* en inglés) no se pueden pasar al código a través del teclado.

C++ reconoce algunas secuencias de escape que comienzan con la barra contra-inclinada y siguen con un carácter.

Tabla 2. Principales secuencias de escape

Escape	Nombre	En español	Significado
`\b`	back space (**`bs`**)	Retroceso	Retrocede un carácter.
`\n`	new line (**`nl`**)	Nueva línea	Emite una nueva línea.
`\r`	carriage return (**`cr`**)	Retorno del carro	Pone el cursor en la primera columna.
`\t`	Tabulator (**`tab`**)	Tabulador	Permite la tabulación horizontal.
`\\`	Backslash (**`bs`**)	Barra contra inclinada	Emite una barra contra inclinada simple
`\”`	double quote	Comilla doble	Emite una comilla doble.

Nota: en la era de las máquinas de escribir, al bajar la palanca y retroceder hacia la extrema izquierda el rodillo donde estaba enrollado el papel, se disponía de una línea nueva para seguir escribiendo y se retornaba el carro (el rodillo) a la primera posición de escritura. Y de ahí viene este legado de **`nl/cr`**. También **`bs`** era una tecla que retrocedía el rodillo en un espacio.

Ahora se verán varios tópicos asociados a la escritura de un programa.

Errores

Independientemente de los múltiples errores que se pueden encontrar en el código de una aplicación, existen tres grandes categorías que engloban a la mayoría de aquellos:

1. Error en tiempo de compilación. Es el más “deseable” de todos. Aparece cuando el programador viola una característica propia del lenguaje. Es de lo más molesto, porque le obliga a detenerse y eliminarlo antes de poder seguir, pero lo mejor que tiene es que el compilador le ayuda en su trabajo de depuración, y lo hace mejor que los humanos. Error advertido por el compilador no puede pasar de categoría y ya subsanado, el desarrollo avanzará normalmente, lo cual lo hace preferible frente al error de ejecución.

2. Error en tiempo de ejecución. También llamado *bug*, aparece durante la ejecución y se produce porque el programador violó de alguna suerte la lógica del programa[10]. Es de lo más peligroso y desagradable, ya que puede estar escondido por largo tiempo y aparecer cuando menos se le espere. Puede ir desde una pequeña molestia hasta la terminación catastrófica de la ejecución y a veces es muy, pero que muy difícil de encontrar y eliminar. Existen programas llamados depuradores (*debbuguers* en inglés) que ayudan en estos trances; el entorno integrado de desarrollo que aquí se usa facilita uno, pero su estudio queda fuera de límites…por ahora.
3. Error de enlace. A veces hay pérdida de sincronismo entre la fase de compilación (cuando el compilador traduce el código que escribió el programador al leguaje de máquinas de la PC) y la de enlace (cuando el enlazador adiciona el código de las bibliotecas y rutinas incluidas), y se produce este error. El compilador avisa, y es relativamente fácil de encontrar y de eliminar…bueno, quizás. Una solución rápida e informal, en lo adelante solución QAD (*Quick-And-Dirty* en inglés), es recomponer (`rebuild`) el código, lo cual subsana el error en muchas ocasiones, y si no… ¡a trabajar!

Documentación de las aplicaciones

En los programas profesionales, una parte del equipo de desarrollo se dedica exclusivamente a preparar la ayuda necesaria para la explotación del producto, que suele venir integrada en el código, y/o en forma de folletos que se distribuyen con su compra, y/o también puesta como enlace de alguna página Web. Un producto profesional carente de ayuda es casi imposible de explotar correctamente, al menos con la eficiencia y eficacia normalmente esperadas.

- ✓ Las ayudas vienen en tres clases: a través del Manual del Usuario, suministrado con la aplicación; en línea con la internet, dónde el usuario se ubica en una página de ayuda del producto; y los comentarios hechos en el programa.

A la par, los demás equipos desarrolladores escriben código que va desde cientos de miles de líneas hasta decenas de millones, las que ciertamente necesitarán en algún momento de su vida útil, operaciones de mantenimiento como son la refactorización, sustitución, ampliación y reducción de código, o la erradicación de *bugs* descubiertos durante su explotación.

Sin la documentación desarrollada durante el proceso de creación, estas operaciones, aun hechas por aquel que las codificó hace no tanto tiempo, e incluso dentro de un módulo relativamente pequeño y siguiendo todas las reglas del juego, muchas veces pueden llegar a ser imposibles de realizar por el grado de ofuscación presente, forzando a una reescritura, con la consiguiente pérdida de tiempo, el aumento de gastos y el consumo de recursos.

- ✓ Dentro de un programa, un módulo es una porción autocontenida de código que realizará una o varias tareas.

Comentarios en el programa

Los Comentarios dentro del código son básicamente descripciones a leer por otros, que detallan lo qué se hace en ese punto. No se ven especialmente importantes, pero un código sin comentarios es una de las muchas áreas de frustración de la programación… (Linux Tricks and Tips, pág. 50)

Una fuente de ayuda con la aplicación y una buena base para el mantenimiento, se consigue mediante la auto-documentación por comentarios, con el añadido de que todo texto comentado es ignorado por el compilador y no contribuye al tamaño final del programa ejecutable. Se advierte al lector que aun cuando los comentarios pueden anidarse bajo ciertas circunstancias (mayormente bajo operaciones de mantenimiento retroactivo al código) no es recomendable el hacerlo durante la creación de código nuevo.

Al crear código, no anidar comentarios; siempre usarlos por separado

[10] Aunque en teoría se puede producir una aplicación de grado comercial sin errores lógicos, al día de hoy se sigue buscando una.

¿Qué es un buen comentario?

Aunque todos concuerdan en que comentar lo obvio sólo consume tiempo, en lo demás no hay consenso. He aquí dos directivas generales que se aplican a un buen comentario:

1. Cada parte constituyente del programa (estructura, clase, módulo, etc.) debe ser comentada en su propósito principal, en los parámetros que espera y en la posible salida que proporciona. El benevolente lector comprenderá por la limitada extensión del texto, que casi nunca se podrá presentar este tipo de comentario.
2. Cada función debe ser comentada, pero no en lo que se hace en cada línea del código, sino en su objetivo, sus parámetros y su valor de retorno, porque en una revisión posterior puede cambiarse el algoritmo y no reflejarse en el comentario, con alta probabilidad de confundir, o peor aún, llevando a equivocarse sin percatarse de ello a quien lo leerá en ocasiones posteriores, durante una revisión de control de calidad, o en un mantenimiento programado. Pero en el texto, que se precia de ser introductorio, se verá mucho el comentario por líneas de código

Comentario de una línea

Los comentarios de una línea comienzan con el par de símbolos de la barra inclinada o *slash*, en inglés

```
// que cuentan como uno hasta el fin de la línea donde aparecen
```

Mayormente se usan dentro del código, para ilustrar elementos afines a una función o estructura cualquiera, o para breves comentarios sobre una instrucción o partes obscuras de código, y a veces, como encabezado de una función, sustituyendo al comentario de múltiples líneas. Por ejemplo,

```
#include <cstdlib> // para system
```

El amable lector comprenderá que, por la naturaleza didáctica del texto introductorio, el comentario de una línea se hará quizás muchas veces más de lo debido, en un intento por aclarar al estudiante el código presente, caso ineludible de *haz lo que bien digo y no lo que mal hago*.

Comentario de múltiples líneas

Los comentarios de múltiples líneas comienzan con el par de símbolos de apertura

```
/* que cuentan como uno, y se extienden
   hasta que topan con el par de cierre */
```

Se usan principalmente para ilustrar elementos afines a la creación del código y su mantenimiento, así como para poner el seudocódigo de algoritmos no triviales, empleados en un módulo. Usualmente van al inicio de un programa, módulo o función. Por ejemplo,

```
/*
   main.cpp
   Programa     : LSE en C++
   Por          : Simón Galliano
   Módulo       : programa principal
   Descripción  : aplicación de la estructura avanzada de datos
                  llamada lista, en su variedad simplemente enlazada.
   Incluye enfoques sobre:
      Uso de la tienda gratis
      Interfaz manejada por menú
      Manejo elemental de errores
*/
```

En el texto se suprimen datos redundantes tales como la descripción del programa, el autor y las fechas

Estilos de uso de los comentarios

El añadir comentarios también forma parte del estilo del programador: por ejemplo, y para que el lector se forme una idea, he aquí dos estilos que se usan con frecuencia. Hay quien usa los dos tipos abajo mostrados y hay quien usa uno solo en múltiples patrones.

```
//----------------------------
// programa 1: ¡Hola mundo!
// Autor: S. Galliano.
// Fecha: 11/05/18; 19:30
// Revisión: 20/12/19; 14:45
//----------------------------
/*********************************************
 * Programa 1: ¡Hola mundo!                  *
 * Autor: S. Galliano, 2018                  *
 * Descripción: el primer programa - ¡Hola mundo! *
 * Fecha: mayo 11, 2018 - 7:30 pm            *
 * Revisión: dic 20, 2019 - 2:45 pm          *
 *********************************************/
```

Preprocesador

Aunque forma parte del sistema C++, en realidad el preprocesador no le conoce. Solamente es eso…un peculiar procesador de textos empleado por el lenguaje, embebido en el sistema, actuando exclusivamente sobre las directivas de preprocesamiento y antecediendo a la compilación de la primera línea de código.

Hay que tener presente que:

- Su sintaxis es distinta de la de C++.
- No puede ir dentro de un bloque.
- Las líneas que comienzan con el carácter **#** son directivas de preprocesamiento y se comunican con el preprocesador. Aunque pueden ser precedidas por uno o más espacios en blanco, la costumbre es poner el carácter **#** en el mismo inicio de la línea, y así se hará en este texto.
- La declaración de una directiva ocupa por completo la línea donde se pone y no lleva el carácter de terminación de instrucción (esto es el *punto-y-coma*.)
- El efecto de una directiva comienza allí donde aparece y continúa hasta el final del fichero, o hasta que su efecto es anulado por otra directiva posterior, presente dentro del mismo.

Formato de inclusión

El estándar ANSI C++ permite para el sistema nombres de ficheros más allá de los clásicos ocho caracteres de las primeras versiones del venerable DOS (*Disk Operating System*, o sistema operativo en disco, en español) y elimina la extensión. Si el fichero proviene de ANSI C y ha sido convertido a C++, se le antepone la letra c y se le quita la extensión. Por ejemplo, en vez de escribir como antes algo así: **`#include <math.h>`**, ahora se escribirá así: **`#include <cmath>`** Las cabeceras propias del ISO C++ no llevan extensión y las bibliotecas clásicas del ANSI C están disponibles con su nombre tradicional.

- ✓ ¡Atención! Los ficheros de las bibliotecas ISO C++ no son lo mismo que sus contrapartes de ANSI C. Usar el viejo sistema **`.h`** da como resultado el tener que obviar plantillas y orientación a objetos, ya que C no les conoce. Omitiendo la extensión se trabaja C++ con todos sus adelantos y la oportunidad de programar en súper C.

En el sistema del autor las bibliotecas ISO C++ están en **`C:\Program Files (x86)\CodeBlocks\MinGW32\lib\gcc\x86_64-w64-mingw32\8.1.0\`**. De más está decir que del manejo de los caminos del DOS se encarga automáticamente el IDE. No hay que preocuparse mucho por ello.

Podrían surgir problemas sumamente difíciles de encontrar y erradicar si se intenta mezclar ambas formas de inclusión en un mismo programa. ¡No debe hacerse nunca! En código nuevo usar siempre la forma moderna de inclusión.

En los sencillos programas de ejemplo, casi siempre se usarán como mínimo estas dos:

```
#include <cstdlib>  // para usar las instrucciones "system" y "cls"
#include <iostream> // para operaciones de E/S a consola
```

Orden de inclusión

John Lakos en *Large-Scale C++ Software Design*, Addison-Wesley, 1996, citado por (Eckel, pág. 529), plantea que los ficheros de cabecera se deben de incluir de lo particular a lo general, siguiendo este orden:

1. Cualquier fichero de cabecera codificado por el programador en el directorio local, e invocado entre comillas dobles, respetando el hecho de que para usar en C++ un elemento cualquiera, primero debe ser declarado.
2. Si las hubiere, las "herramientas" compuestas por código propiedad de la casa productora de software a la que se trabaja, y que traen consigo su propia ubicación. En este texto Code::Blocks no pone nada.
3. Si las hubiere, las cabeceras de bibliotecas de terceros, que consisten en paquetes especializados, usualmente para trabajos gráficos o concebidos específicamente para un área científico-técnica dada, y que también traen su propia ubicación. En este texto no se usaron bibliotecas de terceros.
4. Las cabeceras de la biblioteca estándar del ISO C++, invocadas entre paréntesis angulares y suministradas por el entorno de desarrollo o IDE.
5. Las cabeceras de la biblioteca estándar ANSI C, también suministradas por el IDE, e invocadas entre paréntesis angulares si es que se usan, pero siempre recordando que (**a**) no deberían mezclarse con las del C++; (**b**) no deben usarse en código nuevo. Aquí no se usaron.

Respetar siempre este orden permitirá descubrir prematuramente si el fichero de cabecera no es coherente porque contiene declaraciones o definiciones externas, o dobles, etc. y se prevendrán disgustos futuros.

Directivas

Son órdenes que se les da al preprocesador. Si las reconoce, las ejecuta y si no, las ignora.

Directiva #pragma

Está determinada por cada implementación y se utiliza por la casa creadora del entorno de desarrollo para definir parámetros propios que controlen específicamente su compilador. Debido al fuerte enlace con el sistema, estas directivas deben evitarse tanto como sea posible, evadiendo el comprometer la portabilidad. En el texto no se usan.

Directiva #include

Le dice al preprocesador que abra el fichero indicado e incluya su contenido en ese mismo lugar. Un **`#include`** puede indicar un fichero de dos maneras:

1. Mediante paréntesis angulares. Se aplican a ficheros del sistema ISO C++ (por ejemplo, **`#include <iostream>`**) y hacen que el preprocesador los busque dentro del IDE que se está usando. Están localizados en subdirectorios tipificados, o predefinidos por el entorno desarrollador.
2. Mediante comillas dobles. Se aplican a los ficheros codificados para el problema en desarrollo (por ej., **`#include "menu.h"`**) y le dicen al preprocesador que le busque de forma relativa al directorio actual. Si no lo encuentra, entonces la directiva es procesada como si tuviera paréntesis angulares en lugar de comillas dobles.

Cumplir con estas reglas aun cuando parezca que funcione incluir todo entre comillas dobles

Directiva using namespace std

No basta con integrar una característica en el código; el sistema tiene que reconocerla y tenerla en el ámbito de ejecución:

1. Reconocerla. Para poder usar la característica hay que incluir la biblioteca que la contiene —C++ *es terso*. Originalmente todos los componentes de la biblioteca estándar de C++ estaban en ámbito global, pero la norma los pasó para el espacio de nombres **`std`** —el cual, por cierto, contiene varios centenares de ellos— por lo que algunos programas realmente antiguos no compilarán hogaño si no se arregla este lapso.
2. Tenerla en el ámbito actual de ejecución del código (*in code scope*, en inglés.). Mediante la directiva **`using namespace std`** se pueden usar directamente todas las funciones de la biblioteca estándar de C++, incluyendo **`cin`**, **`cout`** y **`cerr`**, que están especializadas para las operaciones de E/S.

4 - ENTRANDO NÚMEROS

> *C++ utiliza la E/S con seguridad de tipos. Cada operación de E/S se ejecuta de una manera sensible al tipo de datos. Si se ha definido una función miembro de E/S para manejar un tipo de datos específico, entonces se hace una llamada a esa función miembro para manejar ese tipo de datos. Si no hay coincidencia entre el tipo de los datos actuales y una función para manejar ese tipo de datos, el compilador genera un error. Por ende, los datos inapropiados no pueden "infiltrarse" por el sistema. (Deitel & Deitel, págs. 649-650)*

Un programa de computación da solución a algún problema que aparece en cierto ámbito. Para completar la solución, usualmente se requiere conocer de algunos datos que por ahora se reducen a dos categorías: números de cualquier tipo y palabras o frases referentes al problema. A estos saberes se les llama datos. La respuesta obtenida es la información que interpretada correctamente da solución al problema.

La organización del trabajo es primordial. Por ahora se dividirá en tres grandes partes: la entrada de datos, el cálculo de la respuesta y la salida de la información de forma fácilmente legible. Es una técnica de diseño de programas llamada Top-Down que será explicada con más detalle en la página 73 y aplicada explícitamente en el programa dado en la página 79.

Variables

Son direcciones de memoria, representadas por números, donde se guardan datos. Identificadas (declaradas) con nombre y tipo, el sistema las sigue mediante sus direcciones. Si se quiere, se les puede asignar un valor de inicio. Se dicen variables porque durante la ejecución del programa pueden cambiar sus valores previamente asignados. Entonces, una variable presenta cuatro características que deben ser gestionadas de común acuerdo entre cualquier lenguaje de programación y el programador que le usa:

1. Una identificación o nombre legal en el lenguaje.
2. Un tipo de dato legal, también en el lenguaje.

3. Un valor que es cambiable según requerimientos del programa.
4. Una dirección (un número) de memoria que depende del SO, y que en C++ se puede manejar por código, mediante la aplicación de punteros, a ver en la página 127.

Por ejemplo, `float ancho = 25.4` estipula una variable identificada por el nombre de `ancho`, de tipo `float`, que al momento de ser declarada se le asigna de inicio el valor `25.4` (aunque puede ser cambiado posteriormente) y que posee una dirección (número) en el espacio de la memoria que el SO le facilita a C++, y que el programador puede manejar directamente usando punteros o referencias, técnicas que se verán en su momento.

¿Dónde declararlas?

En C++ se acostumbra a declarar las variables lo más cerca posible de donde será su primer uso, excepto cuando trabajan dentro de ciclos, pues si este es el caso, siempre se deben declarar fuera de los mismos. No es una costumbre; detrás de ello se esconde algo más que se verá en su momento y se recomienda al programador novel el apegarse a la misma, ya que hace la posterior lectura e interpretación del código más clara.

En otros lenguajes, señaladamente ANSI C, ANSI FORTRAN, ANSI Cobol y ANSI Pascal, se sigue un patrón rígido de declaraciones en ciertos lugares del código. C++ es mucho más flexible y cómodo en ese sentido.

Declarar las variables lo más cerca posible del lugar donde las emplearán por primera vez

¿Cómo nombrarlas?

Es muy recomendable utilizar nombres autodocumentados para las variables. Por ejemplo, si una variable representa el total anual de ventas, su identificador debería ser algo así como `totalVentasAnuales`, `total_ventas_anuales`, etc. y no `tva` ¡o mucho menos `x`!

C++ presenta un juego obligatorio de cinco reglas para formar identificadores sintácticamente legales, y en su orden son:

1. Un identificador sólo puede contener caracteres alfanuméricos y el carácter de subrayado (_), pero su primer carácter no puede ser un número.
2. Un carácter en mayúsculas es tratado como diferente del mismo en minúsculas, y viceversa.
   - ✓ *case sensitiveness* en inglés, es la propiedad de un lenguaje de considerar si una letra es mayúscula o minúscula. Por ejemplo, C++ es sensitivo al tamaño de las letras y Pascal no. Según lo expuesto, en Pascal los identificadores `MINUMERO`, `miNumero` y `minumero` representan a una misma variable, pero en C++ representan a tres variables diferentes. ¡Mucho cuidado!
3. Una palabra reservada no se puede tomar como identificador.
4. No hay límites al largo del nombre. Por razones de compatibilidad retroactiva, portabilidad y claridad, los nombres de identificadores deben ser razonablemente cortos. Se recomienda mantener el tamaño máximo de los nombres alrededor de los 10–12 caracteres, pero sin olvidar el Zen de Python: *lo práctico gana a lo puro*.
5. Se reservan los nombres de identificadores que comienzan con uno o dos subrayados. Siempre se debe respetar esa reserva, ya que esas variables pertenecen al espacio de nombres que es patrimonio del desarrollador del compilador que se está usando, territorio prohibido, donde nadie debiera entrar. ¡No usarlos!

Estilo de nombrado

Nombrar significativamente a las variables es magnífico para eliminar errores comunes de codificación. Usando letras no está mal, pero ¿qué pasa si el compilador reporta un error con la variable x? No es muy difícil nombrar las variables como ciclos, pago, jugador1, etc. (Linux Tricks and Tips, pág. 64)

El compilador C++ requiere que le nombren sus variables siguiendo las cinco reglas sintácticas anteriores, pero aplicar a la formación del nombre un esquema consistente, claro y preciso, le servirá muy bien a cualquier código. En el texto se usa mayormente el llamado estilo de *notación camello*, que dicta:

1. Los nombres de variables siempre comienzan con al menos, un carácter en minúsculas.
2. Si está compuesto por varias palabras, todas van juntas, la primera palabra va en minúsculas y el resto se capitaliza: `totalVentasAnuales`
3. Los nombres de constantes van en mayúsculas: `const PI = 3.14;`
4. Los nombres de datos compuestos van con mayúscula inicial: `struct Registro {...};`

Notas:

- Ocasionalmente un nombre en notación camello se ve feo (*bello es mejor que feo*) o no se entiende bien (*la legibilidad cuenta*.) En ese caso se recurre al guion bajo para separar las palabras. Por ejemplo, `ficha_1`, `entra_o_sale`, etc. Es el estilo preferido de los programadores de ANSI C.
- Otro estilo que también usa el texto, sobre todo para trabajar con punteros, es conocido por *notación húngara*, por su inventor, el húngaro Charles Simonyi, y añade información extra consistente en preceder el nombre de la variable con una letra o palabra corta que describa su tipo: `i (int)`; `f`, `flt (float)`; `d`, `dbl (double)`; `ch`, `chr (char)`; `p`, `ptr (pointer)`; `r`, `ref (& referencia)`. Por ejemplo, para identificar un puntero, en el texto se usa algo así como: `*pVentas`, `*ptrVentas`, etc. También se ve mucho en ANSI C.

El programador emplea los nombres de variables **(a)** que mejor le gusten; **(b)** que mejor se ajusten al problema, según sus necesidades, preferencias o viejo estilo que trae de otros lenguajes que ya domina; o **(c)** que mejor se adaptan a las necesidades de su equipo de trabajo, normas estatales, ramales, provinciales, empresariales, de fábrica, etc. Pero siempre será consistente en su escritura a lo largo de una aplicación.

Debe haber consistencia en todo el código escrito para una aplicación

---

Palabras reservadas de C++

Una palabra reservada o palabra-clave, es un nombre que el lenguaje gestiona, el sistema se las reserva y el programador siempre debe emplearlas en su contexto. Por ejemplo, la palabra-clave `and` es un operador lógico (booleano) y sólo puede usarse en operaciones lógicas. Sigue un listado escogido de palabras-claves.

Tabla 3. (simplificada) Palabras reservadas en C++

`and`	`delete`	`if`	`public`	`this`
`auto`	`do`	`inline`	`register`	`throw`
`bool`	`double`	`int`	`reintepret_cast`	`true`
`break`	`dynamic_cast`	`long`	`return`	`try`
`case`	`else`	`namespace`	`short`	`typedef`
`catch`	`enum`	`new`	`signed`	`typename`
`char`	`extern`	`not`	`sizeof`	`union`
`class`	`false`	`not_eq`	`static`	`unsigned`
`const`	`float`	`operator`	`static_cast`	`using`
`const_cast`	`for`	`or`	`struct`	`virtual`
`continue`	`friend`	`private`	`switch`	`void`
`default`	`goto`	`protected`	`template`	`while`

Algunas variables predefinidas

Aunque el tema de las variables se vio antes, se aclara que todos los preprocesadores C++ reconocen al menos las siguientes tres, propias del sistema:

Tabla 4. (simplificada) Variables predefinidas para uso del compilador

Nombre	Acción	Expande al literal
__FILE__	fichero en compilación	Su forma final depende del IDE que le gestiona
__DATE__	fecha de su compilación	nombre abreviado del mes, día, año
__TIME__	hora de su compilación	hora_militar:minuto:segundo

Programación secuencial

En la programación moderna, visual, las entradas y salidas están condicionadas a la generación de eventos por parte del usuario, comúnmente al clicar con el ratón en orden caótico por toda el área de la consola, en botones situados dentro de cajas visuales llamadas…ventanas (*windows* en inglés, por Windows™), que están dispersas en disímiles posiciones. Por eso a esta forma de trabajo se le suele llamar *Ejecución Conducida por Eventos*. Se dice que el control del programa pertenece al usuario.

En la programación antigua, también llamada *Ejecución Secuencial*, las entradas interactivas son por teclado y las salidas van a consola —a veces van a ficheros o a bitácoras. Esas operaciones se van sucediendo según aparecen en pantalla, por lo que hay que planificarlas con lógica y cuidado. Se dice que el control del programa pertenece al programador.

El texto usa exclusivamente la ejecución secuencial

Tipos básicos de datos

El sistema de números reales, visto como construcción matemática, es infinito, pero su capacidad de representación en una computadora no lo es. Cada uno de estos componentes —enteros y de punto flotante (reales)— tienen en la PC valores límites, que son fijados según el compilador, el SO y el lenguaje de programación. En C++, en este SO, bajo este IDE y en este equipo son:

Tabla 5. Tipos básicos de datos C++ (tamaño mínimo en bytes)

Tipo	Descripción	Tamaño
void	Tipo no asociado con dato alguno.	1
bool	Valor booleano: **false** (cero), o **true** (distinto de cero)	1
char	Número del código (normalmente ASCII) de un carácter.	1
float	Número en punto flotante de simple precisión.	4
double	Número en punto flotante de doble precisión.	8
long double	Alberga límites superiores para doble precisión.	12
int	Número entero.	4
short	Número entero de corto alcance.	2
long	Al menos es del mismo tamaño que un entero.	4
long long	No es menor que un tipo **long**.	8

Como se puede ver, excepto **void**, todos los demás tipos básicos son (o pueden ser) representados como números. Más adelante, en las págs. 46-53, se verá cómo interrogar a cualquier sistema acerca de sus límites. Los calificadores aplicables

en C++ a los datos básicos son **`signed`**, **`unsigned`**, **`short`**, y **`long`**. Cuando uno de estos calificadores de tipo es usado por sí mismo, se asume que actúa sobre un número entero. Así pues, estas declaraciones son equivalentes:

```
unsigned int i; // int está especificado
unsigned i;     // int está implícito
```

Tabla 6. Posibles tipos básicos cualificados (cualificadores omisibles entre corchetes)

`int`	`float`	`double`	`char`	`bool`
`unsigned [int]`	`signed [int]`	`short [int]`	`unsigned short int`	
`unsigned long int`	`signed long int`	`long [int]`	`signed short int`	
`unsigned char`	`signed char`	`long double`		

Operadores aritméticos

Los operadores de C++ son múltiples, variados y se irán viendo a medida que se avanza. En la tabla que sigue se muestran los operadores aritméticos.

Tabla 7. Operadores aritméticos

Op	Operación	Nombre	Propósito. Para los ejemplos, `x = 10; y = 3`
`+`	`x + y`	Suma	Suma ambos operandos. Por ejemplo, **x + y = 13**
`-`	`x - y`	Resta	Resta ambos operandos. Por ejemplo, **x - y = 7**
`*`	`x * y`	Producto	Multiplica ambos operandos. Por ejemplo, **x * y = 30**
`/`	`x / y`	División	Divide ambos operandos. Por ejemplo, **x / y = 3.3333**
`%`	`x % y`	Módulo	Resto de la división de enteros. Por ejemplo, **x % y = 1** Si aplica este operador a números reales, habrá error de compilación.
`++`	`x++` `++x`	Incremento	Incrementa el operando en una unidad. Por ejemplo, si se hace **++x**, ahora x vale 11.
`--`	`x--` `--x`	Decremento	Disminuye el operando en una unidad. Por ejemplo, si se hace **--x**, ahora **x** vale 9.

Nota: los cinco primeros son llamados operadores binarios, porque manejan dos términos u operandos: uno a su izquierda y el otro a su derecha. Los dos restantes son unarios porque actúan sobre un operando, ya sea a la izquierda, ya a la derecha.

Programa 2: dos operaciones de E/S

La entrada del usuario es a menudo un error paralizante en el código. Por ejemplo, cuando al usuario se le pide la edad como un número, pero él la entra con letras. O cuando entra tanto que desborda algún búfer interno y el programa colapsa. Observe esas entradas y explicite claramente que se espera en ellas. (Linux Tricks and Tips, pág. 65)[11]

En este segundo programa de ejemplo se usan comentarios y algunas de las variables predefinidas por el preprocesador; se ve cómo entrar normalmente los números; se realizan dos operaciones aritméticas simples; y se utilizan dos formas de salida a consola.

Programa

```
#include <iostream>
#include <cstdlib>
using namespace std;
```

[11] Eso se hará en el texto…cuando sea absolutamente necesario.

```
int main() {
    // breve instrucción para el usuario
    cout << __FILE__ << '\n';
    cout << "Programa 2: toma dos enteros, los suma y resta.\n\n";

    // identificadores autodocumentados
    int entero_1;
    cout << "Entre el primer entero: ";
    cin >> entero_1;
    int entero_2;
    cout << "Entre el segundo entero: ";
    cin >> entero_2;

    // declaración e instrucción juntas
    int suma = entero_1 + entero_2;

    // salida con variable
    cout << "\nLa suma es: " << suma << '\n';

    // salida directa
    cout << "La diferencia es: " << entero_1 - entero_2 << "\n\n";

    // fin del programa
    cout << "Fecha de ejecuci\242n: " << __DATE__ << ", " __TIME__ << '\n';
    system ( "pause" );
    return EXIT_SUCCESS;
}
```

Salida

```
E:\Texto\Tomo 1\Programas\p02-dos operaciones de E&S\main.cpp
Programa 2: toma dos enteros, los suma y resta.

Entre el primer entero: 12
Entre el segundo entero: 5

La suma es: 17
La diferencia es: 7

Fecha de compilación: Nov 12 2020, 07:19:25
```

5 - ENTRANDO CADENAS

> *Las operaciones de E/S son fundamentales en un programa estándar de computación*[12]*: en la inmensa mayoría de los casos puede carecer totalmente de salida a consola o puede carecer totalmente de entrada activa, pero su utilidad estará severamente limitada. El Autor.*

Muchos de los ejemplos aquí presentados carecen de entrada activa, pero eso se debe: **(1)** a la limitación de espacio del texto; **(2)** por sus objetivos netamente pedagógicos: están ahí para enseñar cosas que no tienen que ver con estas operaciones.; y **(3)** por su fuerte contenido didáctico: están ahí para demostrar cosas que no son estas operaciones.

[12] Existen programas aplicados en muchas áreas técnicas que toman la entrada de sus datos de sondas automáticas, o envían sus resultados a periféricos especializados.

Las E/S se hacen con números, literales, caracteres y conjuntos de caracteres en secuencia acotada (conocidos en español simplemente como cadenas.) Aunque una palabra se alberga sin problemas en una variable de caracteres lo suficientemente grande, usualmente una cadena de caracteres está diseñada para contener una frase limitada hasta llegar al símbolo de *fin-de-cadena* **\0** y formada como mínimo por dos palabras separadas entre sí por cualquier elemento que C++ considere como espacio en blanco: una tabulación, un pulso de la barra espaciadora, etc.

Antes se vio la entrada numérica. Seguidamente se verán las formas correctas de manejar una cadena de caracteres. Este tipo de dato —que en el texto será llamado como cadena clásica— es heredado del lenguaje ANSI C, donde se define como un arreglo[13] de caracteres cerrado por el símbolo **\0**. Un carácter es una forma especial de ver un número, mediante el mapeo entre un símbolo y un entero.

- ✓ Casamiento o mapeo (anglicismo de *mapping*), en este contexto es dar un valor al programa y dejar que el compilador elija automáticamente otro que le está asociado. El mapeado más común pertenece al código ASCII donde el entero decimal 97 u octal \141, es pareado con el carácter que representa la letra minúscula a.

Tabla 8. (simplificada) Funciones de la clase <cstring> para cadenas ANSI C

Función	Significado
`char* strcpy(char *destino, const char *fuente)`	Copia la fuente en el destino y retorna el destino.
`int strcmp(const char *cad1, const char *cad2)`	Compara **cad1** con **cad2** y retorna **1** si **cad1** es mayor, **-1** si **cad1** es menor y cero si son iguales.
`char* strcat(char *destino, const char *fuente)`	Añade fuente a lo que está en destino y retorna el destino.
`size_t strlen(const char *cadena)`	Calcula el tamaño de la cadena y retorna su tamaño real.
`int atoi(const char* cad)` `double atof(const char* cad)`	Toma una cadena de números y retorna su valor entero (**atoi**) o de doble precisión (**atof**), según sea la función.
`char* strtok(char *str, const char *delim)`	Separa una cadena ANSI C **str** en tokens* aislados por el carácter delimitador **delim** y retorna una cadena ANSI C con el token recién aislado.

Nota: un token es un componente léxico del lenguaje, que aquí significa una palabra extraída de una cadena.

Entrando una cadena clásica de caracteres

Cuando se declara una variable de tipo **char** y seguidamente se pone un número entre corchetes, automáticamente se está en presencia de una cadena clásica de caracteres. Las funciones que actúan sobre cadenas clásicas se ponen en ámbito con **#include <cstring>**. Su declaración e inicialización es como se muestra:

```
    char cad[7];                // variable de cadena con capacidad nominal = 7
1.  strcpy(cad, "Hola");        // la asignación de su contenido – preferido
2.  const char cad[7] = "Hola"; // todo en una sola operación - a veces
3.  char cad[7] = "Hola";       // todo en una sola operación – desaprobado
```

Con este último tipo de declaración, este compilador se queja, marcándola como desaprobada (*deprecated* en inglés), elemento del lenguaje cuyo uso no se recomienda ya sea porque hay una versión superior, ya porque los desarrolladores de C++ y de sus normas se reservan el derecho a eliminarlo sin previo aviso. Esta forma es para uso exclusivo de cuestiones de compatibilidad retroactiva o mantenimiento de código POD, pero debe ser desterrada de todo código nuevo.

Si el conjunto de caracteres no está cerrado por **\0**, no es una cadena de caracteres; es un arreglo simple de caracteres que no puede tratarse con las funciones destinadas por C++ para las cadenas.

[13] El arreglo es un dato estructurado que se verá más adelante.

La cadena que se muestra en la ilustración fue declarada con capacidad nominal de 7 caracteres, pero su tamaño real máximo es de hasta 6, porque el séptimo es el que la cierra, que puede hacerlo antes que la cadena esté totalmente llena, como muestra la figura. En las cadenas, los caracteres que siguen después del cierre no tienen sentido y se conocen como basura (*trash* en inglés) ya que el último que se acepta es obligatoriamente el carácter especial de cierre **`\0`**.

Si se puso el dato "Hola" en esa variable, su capacidad nominal no varía, pero el tamaño real sólo es de 4 caracteres (H-o-l-a), pues es lo que cuenta la función **`strlen()`**, que no incluye el cierre.

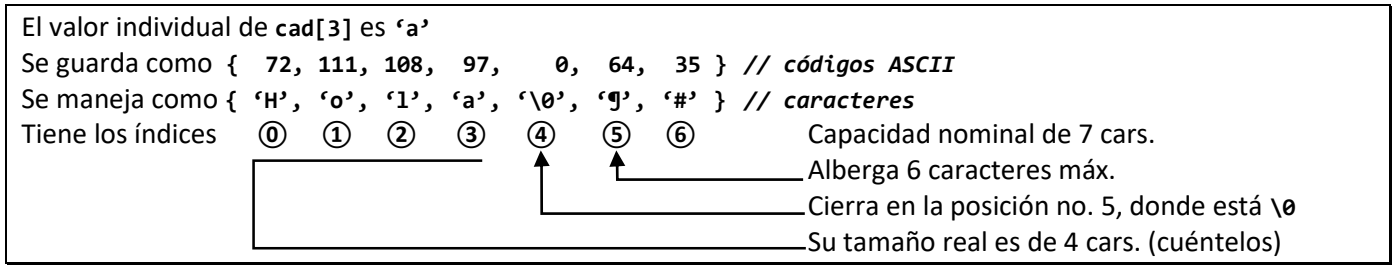


Ilustración 2. Elementos de una cadena clásica

El índice de un arreglo cualquiera numérico o no, invariablemente es positivo, comienza en cero y es un número entero como tal, o como resultado de un cálculo. Por ejemplo, el índice de este arreglo de siete elementos va de cero a seis. Esto pasa siempre y se le denomina *indizado a cero*, característica absoluta heredada por el C++.

La instrucción **`cin >> saludo`** puede entrar varios caracteres, pero no una cadena, ya que lee carácter a carácter hasta llegar a un espacio en blanco (o lo que C++ considera que lo es), dejando los caracteres restantes y el retorno en el búfer.

- ✓ Búfer: área de memoria reservada para almacenar temporalmente datos.

Muchos periféricos poseen búferes. En especial el teclado hace eco de las pulsaciones a consola, pero guarda el dato en su búfer y no lo envía al programa hasta que el usuario pulsa retorno. Si hay que leer un número con **`cin`** antes que una cadena, luego de la lectura hay que vaciar el búfer (*to flush the buffer* en inglés), ya que **`cin`** no lo hace y más tarde, al leer la cadena, lo primero que lee es el retorno que ya está en el búfer, pasando por alto la entrada. Una forma QAD de vaciar el búfer es, por ejemplo:

```
cin >> edad; // lee el valor y lo pone en la variable numérica edad
cin.get();   // limpia el búfer
```

pero es mejor de escribir y más claro de interpretar si se concatenan (unen) ambas instrucciones:

```
(cin >> edad).get(); // lee el valor, lo pone en la variable edad y seguidamente limpia
```

Si se deben entrar dos o más palabras, **`cin`** entraría la primera y al llegar a un espacio en blanco dejaría el resto de la cadena en el búfer, algo que comúnmente no es lo deseado. Pero **`cin`** tiene una función llamada **`getline()`** que se encarga de entrar toda una línea de caracteres incluyendo hasta el retorno. Su sintaxis es:

```
cin.getline(cadena, capacidad_máxima_a_entrar);
```

Para solicitar el valor de una cadena clásica, usar siempre **`cin.getline()`**

Operaciones sobre caracteres y cadenas clásicas

Las operaciones típicas con cadenas de caracteres (asignación, copia, añadidura, comparación, etc. de dos o más palabras), pueden hacerse en base individual. Están declaradas en el fichero de encabezamiento **`<cstring>`** y siempre se harán con sus respectivas funciones. Las operaciones sobre caracteres están declaradas en el fichero **`<cctype>`** y siempre se deben de hacer con sus respectivas funciones. No usar otras formas para trabajar caracteres y cadenas clásicas, de otra suerte se incurre con facilidad en advertencias y/o errores de compilación.

Por ejemplo, asumiendo **`char cad_1[80], cad_2[80]`**:

1. Nunca se debe leer directamente una cadena clásica.

```
no: cin >> cad_1;            // no en directo: sólo entrará una parte
SI: cin.getline(cad_1, 80);  // usar la función de entrada de línea
```

2. Nunca se deben comparar directamente dos cadenas clásicas.

```
no: if ( cad_1 == cad_2 ) { ... }           // no en directo: es un error de compilación
SI: if ( strcmp(cad_1, cad_2) == 0 ) { ... } // usar la función comparativa
```

3. No se pueden concatenar directamente dos cadenas clásicas.

```
no: nombre = cad_1 + cad_2; // es un error de compilación
SI: strcpy(nombre, cad_1);  // primero se copia una cadena...
    strcat(nombre, cad_2);  // ...y después se le une la otra
```

4. Y aun cuando se puede hacer, no se recomienda probar directamente el diapasón donde pudiera caer un carácter, es mejor usar una función de clasificación:

```
no: if ( n >= '0' or n <= '9' ) { ... }// tratar de no hacerlo así
SI: if ( isdigit(n) ) { ... }          // mejor hacerlo así: menos chance de error
```

5. En la biblioteca **`<cstdlib>`** se define el tipo **`size_t`**, calculado automáticamente por el sistema como el resultado del operador **`sizeof`** y su valor es del tipo **`unsigned`** en alguna forma adecuada al caso. Puede ser usado con las funciones de bibliotecas que requieran de algún tamaño grande de dato, o donde haya que emplear un entero grande.

En la biblioteca **`<cctype>`** están las funciones de clasificación, que actúan sobre caracteres aislados o elementos que forman parte de una cadena C++, como se muestran en la tabla.

Tabla 9. (simplificada) Funciones de la clase <cctype> para cadenas ANSI C

Función	Acción
`b = isalnum(c)`	Devuelve **`true`** si **`c`** es una letra o un dígito
`b = isalpha(c)`	Devuelve **`true`** si **`c`** es una letra
`b = isdigit(c)`	Devuelve **`true`** si **`c`** es un dígito
`b = islower(c)`	Devuelve **`true`** si **`c`** es minúscula
`b = isupper(c)`	Devuelve **`true`** si **`c`** es mayúscula
`b = isprint(c)`	Devuelve **`true`** si **`c`** es un carácter imprimible, incluyendo espacios en blanco
`b = ispunct(c)`	Devuelve **`true`** si **`c`** es un signo de puntuación
`b = isspace(c)`	Devuelve **`true`** si **`c`** es un espacio en blanco
`c = tolower(c)`	Devuelve **`c`** en minúsculas
`c = toupper(c)`	Devuelve **`c`** en mayúsculas

Constantes

Identificadas por el calificador **`const`** al momento de declararlas con nombre y tipo, el sistema las sigue mediante sus direcciones, pero se les tiene que asignar un valor de inicio, porque durante la ejecución del programa ya no se les puede cambiar ese valor. Son puestas en direcciones de memoria especiales.

Es costumbre arraigada en C++ el escribir los nombres de constantes en mayúsculas

Por regla general, una constante se declara en algún lugar del código y luego se usa donde sea necesario. Más tarde, si por motivos de mantenimiento o de otro tipo se requiriera cambiar su valor original, se actualizará donde se declaró.

Incremento unitario

El operador incremental ++ puede aplicarse a cualquier variable numérica (aunque no a una expresión),[14] pero si se aplica a una variable de punto flotante, sólo se incrementa la parte entera. Este operador es el preferido para aumentar el valor de las variables enteras o de caracteres, de uno en uno, ya que el programador escribe menos y el compilador codifica con más eficiencia.

Siempre que sea posible usar esta optimización

Preincremento y posincremento

Cuando el operador se usa como prefijo se está diciendo al compilador: *incrementa primero y luego usa su nuevo valor*: es el preincremento. Cuando el operador se usa como posfijo se está diciendo al compilador: *usa el valor actual y luego increméntalo*: es el posincremento. El operador unitario de disminución (−−) funciona exactamente como su hermano, pero en vez de aumentar el valor, lo que hace es disminuirlo.

Si se desea incrementar en una unidad el valor de una tal variable **`i`**, se puede perfectamente codificar **`i = i + 1.`** Si se desea disminuirla, también se puede codificar **`i = i – 1`**. Pero lo más rápido, eficiente y acostumbrado es usar los incrementos o decrementos unitarios:

- La instrucción **`i++`** usa el valor de **`i`** y entonces lo aumenta una unidad.
- La instrucción **`++i`** aumenta el valor de **`i`** una unidad y entonces lo usa.
- La instrucción **`i--`** usa el valor de **`i`** y entonces lo disminuye una unidad.
- La instrucción **`--i`** disminuye el valor de **`i`** una unidad y entonces lo usa.

Generalmente en cualquiera de las dos variedades de pre, o posincremento, el operador da un resultado correcto, aunque el preincremento es algo más eficiente, pero se advierte: si en una misma línea de código el operador se usa más de una vez sobre la misma variable y en la misma instrucción, el código se ofusca y pueden fácilmente aparecer efectos colaterales que provocan sutiles y molestos errores difíciles de encontrar.

- ✓ Un efecto colateral es un cambio no deseado en el entorno de ejecución del programa. Es la aparición de una acción involuntaria, que no es ni requerida por la tarea que se resuelve, ni pretendida por el programador. Generalmente es dañina a la aplicación, no avisa y usualmente produce falsos resultados o una terminación anormal.

Este caso sucede a menudo cuando se declaran macros que manejan incrementos. Una de las muchas razones porque en C++ se prefiere sustituirlas por funciones **`inline`**. Recordar el Zen de Python: *la legibilidad cuenta*.

El operador condicional ternario

Como se comprende, la programación de computadoras, una de las tareas más retadores y difíciles que jamás ha enfrentado la humanidad, no se puede aprender linealmente. Las técnicas pedagógicas han avanzado muchísimo en este campo, pero aún hay que apoyarse en la retroalimentación o realizar saltos hacia lo ignoto, esperando no hacerse mucho daño al caer, y luego proseguir el aprendizaje en espiral dialéctica: mayores contenidos en cada nueva rama de la espiral, apoyados en los ya adquiridos, avanzando de lo conocido a lo desconocido, de lo simple a lo complejo. No queda de otra…

Por ejemplo, aunque hasta ahora se ignora cómo tomar decisiones, en este momento se dará un imprescindible salto hacia adelante. Suponga el lector la siguiente instrucción:

[14] Una expresión es un conjunto de tokens con significación semántica. Por ejemplo, la expresión **`c = a + b`** está compuesta por 5 tokens: **`a`**, **`b`**, **`c`**, **`+`**, **`=`**.

```
cout << “La igualdad entre su nombre y direcci\242n es “;
(nombre == direccion) ? cout << “verdadera.\n” : cout << “falsa.\n”; // operador ternario
```

El operador ternario se lee como: *“si el valor de la variable* **nombre** *es igual al valor de la variable* **dirección**, *ejecutar la parte del cierre de interrogación* (**?**) *y si no lo es, entonces ejecutar la parte de los dos puntos* (**:**)*”*

Se ha usado el operador condicional ternario, así llamado porque actúa sobre tres operandos (que son expresiones.) Es el único de su tipo en C++ y produce un código muy rápido y eficaz en su trabajo. Como es una sola instrucción, termina con punto-y-coma final. La forma de escritura que este texto recomienda es una de dos, dependiendo del tamaño de las expresiones:

```
1.   ( expresion_de_comparación )
       ? expresion_1_larga
       : expresion_2_larga;
2.  ( expresion_de_comparación ) ? expresion_1_corta : expresion_2_corta;
```

Aunque esta recomendación es totalmente opcional, el programador novel debería acostumbre a poner la comparación entre paréntesis para hacer la instrucción mejor leíble y más entendible. Si se requiere por su complejidad, se sangra. Observar el uso en los ejemplos dados.

Ambas condicionales han de seguir al menos una de las siguientes reglas, pero ante cualquier contingencia el compilador avisa. Las expresiones que pueden manejar son:

- Aritméticas.
- Lógicas.
- De salida.
- Estructuras o uniones compatibles.
- Punteros tipo **void**.
- Lanzamientos de excepciones.

Por su eficiencia se prefiere aplicar este operador a expresiones lo más simples posibles antes que la forma completa del **if**, a ver en la página 51.

Programa 3: la cadena clásica de C

Este programa muestra cómo se mezclan las peticiones de cadenas clásicas y números. Es necesario conocer cómo se hace porque tal vez, en operaciones de mantenimiento retroactivo, o por cualquier otra razón, se tenga que lidiar con este tipo de dato. Después de todo, C++ es el heredero de C. El lector debe fijarse cómo se usan los corchetes.

Programa

```
#include <iostream>
#include <cstdlib>
#include <cstring> // para strcpy() y strcmp()

int main() {
  using std::cout;
  using std::cin;
  // información al usuario
  cout << "Programa 3: un trabajo con cadenas cl\240sicas.\n"
    "Se pedir\240n algunos datos personales.\n\n";

  // máxima cantidad de caracteres en estas cadenas
  const short TAM = 60;
```

```
  // 1) entrada de una cadena clásica
  char nombre[TAM];               // define la cadena
  cout << "Su nombre completo: "; // pide un valor
  cin.getline( nombre, TAM );     // entra el valor

  // 2) entrada numérica: como inmediatamente después de ésta
  //    se va a hacer otra entrada de caracteres, hay que
  //    limpiar el búfer con get()
  int edad;             // define el entero
  cout << "Su edad: "; // pide un valor
  (cin >> edad).get(); // entra el valor y limpia el búfer

  // 3) entrada de otra cadena clásica
  char direccion[TAM];           // define la cadena
  cout << "Su direcci\242n: ";   // pide un valor
  cin.getline( direccion, TAM ); // entra el valor

  // las copias de cadenas clásicas siempre son con asignación
  // diferida; strcpy es una función de manejo de cadenas
  char copiaNombre[TAM];               // primero la variable
  strcpy( copiaNombre, nombre );       // después la asignación
  char copiaDireccion[TAM];            // primero la variable
  strcpy( copiaDireccion, direccion ); // después la asignación

  // la salida
  cout << "\nLa copia de su nombre es " << copiaNombre;
  cout << "\nLa de su direcci\242n es " << copiaDireccion;
  cout << "\nEl a\244o que viene Ud. tendr\240 " << ++edad << " a\244os.";

  // comparación entre cadenas
  cout << "\nSu nombre y direcci\242n";
  //  strcmp es otra función de manejo de cadenas
  ( strcmp( nombre, direccion ) == 0 )  // ¿son iguales?
     ? cout << " son iguales.\n\n"      //  si lo son
     : cout << " no son iguales.\n\n"; //  no lo son

  // fin
  system ( "pause" );  // la salida se detiene
  return EXIT_SUCCESS; // el programa termina OK
}
```

Salida

```
Programa 3: un trabajo con cadenas clásicas.
Se pedirán algunos datos personales.

Su nombre completo: Santos Aldorso Calco
Su edad: 33
Su dirección: Martí #1332 esq. Ave. Palma, pueblo Mocho. Cuba

La copia de su nombre es Santos Aldorso Calco
La de su dirección es Martí #1332 esq. Ave. Palma, pueblo Mocho. Cuba
El año que viene Ud. tendrá 34 años.
Su nombre y dirección no son iguales.
```

Se codificó con secuencias de escape la letra á de las palabras 'clásicas' y 'pedirán', la letra ó de la palabra 'dirección', y la letra ñ, de las palabras 'año/años', pero el carácter extendido entrado por teclado (í) fue mostrado correctamente a la salida, puesto que **`cin`** los lee e interpreta fielmente.

Entrando una cadena de C++

C++ ofrece una forma más segura de usar y más simple de manejar las cadenas. En el fichero de cabecera **`<string>`** hay una clase dedicada a tratar cadenas, que oculta la responsabilidad de su administración, oferta muchos métodos que hacen el trabajo más intuitivo, y permite su tratamiento casi como si fueran variables ordinarias, cubriendo un amplio espectro de posibilidades en sus aplicaciones, aunque desafortunadamente, no todas.

Para los propósitos de esta parte, por ahora el benevolente lector aceptará que un constructor es la forma en que se declara cierto tipo de variable atada a una estructura, y un método ilustra lo que se puede hacer con ella. Esto se ampliará un poco más en la página 116, pero su estudio a fondo se deja para el tomo 3

Tabla 10. (simplificada) Interfaz de la clase <string> para cadenas C++

Constructores	**Resultados**
`string c`	Crea una cadena **`c`** vacía
`[const] string c(texto)` **`[const] string c = texto`**	Crea una cadena **`c`** conteniendo un texto que puede ser o no constante.
Métodos	**Resultados**
`c.assign(texto)`	**`c`** fue creada y ahora se la asigna un texto
`c.at(i)` **`c[i]`**	Accede al carácter iésimo de la cadena con y sin comprobación de límites, respectivamente. La primera forma es menos eficiente en tiempo, pero da mayor seguridad.
`c += texto` **`c.append(texto)`**	Añade texto a **`c`**
`c = c1 + c2`	Concatena (une) en **`c`** a **`c1`** con **`c2`**
`c.clear()`	Vacía la cadena **`c`**
`==, !=, not_eq, <, <=, >, >=`	Compara lexicográficamente dos cadenas
`n = c.size()`	Devuelve en **`n`** el tamaño de la cadena **`c`**
`b = c.empty()`	Devuelve **`true`** si **`c`** está vacía
`getline(fl, cd[, d])`	Lee la cadena **`cd`** al flujo de entrada **`fl`**; si está presente sólo lee hasta el delimitador **`d`**
`f_in >> v`	El flujo de entrada **`f_in`** lee un valor a la variable **`v`**
`f_out << cd`	Escribe la cadena **`cd`** al flujo de salida **`f_out`**
`c = c1.c_str()`	Devuelve el valor de **`c1`** como cadena constante ANSI C
`c.substr(c1)`	asigna a **`c`** la subcadena **`c1`** desde su inicio
`i = c.find(c1);`	Devuelve en **`i`** la posición de **`c1`** si está en **`c`**. Si no está devuelve el valor **`str::npos`**.

Notas:

- **`texto`** significa cualquier expresión textual legal y si es un literal, va entre comillas.
- Siempre usar **`string::size_type`** para el valor de retorno; no usar otro tipo puesto que quizás la comparación contra **`str::npos`** no funcione.
- Si la posición de inicio está fuera de límites, al usar **`c.at(i)`** se lanza una excepción:
  `terminate called after throwing an instance of 'std::out_of_range' what(): basic_string::substr`
- La comparación lexicográfica o de diccionario entre dos textos es como sigue:
  - Se hace carácter contra carácter
  - Los números vienen primero y entran en su orden natural, 1 antes que 2, 11 antes que 2, etc.
  - A viene antes que B, AA antes que AB, etc.
  - A viene antes que a, etc.

- La forma correcta de entrar una cadena C++ es empleando la ya antes vista **getline**, donde ahora trabaja como función-miembro de la clase **<string>**. Lee todos los caracteres incluyendo el de retorno, aunque no lo pasa. Si está presente un carácter como delimitador, sólo lee hasta el mismo. Su interfaz cambió: ahora su primer parámetro es un flujo de lectura **f_in,** su segundo parámetro es el nombre de la variable tipo **string** donde se entra la lectura y su tercer y opcional parámetro es un delimitador tipo **char**.
  - ✓ Interfaz: define la conexión entre módulos, permitiendo el intercambio de información mediante los parámetros puestos entre paréntesis. Su plural es interfaces.
  - ✓ Parámetro: identificador legal definido en la interfaz de una función, que será reemplazado por un valor real, suplido por quién la llama, para que aquella pueda hacer su tarea.

La cadena C++ se alberga en la tienda gratis, su administración corre a cargo del C++ y no tiene longitud fija: en la medida que se necesita, se agranda o reduce para conformarse al nuevo contenido, lo cual es muy cómodo para el programador. Si se requiere en un momento dado, su longitud se toma del método **cad.size()**. La vía del C++ es usar la clase **<string>**, para manejar las cadenas de caracteres.

En código nuevo, siempre tratar de usar las cadenas C++

Utilizando lo necesario

Al cargar todo el **namespace std** el código se abre a la posibilidad de deslizar errores muy difíciles de encontrar cuando hay en juego muchas líneas, pues inadvertidamente se puede llamar sin querer a una de las numerosas funciones contenidas en ese espacio de nombres, que pudiera hacer fallar la lógica del programa. Lo mejor es usar solamente las características deseadas, estrechando la directiva **using** con el operador de ámbito, como se verá de inmediato.

No se recomienda usar directamente el **namespace std** en código nuevo

El operador de ámbito

Cada declaración añade un nombre que la identifica en el ámbito donde se hace, y cada uso de un nombre obliga al compilador a identificar el ámbito que le contiene. Cuando el compilador conoce el ámbito, establecerá qué cosa es el nombre: variable, función, clase, objeto, etc.

- ✓ Ámbito, o *scope* en inglés, es una región del código que contiene alguna declaración.

El ámbito puede ser determinado o anónimo. A veces es el programador quien le dice al compilador cuál ámbito contiene un nombre, pero en un programa típico la mayoría de los nombres son anónimos y el compilador es quien determina el significado del nombre. Cuando el programador es quien quiere o debe decirlo, usará el operador de ámbito dobles-dos-puntos **(::)** que cuentan como un solo símbolo.

Las clases y los **namespaces** definen ámbitos determinados, mientras que los bloques (código entre llaves), funciones y **namespaces** anónimos los definen...anónimos, significando que no pueden ser cualificados con el operador de ámbito.

Seguidamente se muestra un programa que sólo necesita conocer dónde están las facilidades **string**, **cin**, **cerr** y **cout** (que como ya se sabe están en el espacio **std** de nombres), por lo que su ámbito se estrecha con estas directivas:

```
using std::cout;   // operación de salida
using std::cin;    // operación de entrada
using std::cerr;   // salida a consola por error cometido
using std::string; // uso de la clase string
```

El ámbito de las directivas es de fichero, es decir, si se pone una directiva **using** en un fichero, es válida hasta que la ejecución llega al fin del mismo, por eso es seguro ponerlas en los ficheros de declaraciones, a ver en la página 87.

Manejando un valor lógico

En C++ los valores booleanos son dos: cero que simboliza un valor falso **(false)** y cualquier otro, que representa uno verdadero **(true)**.

En el fichero cabecera **<iomanip>** están los manipuladores (formateadores) de E/S (*I/O manipulators,* en inglés), que pueden emplearse para alterar la forma en que se ve el flujo de bits.

- ✓ Los formateadores son secuencias de valores de bits que pueden ser creados con la función-miembro **setf**, y destruidos con la función-miembro **unset**.

Suministrados por la clase **ios** y simples (aunque EODA a veces fastidiosas) de usar, son imprescindibles para armar salidas visualmente placenteras. Por ejemplo, sea la siguiente instrucción:

```
cout << "La igualdad entre su nombre y direcci\242n es "
     << std::boolalpha << (nombre == direccion) << "\n";
```

En el ejemplo, **std::boolalpha** activa el manipulador que muestra los nombres de los valores booleanos. Ahora, si ambos contenidos son iguales **cout** pone en consola: **La igualdad entre su nombre y dirección es true** y si no lo son, entonces pone: **La igualdad entre su nombre y dirección es false** ¡La pega es que la salida del valor booleano es en inglés!

Otros manipuladores de salida son **hex**, que saca los números en formato hexadecimal, **oct**, que saca los números en formato octal, y **dec**, que se usa por omisión y saca los números en formato decimal. Más adelante en la página 98, se ampliará este tema.

Operadores relacionales y lógicos

Son utilizados para obtener resultados booleanos y frecuentemente se usan juntos.

- Operadores relacionales. Comparan dos valores y producen un resultado booleano.

  Tabla 11. Operadores relacionales

Operador	Operación	Nombre	Para los ejemplos, x = 10; y = 3	
>	x > y	Mayor que	10 > 3	verdadero
>=	x >= y	Mayor o igual que	10 >= 3	verdadero
<	x < y	Menor que	10 < 3	falso
<=	x <= y	Menor o igual que	10 <= 3	falso
==	x == y	Igual que	10 == 3	falso
!= not_eq	x != y x not_eq y	No es igual que	10 != 3 10 not_eq 3	verdadero

- Operadores lógicos. Conectan dos valores y el resultado es un booleano. El operador unario de negación, o no-lógico, niega (cambia) el valor actual.

  Tabla 12. Operadores lógicos

Operador	Operación		Nombre	Resultado
&&	x && y	x and y	Y lógico	**true** si ambos son verdaderos.
||	x || y	x or y	O lógico	**false** si ambos son falsos.
!	!x	not x	NO lógico	**false** si es verdadero y viceversa.

  Aunque el operador **not** en muchas ocasiones se presta de forma natural a ser usado en una expresión lógica sencilla, se advierte puede ser obviado si el programador lo desea, puesto que siempre habrá una expresión afirmativa equivalente y por regla general es mejor operar con valores que no estén negados. Además, si la expresión a probar es

complicada, el uso de **not** puede ofuscar el código. Recordar la filosofía del Zen: *Lo práctico gana a lo puro y frente a la ambigüedad, rechaza la tentación de adivinar.*

Programa 4: la cadena de C++

Se volverá a hacer el programa anterior, pero ahora con cadenas de C++, como comparación entre ambos tipos y se usará sólo lo necesario. Observar con cuidado las similitudes y diferencias.

Programa

```
#include <iostream>
#include <cstdlib>
#include <string>

int main() {
  // usando lo necesario
  using std::cout;    // para salidas
  using std::cin;     // para entradas
  using std::string;  // para hacer visible la clase

  // información al usuario
  cout << "Programa 4: un trabajo con cadenas C++.\n"
          "Se pedir\240n algunos datos personales.\n\n";

  // 1) entrada de una cadena C++
  string nombre;                  // define la cadena vacía
  cout << "Su nombre completo: "; // pide un valor
  std::getline( cin, nombre );    // entra el valor

  // 2) entrada numérica: como inmediatamente después de esta
  //    se va a hacer otra entrada de caracteres, hay que
  //    limpiar el búfer con get()
  int edad;             // define el entero
  cout << "Su edad: "; // pide un valor
  (cin >> edad).get(); // entra el valor y limpia el búfer

  // 3) entrada de otra cadena clásica
  string direccion;               // define la cadena vacía
  cout << "Su direcci\242n: ";    // pide un valor
  std::getline( cin, direccion ); // entra el valor

  // además de la declaración de una cadena vacía, la
  // iniciación de cadenas C++ puede hacerse de dos formas:
  string copiaNombre = nombre;        // por asignación directa
  string copiaDireccion( direccion ); // por constructor

  // la salida
  cout << "\nLa copia de su nombre es " << copiaNombre;
  cout << "\nLa de su direcci\242n es " << copiaDireccion;
  cout << "\nEl a\244o que viene Ud. tendr\240 " << ++edad << " a\244os.";

  // comparación entre cadenas
  cout << "\nSu nombre y direcci\242n";
  ( nombre == direccion )             // ¿son iguales?
    ? cout << " son iguales.\n\n"     //  sí lo son
    : cout << " no son iguales.\n\n"; //  no lo son

  // fin
```

```
  system ( "pause" );  // la salida se detiene
  return EXIT_SUCCESS; // el programa termina OK
}
```

Y con los mismos datos se obtiene un encabezamiento distinto, pero la misma salida.

PRIMER BLOQUE DE EJERCICIOS

En cada bloque de ejercicios se deben considerar todos los tópicos anteriormente estudiados y aplicar lo necesario en cada caso. Deben resolverse todos los problemas propuestos, tomando el tiempo que eso requiera, antes de pasar a otros tópicos. Si el lector es novel, es muy aconsejable que adopte sin reservas el estilo de codificación mostrado, pero si no es el caso, entonces deberá comprobar su estilo contra el dado en el texto. Si a su parecer, el que ya conoce es mejor, más legible, deberá atenerse al mismo a lo largo del curso. Si no, siempre puede tratar de realizar una readaptación.

Todas las soluciones deben ser guardados aparte y cada uno de sus códigos debe llevar amplios comentarios, incluso superfluos, pues son hechos por y para el lector. De tiempo en tiempo, deben ser reconsiderados y rehechos si se está insatisfecho con lo que se hizo, lo cual dará una medida cualitativa de su progreso.

Todos los problemas fueron resueltos, y el autor garantiza que cada bloque de tópicos, con adición incremental, da los elementos necesarios y suficientes para encontrar las soluciones. No hay problema presentado que no la tenga en el ámbito de este texto. Si en algún bloque o ejercicio, en aras de una mejor pedagogía o del cumplimiento de algún objetivo en particular se piden restricciones, el lector será advertido.

1. Codificar un programa que ponga en consola un rectángulo hecho de asteriscos, de tamaño 8 columnas de ancho por 5 filas de alto. Debería verse así:

```
Problema 1: un rectángulo de asteriscos.

        ********
        *      *
        *      *
        *      *
        ********
```

2. Codificar un programa que pida el radio (r) de un círculo y muestre su área (a.) La fórmula es: $a = \pi r^2$ Se puede tomar $\pi \approx 3.14$ La salida debería verse más o menos, así:

```
Problema 2: cálculo de un área.

Radio del círculo: 11
El área vale 379.94 unidades cuadradas.
```

3. Codificar un programa que pida entrar una cantidad entera de grados Fahrenheit (F) de temperatura y que devuelva tres valores convertidos a grados Celsius (C), el que se le dio, el anterior y el siguiente. La fórmula es: $C = (F - 32)\frac{5}{9}$ y la salida debería verse así:

```
Problema 3: conversión de temperatura.
212 grados Fahrenheit equivalen a 100 grados Celsius.

¿Grados Fahrenheit? 211
210 F equivalen a 98.8889 C.
211 F equivalen a 99.4444 C.
212 F equivalen a 100 C.
```

4. Codificar un programa que declare dos variables de cadenas clásicas. Tomar una frase y dividirla en dos: una parte va a la primera variable (por ejemplo, Una tarde) y la otra a la segunda (por ejemplo, fresquita de mayo.) Sacar a consola

su unión. Luego tomar dos valores enteros (por ejemplo, 13 y 9), poner uno en la primera variable y el otro en la segunda. Sumar y restar los valores de las cadenas y poner las ecuaciones en consola (por ejemplo, **`Suma: 13 + 9 = 22.`**) Sugerencia: la biblioteca **`<cstring>`** posee una función **`atoi`** (***A****SCII* ***to*** ***i****nt* en inglés), que convierte los caracteres ASCII de la cadena a enteros. La salida debería verse más o menos, así:

```
Programa 4: declara dos variables de cadenas clásicas.
Con ellas hace operaciones de cadenas y conversiones numéricas.

Una frase: una tarde
Otra frase: fresquita de mayo.
Unión: una tarde fresquita de mayo.

Un entero: 12
Otro entero: 9
Suma: 12 + 9 = 21
Resta: 12 - 9 = 3
```

5. Codificar un programa que declare dos variables de cadenas C++. Tomar una frase y dividirla en dos: una parte va a la primera variable (por ejemplo, Una tarde) y la otra a la segunda (por ejemplo, fresquita de mayo.) Sacar a consola su unión. Luego tomar dos valores enteros (por ejemplo, 13 y 9), poner uno en la primera variable y el otro en la segunda. Sumar y restar los valores de las cadenas y poner las ecuaciones en consola (por ejemplo, **`Suma: 13 + 9 = 22`**)

   Con los mismos datos, menos el encabezamiento, la salida debería verse igual. Sugerencia: la biblioteca **`<string>`** de C++ contiene un método que toma una cadena de C++ (**`cadena.c_str()`**), permitiendo usar aquella en una función propia de estas. Por ejemplo, llamando **`atoi(cadena.c_str())`** se regresa un entero sin problemas.

6 - VALORES LÍMITES

La cabecera **`<climits>`** *declara la plantilla de clases* **`numeric_limits`** *y los tipos y especializaciones relacionadas que definen los límites y características de los tipos aritméticos fundamentales. (Lischner, p. Cap. 13.32)*

Seguidamente se muestran los valores límites de C++. Cada realización está atada a la combinación PC/SO y la salida muy probablemente será distinta para cada equipo donde se ejecute. Se recomienda al lector averiguar las capacidades de su sistema.

Enteros y caracteres

El fichero C++ de encabezado **`<climits>`** define los parámetros que caracterizan la representación de los números enteros en su sistema. En la literatura usada se buscaron las macros y sus significados. Observar la salida, anotarla y tenerla en cuenta al programar operaciones matemáticas con resultados muy grandes o muy pequeños.

A. MACROS para caracteres

CHAR_BIT .....Bits por carácter (al menos 8)

B. MACROS para números enteros

INT_MIN .....Valor mínimo de un entero. No es menor que -32,767
INT_MAX .....Valor máximo de un entero. Al menos 32,767
LONG_MAX .....Valor máximo en un entero largo. Al menos 2,147,483,647
LONG_MIN .....Valor mínimo en un entero largo. No es menor que -2,147,483,647
SHRT_MAX .....Valor máximo en un entero corto. Al menos 32,767
SHRT_MIN .....Valor mínimo en un entero corto. No es menor que -32,768

USHRT_MAX.....Valor máximo de un entero corto. Al menos 65,535
UINT_MAX.....Valor máximo en un entero sin signo. Al menos 65,535
ULONG_MAX.....Valor máximo en un entero largo, sin signo. Al menos 4,294,967,295

Programa 5: enteros y caracteres

Programa

```
#include <iostream>
#include <cstdlib>
#include <climits>

int main() {
  //  caracteres
  std::cout <<
  "Programa 5: caracter\241sticas de este sistema (1)"
  "\nBits por byte = " << CHAR_BIT << "\n\n";

  //  enteros
  std::cout <<
  "L\241mites de los valores enteros:"<<
  "\nshort:"
  "\nValor m\241nimo = " << SHRT_MIN <<
  "\nValor m\240ximo = " << SHRT_MAX << '\n' <<
  "\nunsigned short:"
  "\nValor m\240ximo = " << USHRT_MAX  << '\n' <<
  "\nint:"
  "\nValor m\241nimo = " << INT_MIN  <<
  "\nValor m\240ximo = " << INT_MAX  << '\n' <<
  "\nunsigned int:"
  "\nValor m\240ximo = " << UINT_MAX << '\n' <<
  "\nlong int:"
  "\nValor m\241nimo = "  << LONG_MIN <<
  "\nValor m\240ximo = "  << LONG_MAX << '\n' <<
  "\nunsigned long int:"
  "\nValor m\240ximo = " << ULONG_MAX << "\n\n";

  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Salida

```
Programa 5: características de este sistema (1)
Bits por byte = 8

Límites de los valores enteros:
short:
Valor mínimo = -32768
Valor máximo = 32767

unsigned short:
Valor máximo = 65535

int:
Valor mínimo = -2147483648
Valor máximo = 2147483647

unsigned int:
Valor máximo = 4294967295

long int:
Valor mínimo = -2147483648
Valor máximo = 2147483647

unsigned long int:
Valor máximo = 4294967295
```

Nota: **`cout`** puede codificarse línea a línea, pero como permite el encadenamiento de varios flujos de salida, muchas veces es más limpio de codificar y más fácil de leer si se hace con una sola instrucción, como en este caso.

Números de punto flotante

El fichero C++ de encabezado **`<cfloat>`** define los parámetros que caracterizan la representación de los números de punto flotante (números reales matemáticos) en su sistema. Suministra tres prefijos comunes para los tipos fundamentales; para el punto flotante (**`float`**) es **`FLT_`**, para el punto flotante de doble precisión (**`double`**) es **`DBL_`** y para el punto flotante de doble precisión extendida (**`long double`**) es **`LDBL_`**. La aritmética común usa la base **`radix 10`** en sus cálculos, pero la aritmética de las computadoras utiliza las bases **`radix 16`**, **`8`** ó **`2`**, siendo esta última la que más se emplea hoy día. El rango de valores de un número de punto flotante depende ante todo de los valores límites de su exponente.

Observar la salida, anotarla y tenerla en cuenta al programar operaciones matemáticas con resultados muy grandes o muy pequeños.

A. MACROS para números de punto flotante (float)

- **FLT_DIG** ..... precisión decimal
- **FLT_EPSILON** ..... límite de certeza
- **FLT_MAX** ..... máximo valor positivo
- **FLT_MIN** ..... mínimo valor positivo
- **FLT_MAX_10_EXP** ..... exponente decimal máximo
- **FLT_MIN_10_EXP** ..... exponente decimal mínimo
- **FLT_MAX_EXP** ..... exponente máximo en la base radix implementada
- **FLT_MIN_EXP** ..... exponente mínimo en la base radix implementada
- **FLT_RADIX** ..... base radix implementada

radix es un número entero usado como base para hacer exponenciación. Usualmente es 2 (base binaria.)

B. MACROS para números de doble precisión (double)

- **DBL_DIG** ..... precisión decimal
- **DBL_EPSILON** ..... límite de certeza
- **DBL_MAX** ..... máximo valor positivo
- **DBL_MIN** ..... mínimo valor positivo
- **DBL_MAX_10_EXP** ..... exponente decimal máximo
- **DBL_MIN_10_EXP** ..... exponente decimal mínimo
- **DBL_MAX_EXP** ..... exponente máximo en base radix implementada
- **DBL_MIN_EXP** ..... exponente mínimo en base radix implementada

C. MACROS para números de doble precisión extendida (long double)

- **LDBL_MAX** ..... máximo valor positivo
- **LDBL_MIN** ..... mínimo valor positivo
- **LDBL_MAX_10_EXP** ..... exponente decimal máximo
- **LDBL_MIN_10_EXP** ..... exponente decimal mínimo
- **LDBL_MAX_EXP** ..... exponente máximo en base radix implementada
- **LDBL_MIN_EXP** ..... exponente mínimo en base radix implementada

Programa 6: números de punto flotante

```
#include <cstdlib>
#include <iostream>
#include <cfloat>   // límites de punto flotante

int main() {
    //  números en punto flotante
    std::cout
    << "Programa 6: caracter\241sticas de este sistema (2)\n"
    << "\nBase radix = " << FLT_RADIX << "\n\n"
    << "L\241mites de los valores en punto flotante:"
    << "\nfloat:"
    << "\nPrecisi\242n decimal = " << FLT_DIG
    << "\nL\241mite de certeza = " << FLT_EPSILON
    << "\nValor m\240ximo = "      << FLT_MAX
    << "\nValor m\241nimo = "      << FLT_MIN
    << "\nExponente decimal m\240ximo = " << FLT_MAX_10_EXP
    << "\nExponente decimal m\241nimo = " << FLT_MIN_10_EXP
    << "\nExponente radix m\240ximo = "   << FLT_MAX_EXP
    << "\nExponente radix m\241nimo = "   << FLT_MIN_EXP
    << "\n\ndouble:"
    << "\nPrecisi\242n decimal = " << DBL_DIG
    << "\nL\241mite de certeza = " << DBL_EPSILON
    << "\nValor m\240ximo = "      << DBL_MAX
    << "\nValor m\241nimo = "      << DBL_MIN
    << "\nExponente decimal m\240ximo = " << DBL_MAX_10_EXP
    << "\nExponente decimal m\241nimo = " << DBL_MIN_10_EXP
    << "\nExponente radix m\240ximo = "   << DBL_MAX_EXP
    << "\nExponente radix m\241nimo = "   << DBL_MIN_EXP
    << "\n\nlong double:"
    << "\nPrecisi\242n decimal = " << LDBL_DIG
    << "\nL\241mite de certeza = " << LDBL_EPSILON
    << "\nValor m\240ximo = "      << LDBL_MAX
```

```
    << "\nValor m\241nimo = "      << LDBL_MIN
    << "\nExponente decimal m\240ximo = " << LDBL_MAX_10_EXP
    << "\nExponente decimal m\241nimo = " << LDBL_MIN_10_EXP
    << "\nExponente radix m\240ximo = "   << LDBL_MAX_EXP
    << "\nExponente radix m\241nimo = "   << LDBL_MIN_EXP
    << "\n\n";

    system ( "pause" );
    return EXIT_SUCCESS;
}
```

Salida

```
Programa 6: características de este sistema (2)

Base radix = 2

Límites de los valores en punto flotante:
float:
Precisión decimal = 6
Límite de certeza = 1.19209e-07
Valor máximo = 3.40282e+38
Valor mínimo = 1.17549e-38
Exponente decimal máximo = 38
Exponente decimal mínimo = -37
Exponente radix máximo = 128
Exponente radix mínimo = -125

double:
Precisión decimal = 15
Límite de certeza = 2.22045e-16
Valor máximo = 1.79769e+308
Valor mínimo = 2.22507e-308
Exponente decimal máximo = 308
Exponente decimal mínimo = -307
Exponente radix máximo = 1024
Exponente radix mínimo = -1021

long double:
Precisión decimal = 18
Límite de certeza = 1.0842e-19
Valor máximo = 1.18973e+4932
Valor mínimo = 3.3621e-4932
Exponente decimal máximo = 4932
Exponente decimal mínimo = -4931
Exponente radix máximo = 16384
Exponente radix mínimo = -16381
```

7 – DECISIONES Y CICLOS

Decir que C++ es un lenguaje estructurado, es decir que posee estructuras de control del flujo del programa basadas en decisiones y también en ciclos. Ambos tipos de estructuras son muy útiles cuando se combinan. El Autor.

Conocido por su férrea oposición a la instrucción de programación **goto**, que culminó en 1968 con una carta a la revista Comunicaciones del ACM intitulada *Go To Statement Considered Harmful* (La instrucción **goto** considerada perjudicial, en

español), y fue visto como un paso decisivo de su reemplazo por estructuras de control, en 1968 Edsger W. Dijkstra demostró que con sólo tres instrucciones se podría hacer cualquier programa escrito en lenguaje imperativo: la secuencia (obligatoria), la decisión (comúnmente expresada con ***if-then-else***) y el ciclo (indicado con ***while-do***)

C++ tiene todas esas construcciones

Juego de instrucciones if

Es la instrucción fundamental del control de decisiones en casi todos los lenguajes imperativos contemporáneos y en C++ viene dada en tres variantes: **`if`**, **`if-else`**, e **`if-en-cascada`**, que permite al programa tomar un conjunto de decisiones en base a una prueba lógica. Se recomienda siempre escribir la instrucción **`if`** en cualquiera de sus variaciones, en la forma que se indica:

1. Si la condición lógica es verdadera se ejecuta el bloque de instrucciones y si no, la instrucción se ignora.

```
if (condición lógica) {
    bloque_de_instrucciones;
}; // if
```

2. Si la condición lógica es verdadera se ejecuta el bloque 1 de instrucciones y si no, se ejecuta el bloque 2.

```
if (condición lógica) {
    bloque_1_de_instrucciones;
} else {
    bloque_2_de_instrucciones;
}; // if-else
```

Cuando la condición lógica es simple y ambos bloques están constituidos cada uno por una sola instrucción ni larga, ni compleja —y ambas son afines— es más rápido, eficaz y eficiente usar el operador ternario.

3. Si se tienen múltiples condiciones lógicas, los **`if`** se van analizando en cascada, uno tras otro; por eso la llaman así. Cuando uno de ellos es verdadero se ejecuta su bloque y termina la instrucción. Si ninguno lo es, se ignora.

```
if (condición lógica 1) {
    bloque_1_de_instrucciones;
} else if (condición lógica 2) {
    bloque_2_de_instrucciones;
} else if (...) {
    ⋮             ⋮
} else if (condición lógica enésima) {
    bloque_n_de_instrucciones;
} // if-en-cascada
```

Una vez terminada la instrucción de una forma u otra, el programa continúa con la que sigue.

Refactorización

Es una técnica de ingeniería de software que se usa para describir la modificación del código fuente sin cambiar su comportamiento, lo que se conoce informalmente por *limpiar el código*. Es decir, que ni arregla bugs, ni añade funcionalidad. Antes bien, sus tres objetivos son (**1**) mejorar la facilidad de lectura y comprensión; (**2**) cambiar estructura y diseño favoreciendo uno más simple; y (**3**) eliminar código “muerto”, para facilitar el mantenimiento. Y todo esto, porque añadir nuevo comportamiento a un programa puede ser difícil con la estructura original, así que un desarrollador puede hacer una refactorización para facilitar la tarea y luego añadir el nuevo comportamiento.

Un ejemplo

Sea un juego llamado “Blanco y Negro” consistente en que un jugador hace una apuesta y seguidamente extrae de un artefacto conteniendo la misma cantidad de fichas negras y blancas, dos fichas. Según lo que saca, el resultado es:

Tabla 13. Juego del Blanco y Negro

Ficha 1	Ficha 2	Pago
Blanca	Blanca	0:1
Blanca	Negra	2:3
Negra	Blanca	1:1
Negra	Negra	2:1

O sea, en el primer caso se pierde todo, en el segundo se pierden las dos terceras partes de la apuesta, en el tercer caso ni se gana, ni se pierde y en el último caso, el jugador gana el doble de su apuesta. ¿Cómo codificar esto? Recordar lo dicho en la en la página 15. Aquí se ve un código con la instrucción **`if`**, pero ella se explica totalmente en la página 51.

```
if ( true == ficha1 and true == ficha2 ) {
    std::cout << "Perdió todo.\n\n";
} else if ( true == ficha1 and false == ficha2 ) {
    std::cout << "Perdió " << 0.6667 * apuesta << "\n\n";
} else if ( false == ficha1 and true == ficha2 ) {
    std::cout << "Ni ganó, ni perdió.\n\n";
} else {
    std::cout << "Ganó " << 2 * apuesta << "\n\n";
} // if-en-cascada
```

El código es compacto y algo ofuscado, pero refactorizado se comprende mucho mejor:

```
if ( true == ficha1 ) {
   if ( true == ficha2 ) {
      std::cout << "Perdió todo.\n\n";
   } else {
      std::cout << "Perdió " << 0.6667 * apuesta << "\n\n";
   } // if-else
} else {
   if ( true == ficha2 ) {
      std::cout << "Ni ganó, ni perdió.\n\n";
   } else {
      std::cout << "Ganó " << 2 * apuesta << "\n\n";
   } // if-else
} // if-else
```

Instrucción while

Es la instrucción fundamental del control de ciclos en C++ y en casi todos los lenguajes imperativos contemporáneos. Con ella se puede hacer cualquier tipo de ciclo, pero por su facilidad de codificación, su empleo más común es en iteraciones abiertas, o en las controladas por centinela.

Los ciclos pueden ser de cuatro tipos: (**1**) sinfín: nunca terminan por sí mismos; (**2**) con iteraciones cerradas: se conoce cuántas serán; (**3**) con iteraciones abiertas: no se conoce su cantidad; y (**4**) con cierre por centinela: se conoce un símbolo de terminación, comúnmente llamado…centinela.

La instrucción **`while`** evalúa una condición lógica al mismo inicio, y mientras esta sea verdadera el ciclo se ejecuta. Tan pronto como la condición lógica se torna falsa el ciclo termina. ¡Pero ojo! si la condición lógica es falsa desde el principio,

el ciclo nunca se ejecuta. Por supuesto, y como debe ser para cada tipo de ciclo, existirá una instrucción que condicione su final, si no, éste sigue ejecutándose hasta agotar los recursos a mano y el programa termina con un fallo catastrófico.

Se recomienda siempre escribir la instrucción de la siguiente forma:

```
while (la condición lógica sea verdadera) {
   bloque_de_instrucciones;
} // while
```

Una vez terminada la instrucción de una forma u otra, el programa continúa con la que sigue.

Existen muchas condiciones lógicas que son abreviadas, compactas y…obscuras. Son las favoritas de los programadores que llegan curtidos del ANSI C, pero modernamente —y con mucha energía— se recomienda escribir código claro y limpio, en la línea del Zen de Python: *explícito es mejor que implícito; la legibilidad cuenta; y frente a la ambigüedad rechaza la tentación de adivinar.*

Escribir siempre las condiciones lógicas lo más claras posible; no dar lugar a interpretaciones

Programa 7: código ASCII

Mostrar los códigos ASCII de caracteres imprimibles a tres columnas en consola, sabiendo que de cero a 31, se les reserva para instrucciones al equipo, control de sus periféricos o para el trabajo en redes; de 32 en adelante y hasta 127 son símbolos imprimibles del idioma inglés; de 128 a 256 son códigos ampliados que contienen caracteres empleados en otros idiomas, como el español o el francés, símbolos para el dibujo ASCII (*ASCII artwork* en inglés) y otros. De los caracteres imprimibles, el primero (#32) es el espacio en blanco y el último (#127) es para borrar, y ninguno saca algo visible a consola. Eso se tomará en cuenta en la salida.

La condición lógica del ciclo debe establecer que se haga mientras que el valor del contador **i** no sobrepase el límite de 127. Hay dos formas de hacerlo:

```
1.  while (i < 127) { ... } // mientras que i sea menor que 127
2.  while (127 > i) { ... } // mientras que 127 sea mayor que i
```

¿Cuál debiera preferirse? La primera forma es más intuitiva y se codifica con mayor naturalidad, pero el lector decidirá cuál le gusta más. ¿Y qué pasaría si la condición fuera de igualdad? También hay dos formas de hacerlo:

```
1.  while (i == 127) { ... } // mientras que i sea igual que 127
2.  while (127 == i) { ... } // mientras que 127 sea igual que i
```

¿Cuál debiera preferirse? Aquí ya no va sólo de predilecciones. Si bien la primera forma sigue siendo más clara, más natural, el programador novel enfrenta una molesta y bastante común posibilidad de error: escribir el operador de asignación (=) por el de igualdad (==) como se muestra a continuación. Con ambas formas el ciclo se ejecutará, pero con la asignación se introduce un error impredecible: cada vez que se ejecuta el ciclo, **i** toma el valor de 127, la condición es cierta y eventualmente, el programa agotará los recursos y la ejecución terminará con un fallo catastrófico.

```
while (i = 127) { ... } // valor a variable - ¡error de ejecución!
```

El compilador no será de ayuda, porque es un error en tiempo de ejecución y quizás su localización no salte de inmediato a la vista, pero si codifica así:

```
while (127 = i) { ... } // variable a valor literal - ¡error de compilación!
```

con la asignación se tiene un *literal constante* a la izquierda de la expresión (*lvalue: left value* en inglés) al cual no se le puede asignar el valor de la derecha de la expresión (*rvalue: right value* en inglés.) Ahora el compilador le ayudará de inmediato y el error no prosperará.

Al comparar por igualdad a variables y constantes, siempre poner la constante a la izquierda

Para el ciclo no importa que el valor usado de la variable contadora sea de pos o de preincremento, pero el preincremento es ligeramente más eficiente.

Programa

```
#include <iostream>
#include <cstdlib>

int main() {
  // información
  std::cout << "Programa 7: c\242digos ASCII imprimibles.\n";

  // encabezamiento de 3 columnas
  std::cout <<
    "\nN\243m.\tC\242digo\t"
    "\tN\243m.\tC\242digo\t"
    "\tN\243m.\tC\242digo\n";

  // variables de control de la salida
  const short COLS =  3; //  de 3 en 3 columnas
  short i          = 33; //  primer índice
  short col        =  1; //  contador de columnas

  // primer código: espacio en blanco; se imprime directamente
  std::cout << " 32\tBlanco\t\t";

  // ciclo: mientras no sea el último carácter, ponerlo
  // y avanzar índice y columna
  while ( i not_eq 127 ) {
    std::cout << " " << i << "\t   " << char ( i ) << "\t\t";
    ++i;
    ++col;

    // control del cambio de columnas: si es la 3ra. cambio de línea
    if ( 0 == (col % COLS) ) {
      std::cout << '\n';
    } // if
  } // while

  // último código: instrucción de borrado; se imprime directamente
  std::cout << " 127\t  DEL\n\n";

  //  fin
  system("pause");
  return EXIT_SUCCESS;
}
```

Salida

```
Programa 7: códigos ASCII imprimibles.

Núm.    Código          Núm.    Código          Núm.    Código
 32     Blanco           33        !             34        "
 35        #             36        $             37        %
 38        &             39        '             40        (
 41        )             42        *             43        +
 44        ,             45        -             46        .
 47        /             48        0             49        1
 50        2             51        3             52        4
 53        5             54        6             55        7
 56        8             57        9             58        :
 59        ;             60        <             61        =
 62        >             63        ?             64        @
 65        A             66        B             67        C
 68        D             69        E             70        F
 71        G             72        H             73        I
 74        J             75        K             76        L
 77        M             78        N             79        O
 80        P             81        Q             82        R
 83        S             84        T             85        U
 86        V             87        W             88        X
 89        Y             90        Z             91        [
 92        \             93        ]             94        ^
 95        _             96        `             97        a
 98        b             99        c             100       d
 101       e             102       f             103       g
 104       h             105       i             106       j
 107       k             108       l             109       m
 110       n             111       o             112       p
 113       q             114       r             115       s
 116       t             117       u             118       v
 119       w             120       x             121       y
 122       z             123       {             124       |
 125       }             126       ~             127      DEL
```

8 – CICLOS Y DECISIONES

En 1957 apareció en escena el lenguaje de programación FORTRAN, el primero de alto nivel ampliamente usado. Una de sus características es que emplea las seis letras ***i, j, k, l, m*** *y* ***n*** *como únicos nombres legales para los contadores de ciclos. Esa tradición prendió casi universalmente en el mundo de la programación. C++ no es ajeno a ello y aunque no tiene esa restricción, usarlas no debería traer confusión al código. Pero es solo eso: una tradición. Nombre sus contadores como desee, pero que sean lo más claro posibles y nunca dé lugar a interpretaciones. El Autor.*

Desde luego, C++ es un lenguaje muy completo. Antes que apegarse a la teoría, brinda más opciones al crear ciclos y decisiones, elementos imprescindibles en cualquier programa serio. Para completamiento del lenguaje y comodidad del programador, aporta dos tipos más de instrucciones cíclicas (**`for`** y **`do-while`**) y otro tipo más de instrucción de decisión (**`switch`**.)

Instrucción for

Presente en todos los lenguajes imperativos contemporáneos, el **`for`** de C++ es una generalización sumamente flexible del que suministran otros. Aunque su empleo más común es en iteraciones cerradas, **`for`** al igual que **`while`**, también sirve para cualquier tipo de ciclo. Usarlo siempre es cuestión de gustos, pero entonces hay que tener presente: (**1**) tratar de llenar las tres condiciones del **`for`**; y (**2**) que esas condiciones no sean complejas, aunque siempre recordando el Zen de Python: *lo práctico gana a lo puro*.

- Se recomienda siempre escribir la instrucción **`for`** de la siguiente forma:

```
for ([inicialización del contador]; [condición lógica de terminación]; [incremento del contador]) {
   bloque_de_instrucciones;
} // for
```

- La inicialización del contador puede estar constituida por una o más expresiones del mismo tipo de dato, separadas por comas —aunque se recomienda mantenerla lo más simple posible y hacer las afectaciones secundarias dentro del ciclo, en aras de preservar la claridad. Recordar el Zen de Python: *explícito es mejor que implícito; y simple es mejor que complejo*.
- La condición lógica de terminación evalúa a uno de los valores booleanos, ya sea directamente, ya por medio de un cálculo aritmético adecuado. Si la condición lógica es verdadera (no vale cero), se ejecuta el bloque de instrucciones y seguidamente se ejecuta el bloque de incremento del contador —que puede ser una o más expresiones separadas por coma— realizando otro ciclo. Cuando el incremento torna falsa la condición lógica, el ciclo termina normalmente y el flujo del programa continúa con la instrucción que le sigue.
  - ✓ Naturalmente, la condición lógica de terminación y su vinculación al incremento deben condicionar su final, si no, sigue ejecutándose hasta agotar los recursos a mano y el programa terminará con un fallo catastrófico.
- Cualquiera de los tres elementos que forman el cuerpo del **`for`** puede estar ausente, o como ya se dijo, estar constituido por una o más expresiones separadas por coma. Eso también lo hace muy flexible, pero también puede dar origen a código ofuscado muy fácilmente.
  - ✓ Un cuerpo de **`for`** vacío a veces es usado por programadores expertos para controlar un ciclo, pero en general debe ser evitado ya que la intención de esa instrucción es agrupar y mostrar todos los aspectos de su control en un lugar.

  Si hay necesidad de controlar el ciclo dentro de sí mismo, siempre considerar el uso de las instrucciones **`while`** o **`do-while`** como más adecuadas.

Programa 8: ciclos con for

Se requiere elevar un número cualquiera a una potencia entera. Se conoce que $a^n = \overbrace{a \times a \times a \ldots}^{\textbf{n veces}}$ En este caso a es la base real y n el exponente entero. Codificar un programa que realice el trabajo y lo siga haciendo mientras que el usuario lo desee. Como el ciclo de cálculo está cerrado por el valor del exponente, la instrucción **`for`** puede efectuarlo con mayor comodidad, pero el fin del programa lo decide el usuario, por lo que la instrucción **`while`** se presta mejor para esta tarea.

Programa

```
#include <iostream>
#include <cstdlib>
#include <cassert>
```

```
//  el nuevo nombre sólo es un apodo
typedef unsigned short USHORT;

int main() {
  //  la variables se declaran fuera del ciclo
  USHORT elExponente;
  double laBase;
  double laRespuesta;
  char sn;
  std::cout << "Programa 8: potencia un n\243mero.\n\n";

  while (true) {
    //  entrada: entra la base
    std::cout << "Escriba el n\243mero: "; // pide un valor
    std::cin >> laBase;                    // entra el valor
    //  entrada: entra el exponente
    std::cout << "Escriba el exponente entero: "; // pide un valor
    std::cin >> elExponente;                      // entra el valor
    //  se asegura de que el exponente sea mayor o igual a cero
    assert ( elExponente >= 0 );

    //  cálculo:
    laRespuesta = 1; // neutro del producto

    // hace el ciclo y cálcula la potencia
    for ( USHORT i = 0; i < elExponente; ++i ) {
      laRespuesta *= laBase;
    } // for

    //  salida:
    std::cout << "Respuesta = " << laRespuesta
        << "\n\n\250Una vez m\240s? (s/n) > ";
    std::cin >> sn; // ¿seguimos?

    if ( 'N' == toupper(sn) ) {
      return EXIT_SUCCESS;
    } // if
  } // while
}
```

Salida

```
Programa 8: potencia un número.

Escriba el número: 5.01
Escriba el exponente entero: 2
Respuesta = 25.1001

¿Una vez más? (s/n) > n
```

Otra optimización

Cuando se encuentra un código como **`resp = resp * base`**, puede escribirse como **`resp *= base`**, porque es más fácil de escribir y el código generado está optimizado con respecto a la rapidez. Esto puede y debe hacerse con los operadores aritméticos. Es una práctica fuertemente sugerida a los codificadores de C++, aunque los buenos compiladores contemporáneos hacen automáticamente la optimización mostrada cuando ven a estos códigos en su forma no optimizada.

Tabla 14. Optimización con los operadores aritméticos

Frente a esto	`a = a + b`	`a = a - b`	`a = a * b`	`a = a / b`	`a = a % b`
Codificar esto	`a += b`	`a -= b`	`a *= b`	`a /= b`	`a %= b`

Siempre tratar de usar esta optimización

Control adicional

A veces se requiere de un control extra, más allá del que cualquier instrucción cíclica otorga. C++ ofrece tres instrucciones adicionales de control: **`break`**, **`continue`** y **`goto`**, y todas emplean un cambio brusco de la dirección del flujo del programa, es decir, un salto.

goto

Muchos programadores adictos a la codificación estructurada…llevan su aversión por el goto al extremo. Si piensa que el goto nunca debería ser usado, le sugiero que considere cómo hacer un lenguaje de máquina sin instrucciones de bifurcación... (Jaeschke, CUJ, abril 1991)

El **`goto`** fuerza un salto inmediato a la parte del código donde está una etiqueta esperándola. Una etiqueta se declara por su nombre y luego se pone donde se desea, inmediatamente seguida de dos puntos.

En los albores de la programación, en parte debido a las fuertes limitaciones de los lenguajes ensambladores, y también a que los primeros lenguajes de alto nivel tendían a simular sus instrucciones cíclicas con las de aquellos (el BASIC clásico es un notable ejemplo), el uso del salto incondicional emulado por **`goto`** dio lugar a la programación ofuscada. Los programas codificados con lenguajes tempranos presentaban una lógica llena de saltos imprevisibles, enrevesada como los fideos en un plato, y así mismo se le llamó a dicho pésimo estilo: programación espagueti[15].

La realidad es que **`goto`** es una herramienta más a disposición del programador, y a pesar de haber sido estigmatizada a finales de los 60, no hay nada maligno en ella: es legítimo emplearla para la tarea adecuada y por eso el comité ISO/ANSI de estandarización para el C++ —WG21/X3J16— la dejó en el lenguaje. En particular la delimita y restringe su uso indiscriminado, prohibiendo que pueda saltar fuera del módulo donde se codifica, o cruzar ciertas declaraciones dentro del mismo módulo, y su compilador le ayudará a no sobrepasarse. Si la tiene que usar, no tema.

Su empleo ocasional pide se aplique casi exclusivamente a saltar desde un ciclo profundamente anidado al nivel más exterior del módulo, ahorrando espacio, tiempo y probabilidad de errar al tener que añadir lógica extra para manejar esa situación de manera estructurada. Y aquí es bueno recordar al Zen de Python: *bello es mejor que feo; y lo práctico gana a lo puro*. Al poner un **`goto`**, considerar usarlo sólo para este tipo de salto, y siempre escribirlo así:

```
goto Etiqueta;          <-- goto pide saltar inmediatamente para Etiqueta...
   ⋮                    <-- ...obvia el resto de las instrucciones...
Etiqueta:               <-- ...salta aquí...
próxima instrucción;    <-- ...y el programa sigue inmediatamente aquí
```

[15] ¿El lector quiere ver un seudocódigo espagueti? Le sugiero trate de leer el acápite “Procedure for Reading This Set of Books”, nada menos que del mismísimo Donald E. Knuth, en su primer tomo del «*The Art of Programming Computer, 3rd. Ed., volume 1: Fundamental Algorithms*» ¡Pura poesía del medioevo!

Se recomienda que comience el nombre de la etiqueta con mayúscula y la declare en una sola línea, justo antes del lugar donde se desea que se salte, tal como lo muestra el ejemplo. También se puede poner la instrucción inmediatamente después de la etiqueta, aunque pierde alguna claridad en la lectura.

break

Se usa para salir de un ciclo cualquiera, o para salir de una cláusula del **`switch`**, causando una salida inmediata, pero atención: si es de un ciclo anidado (un ciclo dentro de otro), sólo salta hacia el nivel superior que le quede más próximo.

break no puede usarse fuera de un ciclo o del **`switch`**

Por ejemplo, trabajando en el ciclo más interior de un conjunto anidado en múltiples niveles, si hay que salir directamente del más interior al más externo tendrá que codificar lógica extra para hacerlo estructuradamente, nivel a nivel. Desde luego, es más limpio, claro y eficiente salir con **`goto`**. Recordar el Zen de Python: *la legibilidad cuenta*; *y lo práctico gana a lo puro*.

continue

Cortocircuita una iteración dentro de cualquier ciclo, llevando a comenzar otro, a partir de donde ella aparece.

`break` y **`continue`** también deben ser usados juiciosamente. Se advierte que por la misma razón que **`goto`** es un peligro potencial, ellos también lo son, aunque en menor medida, pues sus saltos son mucho más controlados, contenidos y cercanos. De toda suerte, los programas "saltarines" no son fáciles de comprender y un empleo liberal de cualquiera de estas instrucciones (o de todas ellas) puede ofuscar de forma notable, incluso a una pequeña función.

Programación Defensiva

La programación defensiva es una metodología de diseño aplicada al software, que busca garantizar el comportamiento de ciertos elementos de una aplicación ante cualquier situación de uso, por incorrecta o imprevisible que ésta pueda parecer. (Wikipedia en español)

Al ser metodología, su uso no es obligatorio y se deja al criterio del programador su aplicación, pero se advierte que ignorarla no es nada bueno. Este tipo de programación en general se aplica a componentes críticos cuyo mal funcionamiento —ya sea por descuido o intencionalmente— podría acarrear consecuencias graves o daños catastróficos a la ejecución del producto. Es un enfoque que busca mejorar el código escrito haciéndolo claro, aumentar su fiabilidad reduciendo el número de fallos incontrolados, y lograr un comportamiento predecible frente a entradas inesperadas de datos.

assert

Entre las múltiples y disímiles técnicas propias de esta tarea, una muy simple y útil es la aseveración (*assert* en inglés), que permite verificar si cierta acción no concuerda con lo que el programador asegura que debería ser cierto en ese instante. Si su expresión evalúa a falso, la macro **`assert`**, definida en la cabecera **`<cassert>`**, aborta inmediatamente la ejecución del programa (lo cual a veces es una pega) y envía información a la salida de errores **`cerr`**. Otrosí no hace nada.

- ✓ Macro o macroinstrucción: significa la sustitución por el preprocesador antes de la compilación de la identificación definida. En código nuevo debe evitarse el empleo de macros, sustituyéndoles por funciones **`inline`**.
- ✓ **`cerr`**: es el canal estándar de salida de los mensajes de error, usualmente a consola.

En presencia de la ejecución de un **`assert`**, este compilador emite un mensaje similar al siguiente:

Assertion failed: <expresion>, file <fichero>, line <n>

Donde:

- **expresión** es la expresión del **assert**
- **fichero** es el fichero-fuente dónde ocurrió el fallo.
- **n** es el número de línea donde ocurrió el fallo.

assert es una técnica de cero tolerancias, porque al no cumplirse la afirmación, la ejecución aborta. Se emplea para verificar si un programa trabaja correctamente al procesar una parte crítica del código. Su expresión se construye de modo que evalúa a verdadero si no hubo error. Por ejemplo:

```
#include <cassert> // para usar assert
    ⋮              // más instrucciones aquí
assert(lts > 0);   // se afirma que los litros serán positivos
```

assert puede controlarse globalmente con la directiva de preprocesamiento **NDEBUG**. Si no está activa, entonces las aseveraciones se activan, y viceversa. Para desactivar la directiva basta poner en un fichero **#define NDEBUG** y sus **assert** no trabajarán; para desactivarla sólo se requiere comentar la línea y ahora los **assert** trabajarán.

En un programa con varios centenares de módulos en fase de desarrollo, se les define y comenta **NDEBUG** al inicio. Una vez terminado el proceso de depuración, todos los **assert** de cada módulo se desactivan descomentando a **NDEBUG**. ¡Si hace falta más depuración, de nuevo se comenta **NDEBUG** en ese lugar y ya está!

Una alternativa de assert

¿Qué tal si lo que se desea es una técnica defensiva tolerante, correctora, donde el usuario tenga la oportunidad de rehacer su acción si no fue la mejor? Entre las varias alternativas existentes hay una sencilla: **assert** se puede emular con un **if** evitando abortar el programa, o permitiendo sacar los mensajes de error directamente. Por ejemplo, algo así como:

```
if (condición de aseveración) { ... } [ else { ... } ];
```

donde la gestión de error delega el trabajo al programador, pero en cambio es más flexible. Sin embargo, se le advierte al lector que con estructuras **if** el control global y total de errores no se puede obtener ni tener, aunque hay otras formas que se verán en su momento.

Cambiando un nombre por otro más simple

Puede crearse un nuevo nombre que defina un tipo de dato existente. Su forma general es: **typedef <tipo> nombreNuevo;** Por ejemplo, el código siguiente instruye al compilador de que **uShort** es otro nombre para definir una variable del tipo **unsigned short** (entero corto, sin signo): **typedef unsigned short uShort;** Ahora la declaración mostrada es perfectamente legal y crea una variable llamada **exponente** que es del tipo entero corto, sin signo: **uShort exponente;**

¿Cuándo usar esta característica?

Hay dos escenarios posibles, pero en ambos casos, el usar un identificador más sencillo ahorrará tiempo de escritura del código y evitará errores en tiempo de compilación.

1. Se está empleando con frecuencia un tipo largo o confuso de escribir, como por ejemplo **unsigned short**, entonces es válido usar digamos:

   ```
   typedef unsigned short uShort;
   uShort q = 1;
   ```

2. Como elemento profiláctico frente a posibles cambios de tipo en versiones posteriores de la aplicación, para elevar la eficiencia, mejorar el algoritmo subyacente, etc. Un solo cambio en el **typedef** cubrirá el tener que cambiar decenas, centenas, o más veces, un tipo distribuido por multitud de ficheros.

Generación aleatoria de números

Esta técnica es muy útil en múltiples áreas científico técnicas, y entre otras situaciones se emplea para trabajar con aplicaciones que manejan muchos números cuyos valores son generados al azar en cada prueba, como es el caso del análisis estadístico. Para ello C++ suministra la función **rand()**, que está en la biblioteca **<cstdlib>**, y genera números enteros aleatorios cuyas características son:

- **RAND_MAX**: es una constante del sistema que da el valor máximo que **rand()** puede entregar. No será menor que 32,767
- La instrucción **i = rand()** genera un entero aleatorio entre **0** y **RAND_MAX** y lo pone en **i**.
- La instrucción **i = rand() % Top** genera un entero aleatorio entre **0** y **Top-1** y lo pone en **i**.
- La instrucción **i = a + rand() % Top** genera un entero aleatorio entre **a** y **Top** y lo pone en **i**. Si el valor de **Top** no es conveniente y se desea más grande, hay que afectar (multiplicar) la expresión por un factor adecuado.
- A veces se necesitan valores aleatorios **floats** o **doubles.** ¿Qué hacer? Una solución es afectar el entero por un número de punto flotante que puede ser **1.0**, mayor que **1.0**, o menor que **1.0**, o cualquier otro **double** o **float** de su agrado o conveniencia, o encasillar el valor, tópico a ver en la página 97.
- La instrucción **srand(time(0))** está en la biblioteca **<cstdlib>** y llamada así genera una semilla aleatoria basada en el número de segundos pasados desde que se arrancó la computadora, valor que lee la función **time(0)**, ubicada en la biblioteca **<ctime>**, por tanto es extremadamente baja la probabilidad de que se repitan los datos en varias corridas que no sean verdaderamente masivas.

  Si **srand** no se usa, se advierte que cada vez que el programa se ejecute en una sesión de trabajo, siempre generará la misma secuencia de números. Por eso a estos números se les llama seudoaleatorios.

Una técnica simple, pero efectiva, con fines de puesta a punto, es que durante el desarrollo de una aplicación se comente la función generadora de la semilla aleatoria hasta que se está satisfecho con el resultado obtenido, y luego se le elimina el comentario, para que el programa se comporte aleatoriamente.

Instrucción do-while

Hay ocasiones en que se requiere obligatoriamente que el ciclo se realice al menos una vez y esta es la instrucción idónea. La instrucción **do-while** evalúa la condición lógica en su final. Brinda comodidad al programador que desea codificar un ciclo que sabe se realizará al menos una vez. Como en **while**, mientras la condición sea verdadera el ciclo se ejecuta. Tan pronto como la condición lógica se torna falsa, el ciclo concluye normalmente, pero como ésta se evalúa al final, garantiza que el ciclo se ejecutará como mínimo una vez. Por supuesto, dentro del cuerpo del ciclo debe existir una instrucción que condicione su final, si no éste sigue ejecutándose hasta agotar los recursos a mano y el programa termina con un fallo catastrófico.

Una vez terminada la instrucción de una forma u otra, el programa continúa con la que sigue.

Estos tipos de ciclos se pueden hacer con **while**, pero el código queda más “artificial”

Un problema con esta instrucción es que puede llegar a ser obscura bajo determinada forma de escritura, es por eso que se recomienda siempre escribir a **do-while** de la siguiente forma:

```
do {
   bloque_de_instrucciones;
```

```
} while (condición_lógica_sea_verdadera); // do-while
```

En C++ las condiciones de terminación de **while** y **do-while** son las mismas

Programa 9: ciclos do-while

Realizar un ciclo creando números aleatorios, que termine cuando uno de ellos llegue a un valor dado como límite. Cambiando el valor de la constante **TOP**, se varia el límite; sumando un entero se desplazan los valores; la instrucción **srand(time(0))** asegura que cada ejecución sea diferente.

Programa

```
#include <iostream>
#include <cstdlib>
#include <ctime>   // para srand, time

int main() {
  //  la variables se declaran fuera del ciclo
  srand(time(0));           // semilla aleatoria
  unsigned i;               // control del ciclo
  unsigned cant = 0;        // ciclos hechos
  const unsigned TOP = 10; // factor límite

  // información
  std::cout << "Programa 9: el ciclo se ejecuta hasta que el "
     "n\243mero generado sea igual a " << TOP << "\n\n";

  // crear un número aleatorio, sacarlo a consola y contar
  // el ciclo, mientras que el número no sea igual al tope
  do {
    i = 1 + rand() % TOP;
    std::cout << i << ", "; ←
    ++cant;
  } while ( i not_eq TOP ); // do-while

  // salida: si es uno o son más, lo informa adecuadamente
  std::cout << "\b\b  \n\nDespu\202s de "; ←
  ( 1 == cant )
    ? std::cout << "una iteraci\242n "
    : std::cout << cant << " iteraciones ";
  std::cout << "sali\242 " << i << "\n\n";

  // fin
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Salida

```
Programa 9: el ciclo se ejecuta hasta que el número generado sea igual a 10

8, 1, 6, 3, 4, 8, 10

Después de 7 iteraciones salió 10
```

¿Se notó la coma de separación y cómo falta la última? El código es fácil (note las flechas):

1. poner la coma en el ciclo con `std::cout << i << ", ";`
2. y al final, fuera del ciclo, quitarla con `std::cout << "\b\b  \n\n. . .";`

Cómo hacer los ciclos con cadenas clásicas

Las operaciones típicas con cadenas de caracteres (asignación, copia, añadidura, comparación, etc.), pueden hacerse en base individual. Están declaradas en el fichero de encabezamiento **<cstring>** y siempre se harán con sus respectivas funciones. Las operaciones sobre caracteres están declaradas en el fichero **<cctype>** y siempre se harán con sus respectivas funciones. De otra suerte se incurre con facilidad en advertencias y/o errores de compilación.

Una forma de hacer operaciones cíclicas con cadenas clásicas es mediante terminación por centinela, que siempre es el carácter de cierre **\0**. Así el código se simplifica. Por ejemplo:

- `for (i = 0; cadena[i]; ++i) {...};`
- `while (cadena[i]) {...};`
- `do {...} while (cadena[i]);`

pero la lectura puede ofuscarse. Para el programador novel es preferible usar:

- `for (i = 0; cadena[i] == true; ++i) {...};`
- `while (cadena[i] == true) {...};`
- `do {...} while (cadena[i] == true);`

o mejor aún:

- `for (i = 0; cadena[i] > 0; ++i) {...};`
- `while (cadena[i] > 0) {...};`
- `do {...} while (cadena[i] > 0);`

Instrucción switch

Seguidamente se verá la instrucción **switch**, que permite múltiples decisiones las cuales se van analizando una a una, de una forma más clara y elegante que con el **if-en-cascada**. Más larga de escribir y con sus diferencias y limitantes, puede sustituirlo la mayoría de las veces, aunque no todas. Sus cuatro características básicas son:

1. Dentro de la instrucción se ponen etiquetas numéricas llamadas **case**, que pueden ser un carácter, un valor enumerativo, o el producto de cálculos que dan valores enteros, y que van marcando el punto de entrada al análisis. Si una de las etiquetas coincide con el valor, la ejecución continúa a partir de allí. Si el valor no casa con ninguna etiqueta, la ejecución continúa con la etiqueta por omisión **default**, si existe, o sale del **switch**.
2. La etiqueta **case** no maneja el flujo de control, sólo apunta a su entrada. Una vez pareado un valor con una etiqueta, el flujo de control comienza ahí, pero si no se detiene con **break**, pasa al siguiente caso y así sucesivamente, hasta agotar el **switch**. El último **case** puede obviar el **break**.
3. Un **switch** puede tener cuantas etiquetas **case** sean necesarias para su trabajo, en el orden que se quiera, pero ninguna puede estar repetida, o ser usada fuera del **switch**. El ordenamiento de las etiquetas no tiene efecto alguno en el rendimiento de la instrucción, aunque aporte claridad de lectura. Se recomienda poner el bloque por omisión al final. Entonces, si se desea puede obviarse el **break**, aunque por simetría se recomienda que esté.
4. Los caracteres mapean a números enteros, pero hay una sutil diferencia: para un **case** el número uno vale 1, pero el carácter '1' vale 49, que es su código ASCII decimal ¡Cuidado!

Se exhorta siempre escribir la instrucción **switch** de la siguiente forma, no necesariamente en orden, donde la parte entre corchetes puede estar presente o no:

```
switch (opcion) {
case opcion 1:
   bloque_1_de_instrucciones;
   break;
case opcion 2:
   bloque_2_de_instrucciones;
   break;
        ⋮      ⋮
case opcion enésima:
   bloque_n_de_instrucciones;
[  break; // si no está default, el switch termina aquí
default:
   bloque_de_instrucciones_por_omisión;
   [break;]    ] // usado de últimas se puede omitir el break
} // switch
```

Una vez terminada la instrucción de una forma u otra, el programa continúa con la que sigue.

¿Cuándo usar el switch en lugar del if?

Como consecuencia de las características básicas del **`switch`**, los compiladores de antaño le producían código más eficiente que para un **`if-en-cascada`** y muchos textos y programadores le prefieren si se puede usar la lógica múltiple de decisiones expresada mediante esta instrucción, Pero no es una imposición; con los compiladores modernos prácticamente es un problema de gustos personales. Como el código con el **`switch`** se lee más claro que con el **`if`**, se valorará lo siguiente:

- Con el **`switch`** no se pueden manejar intervalos de condiciones lógicas, por lo que llegado el caso no queda de otra: hay que usar el **`if-en-cascada`**.
- Si las condicionales se pueden manejar con ambas instrucciones, para el caso de una, dos, o tres, el **`if`** en cualquiera de sus variantes es más corto de escribir y se lee e interpreta bien. Si son más, usualmente el **`switch`** es mejor opción.

Programa 10: decisiones con switch

Crear un modelo adecuado de menú. Tener el modelo de un menú simple ayuda a acelerar la codificación mediante la reutilización y la adaptación, y eso se hizo en el texto, ya que muchos ejemplos son más educativos e instructivos si se hacen bajo esta línea de acción.

- ✓ Reutilización: propiedad que consiste en tomar un código desarrollado y depurado, y aplicarlo muchas veces a disímiles situaciones. Esto aporta integridad, fiabilidad, robustez y tipicidad, y contribuye a ganar rapidez en el desarrollo de las aplicaciones.
- ✓ Adaptación: código que no es exactamente lo buscado, pero se le acerca y para usarlo se modifica. Dícese entonces que fue adaptado.

Hay diversas y disímiles formas de crear un menú. En este caso en particular (y durante la serie), se usó la potencia y facilidad de manejo de las cadenas C++. Normalmente el diseño del programa decide si el menú llama por números[16], por nombres de comandos, o por iniciales de opciones. Aquí se da la primera forma, que se usa en el texto, tipificando el uso de un menú vertical, pero en el ejercicio 11 del bloque que viene, se pide desarrollar una forma horizontal que también será usada como alternativa en presencia de pocas opciones.

[16] Un truco para proteger la toma de opciones numéricas es poner el cero siempre para terminar, así la tecla está más alejada de las otras (1; 2; 3; etc.) y no es tan fácil pulsarla por equivocación.

Programa

```
#include <iostream>
#include <cstdlib>
#include <string>

int main() {
  // información
  std::string info("Programa 10: un prototipo de men\243.\n\n");

  // las variables fuera del ciclo
  std::string conjunto("01234"); // opciones legales
  std::string opcion;            // opción tomada
  std::string::size_type idx;    // índice de la opción

  while ( true ) {
    //  el menú:
    system("cls"); // borra la pantalla
    std::cout << info <<
    "Men\243\n"
    "1) Mirar\n"
    "2) Caminar\n"
    "3) Saltar\n"
    "4) Estornudar\n"
    "0) Terminar\n"
    "Entre una opci\242n: ";
    ( std::cin >> opcion ).get();
    idx = conjunto.find(opcion);

    if ( std::string::npos == idx ) {
      std::cerr << "\nError: opci\242n ilegal.\n\n";
      system("pause");
      continue;
    } // if

    switch( idx ) {
    case 0:
      return EXIT_SUCCESS;
    case 1:
      std::cout << "\nMirando a la izquierda.\n\n";
      break;
    case 2:
      std::cout << "\nCaminar mejora su salud.\n\n";
      break;
    case 3:
      std::cout << "\nSaltando hacia el futuro.\n\n";
      break;
    case 4:
      std::cout << "\n\255Salud!\n\n";
      break;
    default:
      std::cerr << "\nError: opci\242n ilegal.\n\n";
    } // switch

    system("pause");
  } // while
}
```

Salida

```
Programa 10: un prototipo de menú.

Menú
1) Mirar
2) Caminar
3) Saltar
4) Estornudar
0) Terminar
Entre una opción: 4

¡Salud!

Presione una tecla para continuar . . .
```

SEGUNDO BLOQUE DE EJERCICIOS

Considerar todos los tópicos anteriormente estudiados y aplicar lo necesario en cada ejercicio. No se está obligado a usar ninguna instrucción en especial, pero deben aprovecharse al máximo los nuevos conocimientos; sólo hay que emplear buen juicio.

6. La Cooperativa de Producción Agropecuaria CPA Santos García paga al personal que labora en el campo un salario básico de 500 CUP por 40 h de trabajo semanal, más tiempo y medio por cada hora extra. Codificar un programa que tome las horas totales trabajadas en el mes y emita el amonto ganado, sabiendo que la empresa toma cada mes como de 4 semanas. Para terminar, deberá entrarse -1. La salida debiera ser algo así:

```
Problema 6: cálculo del salario mensual.

-1 para terminar. Horas trabajadas: 158
Salario $ 1975

-1 para terminar. Horas trabajadas: 167
Salario $ 2131.25

-1 para terminar. Horas trabajadas: 170
Salario $ 2187.5

-1 para terminar. Horas trabajadas: -1
```

7. Codificar una tabla de potencias del uno al diez para el cuadrado, cubo, e inverso. Tratar de conseguir algo así:

```
Problema 7: tabla de números.

N       Cuad.   Cubo    Inverso
------------------------------
1       1       1       1
2       4       8       0.5
3       9       27      0.333333
4       16      64      0.25
5       25      125     0.2
6       36      216     0.166667
7       49      343     0.142857
8       64      512     0.125
9       81      729     0.111111
10      100     1000    0.1
------------------------------
```

8. Un número primo es un entero positivo que no siendo 1, sólo se divide por 1 y por sí mismo. Actualmente el cero no se contempla, 1 se define como no-primo y 2 es el primer y único primo par, porque cumple la condición estipulada. Preguntar por un número cualquiera y devolver a consola si es primo o no. El ciclo terminará entrando -1. Sugerencia**:** discriminar los casos especiales, antes de intentar procesarlos uno a uno. La salida debiera ser algo así:

```
Problema 8: determina si un número es o no primo.

-1 para terminar. ¿Número? 59
59 es primo.

-1 para terminar. ¿Número? 60
60 no es primo.

-1 para terminar. ¿Número? -1
```

9. Una pequeña variación del objetivo puede dar lugar a una remodelación del código que puede ser (o no) bastante seria ¿Qué tal si ahora se pide mostrar todos los divisores de un número que no sea primo? ¿Fue muy grande la remodelación? Sugerencia**:** apoyarse en el problema anterior. La salida debiera ser algo así:

```
Problema 9: determina si un número es o no primo
y si no lo es, muestra todos sus divisores.

-1 para terminar. ¿Número? 59
59 es primo.

-1 para terminar. ¿Número? 60
60 no es primo, divide por: 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, 60

-1 para terminar. ¿Número? -1
```

10. En el Reino del Revés la unidad monetaria es el bitkoin (**bK**) y su ONAT presenta el siguiente tipo de impuesto anual:
    - Los primeros 5,000 **bK** no pagan impuestos.
    - Los siguientes 10,000 **bK** pagan el 10%.
    - Los siguientes 20,000 **bK** pagan el 15%.
    - Entradas mayores de 35,000 **bK** pagan el 20% de impuestos.

    Por ejemplo, si alguien declara 38,000 bitkoins de ganancia, deberá pagar por concepto de impuestos $5000 \times 0 + 10000 \times 0.1 + 20000 \times 0.15 + 3000 \times 0.2 = 4600$ **bK**.

    Codificar un programa que pida una ganancia y devuelva la cantidad a pagar por concepto de impuestos. El programa debe correr hasta que se entre como ganancia el valor -1. La salida debiera ser algo así:

```
Problema 10: cálculo de impuestos del Reino del Revés

Impuestos Reales en bK (bitkoin)
Los primeros 5,000 no pagan
Los siguientes 10,000 pagan el 10%
Los siguientes 20,000 pagan el 15%
Más allá de 35,000 se paga el 20%

-1 para terminar. Ganancia anual: 9513
Paga 451.30 bK.

-1 para terminar. Ganancia anual: 19873
Paga 1730.95 bK.

-1 para terminar. Ganancia anual: 38951
Paga 4790.20 bK.

-1 para terminar. Ganancia anual: -1
```

11. El modelo de menú adoptado en el texto es muy flexible, pero algo voluminoso. Cuando solamente son pocas acciones (unas tres o cuatro) un menú horizontal por letras, comandos o ambas opciones es más estético. Adaptar el programa dado en la página 65. La salida sería algo así:

```
Problema 11: otro prototipo de menú.

M)irar, C)aminar, S)altar, E)stornudar
T)erminar. Entre una opción: e

¡Salud!

Presione una tecla para continuar . . .
```

9 – FUNCIONES

Decir que C++ es un lenguaje modular, es decir que se basa en módulos para realizar su trabajo. Al presentar la codificación súper C, se toma la filosofía del lenguaje ANSI C, que es dividir una gran tarea (un problema) en tareas más pequeñas (módulos) y repetir el proceso hasta conseguir funciones sencillas, quienes apoyándose entre todas dan la solución, metodología llamada Top-Down. El Autor.

Los programas serios, grandes, hechos en ANSI C, están compuestos por muchas (a veces muchísimas) funciones. Esos módulos pueden ser recogidos en agrupaciones afines, llamadas bibliotecas, como dicen en (Lenguaje C. Biblioteca de funciones, pág. viii) y pueden volver a ser usados una y otra vez.

Las bibliotecas oficiales para ISO C++ están definidas por la norma C++11

Reinventar la rueda

Ud. Puede pasarse días tratando de resolver una sección de su código para obtener cierto resultado y eso es frustrante y usualmente un gran consumidor de tiempo. Aun cuando es muy gratificante resolver finalmente el problema, más que a menudo ese código está allí afuera, en algún lugar, sea en internet, un texto, etc. No trate de reinventar la rueda, primero vea si alguien ya lo hizo y úselo o adáptelo. (Linux Tricks and Tips, pág. 65)

- ✓ Reinventar la rueda es una expresión que se utiliza para describir situaciones en las que el esfuerzo para solucionar un problema aparentemente nuevo es redundante o carente de sentido, puesto que la solución existe, pero se desconoce o se niega. (Wikipedia en español)

Tanto en los programas de ejemplo, como en los ejercicios propuestos, se creará código utilizable en la solución de más de un problema y no será necesario reinventar la rueda. Simplemente se adiciona al sistema un directorio más. Sin embargo, en este tomo no se hará hincapié sobre la reutilización, aun cuando se emplee junto con la adaptación. Se aplicará más a fondo en el tomo siguiente[17].

¿Por qué esta línea oficial? Porque el autor supone que el lector está aprendiendo y mientras más código escriba, más destreza tendrá y menos errores verán la luz en sus futuros trabajos.

Procedimientos y funciones

En informática, una subrutina o subprograma (también llamada procedimiento, función o rutina), como idea general, se presenta como una porción de código autocontenido que forma parte del módulo principal del programa e implementa un algoritmo que soluciona una o más tareas. Editado de (Wikipedia en español)

En C++ todos los módulos son funciones, y todos están al mismo nivel: no hay anidación de funciones, como por ejemplo en ANSI Pascal, donde una función puede ser totalmente declarada dentro de otra. Eso en C++ está prohibido.

Las subrutinas pueden ser invocadas en cualquier momento desde dentro de otro módulo o por la interfaz de otro módulo y en algunos lenguajes (como es el C++) también pueden ser llamadas recursivamente, técnica a ver en la página 81. Cuando una subrutina entra a ejecutarse, el programa detiene el código que se está efectuando y le cede el control. Una vez concluido el trabajo, dicho control es devuelto al llamador. Se entiende que el módulo puede llevar o no parámetros y devolver o no un resultado por interfaz o por retorno.

Si no devuelve resultados se llama procedimiento; si devuelve un valor se llama función. En C++ el procedimiento es emulado por una función tipo **`void`**. Entonces, ¿cuándo uno y cuándo el otro?

Tipos de funciones

En C++ pueden clasificarse cuatro tipos de funciones:

1. **Funciones puras**. Generan y retornan un resultado a partir de los datos que le suministran.
2. **Funciones que generan más de un resultado (procedimientos.)** Casi siempre son funciones tipo **`void`** que a partir de los datos que le suministran o ellas mismas piden, devuelven los resultados por su interfaz mediante pases por punteros, por referencias o por otros mecanismos avanzados que se verán en el segundo tomo. Aun cuando no son puras, son consideradas funciones verdaderas. A veces las tornan puras haciéndolas devolver un resultado que indica si la operación fue exitosa o no.
3. **Funciones auxiliares**, usualmente requeridas para obtener cohesión: si un módulo es definido como multitarea, es preferible desgajar sus objetivos en varios procedimientos (uno por tarea) y hacer que les llame. Según la tarea específica pueden ser funciones o procedimientos, y realmente, pueden llegar a ser muchas y muy variadas.
4. **Vistas**. Son procedimientos auxiliares especializados para presentar solamente información al usuario en un dispositivo de salida cualquiera (consola, impresora, etc.) Usualmente sus códigos son complicados y/o largos, manejan los

[17] Pero si el lector quiere, puede hacer un repositorio (directorio donde poner el código que será vuelto a usar) y ponerlo donde desee. Luego, con acciones de corta-y-empasta adaptará lo que necesite.

formatos adecuados, toman los datos necesarios a través de su interfaz, son muy modificables y están fuertemente acopladas a la información que manejan.

Función bien formada

Siempre que pueda, obtenga cohesión. Siempre trate de darle a cada pieza de código —módulo, función o clase— una responsabilidad única y bien definida. (Sutter, Exceptional C++, p. Item 10) Una función se dice que está bien formada cuando responde a la gran mayoría de las características siguientes, dadas aquí en orden de importancia:

Tabla 15. Características de una función bien formada

Característica	Objetivo
Cohesionada	Centrada para realizar una y sólo una tarea bien definida.
Pequeña	Pocas líneas, entre menos, mejor[18].
Autosuficiente	Controla como propio todo lo que necesita para hacer la tarea y nada más.
Ajustada	Interfaz lo más chica posible, que proporciona todo lo necesario para usarla y nada más.
Tersa	No tiene código superfluo, es decir, ciclos sinfín, saltos incontrolados, variables declaradas y no empleadas, o decisiones inalcanzables.
Coherente	Presenta solamente un lugar donde está la salida, si la tiene.

El objetivo del programador novel será codificar funciones adaptadas a la tarea y lo mejor formadas posible (mientras más características se cumplan, mejor formadas estarán), dejando las acciones de apoyo para las funciones auxiliares y las **`void`** mayormente para operaciones de salida, aunque siempre teniendo presente el Zen de Python: *lo práctico gana a lo puro*.

Tratar siempre de codificar funciones bien formadas

Modificadores de tipo

Los modificadores de tipo se usan para cambiar la forma en que C++ dispone el almacenamiento para las variables. Preceden al tipo de variable que modifican.

Tabla 16. Modificadores de tipo

Modificador	Uso
`auto`	Para declarar variables locales y es puramente opcional —nunca se usa. Estas variables se crean cuando son invocadas dentro del bloque al cual pertenecen y cesan de existir cuando el flujo del programa lo abandona.
`register`	Pide que una variable sea optimizada para la velocidad, y ya casi no se usa, porque los compiladores se volvieron mejores que el ser humano en eso de optimizar código. No obstante, si el compilador tiene reserva para registrarlas, lo hará por sí solo. Cuando agote ese límite, ignorará peticiones subsiguientes.
`static`	Instruye al compilador tener una variable local en existencia por la duración del programa en vez de crearla y destruirla cada vez que entra y sale de ámbito, por lo que esas variables mantienen sus valores entre las llamadas al bloque donde fueron declaradas.
`extern`	Se usa para comunicarle al compilador sobre variables declaradas fuera del alcance del programa. Las variables modificadas con **`extern`** no serán ubicadas pues estarán propiamente definidas en algún otro lugar.

Las funciones son **`extern`** por defecto, lo cual las hace accesibles desde cualquier punto de un programa, aunque se encuentren escritas en otros ficheros, permitiendo así la compilación separada (a ver en la página 86.) Pero **`extern`** tiene aún

[18] ¿Cuántas son pocas líneas? Evidentemente es un concepto muy personal. Digamos que a ojo de buen cubano y EODA, no serían mucho más allá de dos páginas completas de 80 líneas cada una.

otro uso: posibilita que ANSI C y C++ coexistan pacíficamente. Debido a diferencias a nivel de implementación y a la redecoración de nombres, si va a emplear en sus programas funciones originalmente codificadas por un compilador C, o por un compilador C++ de otro fabricante, debe de informárselo al suyo, declarándolas **extern**.

- ✓ Redecoración de nombres (*mangling* en inglés) es la forma en que cada fabricante de compiladores C/C++ codifica cada función (o método) de modo que pueda distinguir entre versiones sobrecargadas.

Por ejemplo, es posible que se esté trabajando con una biblioteca especializada cuya cabecera es, digamos **mylib.h**, desarrollada en ANSI C o en otro sistema, y no se tiene el fichero del código, o los deseos de traducirla a C++, o el tiempo requerido para hacerlo. Entonces **extern** viene al rescate: la especificación de enlace **extern "C"** le dice a C++ que las definiciones de **mylib.h** fueron compiladas en C por terceras personas.

```
extern "C" {
#include "mylib.h"
}
```

Función principal

Todo programa C++ obligatoriamente tendrá una y solamente una función principal llamada **main()**, desde donde partirá la ejecución de la aplicación y que retornará un número entero al SO. Esto no significa que habrá un fichero llamado **main.cpp**, aunque bien puede haberlo. De hecho Code::Blocks lo hace. Lo que realmente expresa es que para todo programa será obligatoria la existencia de uno y sólo un fichero llámese como se llame, pero que contendrá la función **main**. Se entiende perfectamente que junto a **main()** pueden haber más funciones en ese fichero que la contiene.

Desde luego, también se entiende perfectamente que en C++ cualquier función puede llamar a otra, incluyéndose a sí misma, característica llamada recursión (a ver en este mismo tema), pero ¿**main** llamando a **main**?... Sí, se puede, pero ¿cuál es el objetivo?

En un programa típico es C++ quien siempre debe llamar a **main**

(Lischner, p. Cap. 5.5) alerta que cualquier implementación ISO C++ soportará al menos dos formas de **main**: una vacía y otra que toma dos parámetros como argumentos en la línea de comandos del SO sobre el cual se ejecuta. La norma ISO/ANSI C++11 recomienda que esos dos primeros parámetros siempre sean **argc** y ***argv[]**, en ese orden, y ambas deberían retornar un valor **0** si hicieron su trabajo u un valor **-1** si fallaron, pero si Ud. no lo hace, C++ siempre retorna un entero al SO en concordancia con lo antes dicho.

```
1.  int main()                        // forma vacía
2.  int main(int argc, char *argv[])  // forma parametrizada
```

Implementaciones de terceras personas pueden soportar otras formas, adicionando más parámetros a los ya descritos, y la norma es algo ambigua en este punto, pero ¡cuidado! eso compromete la portabilidad de alguna suerte.

Los compiladores modernos dan error si no se sigue la norma, pero como **main** no posee prototipo, algunos compiladores antiguos también la aceptan como **void main()**, o sólo como **main()**, sin tipo. Esas formas no concuerdan con ser funciones principales propiamente dichas y son declaradas obsoletas, apartadas del estándar, desaprobadas.

Si se va a escribir código nuevo, la forma vacía es la más corta y fácil de usar

---

En la forma vacía, no hay una lista de posibles argumentos a pasar al SO, impidiendo alguna interacción con el programa. Si dicha comunicación no se requiere, es la forma más fácil de usar. Si se desea abreviar puede ponerse como valor de retorno **return 0** para un programa felizmente terminado, o **return -1**, para uno que falló, evitar así incluir a la biblioteca estándar. Si no se explicita la instrucción de retorno, por omisión se devuelve el valor cero si se ejecutó correctamente, y

otro valor no especificado si falló, pero no se considera buena práctica dejar el retorno de **`main`** implícito. La forma parametrizada se verá con más detalle en la página 204.

Prototipo de una función

En pocas palabras, una declaración es un término para todo aquello que le informa al compilador acerca de un identificador. El compilador C++ debe saber que significa realmente ese nombre para poder usarlo: ¿es un tipo de dato, de variable, función o clase? ¿es otra cosa? Por tanto, un fichero-fuente debe contener definiciones para cada nombre que usa, o conocer las definiciones de los nombres que se le han incluido.

El prototipo describe la interfaz entre la función y el programa principal, o sea, los valores que éste envía para que aquella ejecute su tarea, el tipo de valor que ella le devolverá y la sintaxis de ejecución. Operativamente hablando, el prototipo es el encabezado de la función, sin llaves y terminado con punto-y-coma, puesto antes de la función **`main()`**.

El compilador C++ conoce el nombre de la función mediante la información dada en el prototipo; si el tipo de su retorno concuerda con el declarado, si están correctamente declarados los parámetros que necesita para hacer su tarea y si no hay violación sintáctica. Así puede comprobar la justeza del llamado cada vez que el programa la convoca.

En C++ es obligatorio usar prototipos de funciones

Escriba así sus prototipos:

```
tipo nombre_de_función(lista_de_parámetros); // cierre con punto-y-coma
```

La única forma de evitar codificar un prototipo es poner la función antes que la llamada a **`main`**, lo cual no siempre es posible o deseable. Además, la vía de C++ es poner ante todo a **`main`**, porque usualmente muestra la estructura general del programa al nivel más externo.

Poner los tipos de datos de los parámetros usados en el prototipo es obligatorio; ponerles nombres a esos tipos es opcional. Por ejemplo, a los efectos del compilador ambas declaraciones representan el mismo prototipo —se dice que poseen la misma firma.

```
float potenciar(float, int);                // obligatorio poner los tipos
float potenciar(float base, int exponente); // opcional identificarlos
```

Aunque es totalmente opcional, se recomienda con fuerza identificar los parámetros para que sean fuente de documentación que ayude a comprender la lógica del código. Incluso, los nombres en el prototipo no tienen por qué coincidir con los que la función utiliza realmente, pero para evitar dudas, lo mejor es replicarlos. Recordar aquí el Zen de Python: *explícito es mejor que implícito; la legibilidad cuenta;* y *frente a la ambigüedad rechaza la tentación de adivinar*.

El codificador novicio:

- Para fines de claridad y documentación deberá poner siempre nombre a los parámetros del prototipo y si es posible, procurar que sean iguales a los de la función, lo cual aminora la ofuscación inevitable de tener que leer el mismo significado en dos lugares distintos.
- Si está trabajando solamente con un fichero-fuente, siempre que pueda, aplicará la regla práctica consistente en poner todos los prototipos antes de la función **`main`**, y sus definiciones al final de dicha función.

El sistema C++ ofrece cuantiosas funciones que pueden usarse por demanda, simplemente insertando la biblioteca que les contiene en el código, mediante la directiva de preprocesador **`#include`** y llamándolas allí donde se las necesita. Adicionalmente, el programador puede codificar sus propias funciones y para ello debe seguir como mínimo tres pasos interrelacionados, y no precisamente en este orden, ni necesariamente terminando uno antes de cambiar a otro:

1. Escribir un prototipo o declaración de la función.

2. Escribir una definición de la función.
3. Emplazar la llamada a la función en el lugar del código donde se requiera.

Diseño Top-Down

En el modelo Top-Down[19] se formula un resumen del sistema, sin especificar detalles. Cada parte del sistema se refina diseñando paso a paso. Cada parte nueva es entonces redefinida, cada vez con mayor detalle, hasta que la especificación completa es lo suficientemente detallada para validar el modelo. El modelo Top-Down se diseña con frecuencia con la ayuda de "cajas negras" que hacen más fácil cumplir los requisitos… (Wikipedia en español) Durante la fase de desarrollo, puede aplicarse este diseño, que va de lo general hacia lo particular.

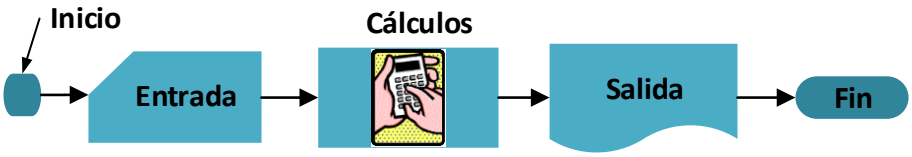


Ilustración 3. Módulo en un programa

Dentro de **`main`** se van poniendo los módulos que el algoritmo declara en cada paso a llamar como **`tags`** (*etiquetas* en español) y en el código provisional se les pone un mensaje que indique al programador que fueron usados, convirtiéndolos así en cajas negras. Si deben devolver valores, se les hace retornar algún número "mágico" (por ejemplo, **`-1`**; **`1.0`**; **`true`**; **`"OK"`**; etc.) En la medida que se resuelve el proyecto, al particularizar las acciones, los módulos se van llenando con código real, haciendo poco a poco las cajas transparentes al programador: dentro de cada uno se insertan los datos, se aplica el cálculo implicado y se devuelve la información obtenida, repitiéndose la técnica hasta terminar el encargo.

Definición de una función

Se recomienda siempre escribir la definición de una de las dos formas:

```
1.  tipo_de_dato nombre_de_función(lista_de_parámetros) { // función pura
        bloque_de_instrucciones;
        return valor; // la función devuelve un valor tipo_de_dato
    }
2.  void nombre_de_función(lista_de_parámetros) { // procedimiento
        bloque_de_instrucciones;
        [return;] // opcional: no devuelve nada y termina la ejecución
    }
```

¿Dónde se define?

Según la ODR, cada función que será usada en un programa, deberá ser definida una y solamente una vez, pero si es fichero único, como ya se dijo, hay dos opciones:

1. Declarar el prototipo antes de **`main`** y luego definirlo después (preferido):

```
long miCubo(long n);                    // prototipo
int main() {...}                        // programa
```

[19] Los métodos Top-Down fueron favorecidos en la ingeniería de software desde finales de los '60, hasta que llegó la OOP, a finales de los '80 y se usan aún. Si el perspicaz lector se ha fijado, los programas resueltos hasta el momento se han hecho siguiendo muy aproximadamente este modelo.

```
long miCubo(long n) { return n * n * n; } // definición
```

2. Definir directamente la función antes de **main**:

```
long miCubo(long n) { return n * n * n; } // definición directa
int main() {...}                          // programa
```

Información incluida en la función

Toda función debería llevar un breve comentario que especifique su propósito, cuáles son los parámetros de entrada, y que se espera a la salida. Y si su lógica no es simple, también debiera incluir su seudocódigo.

- ✓ Seudocódigo**:** descripción compacta e informal de la implementación de un algoritmo. Es independiente del lenguaje y no tiene descripción formal tipificada.

El seudocódigo utiliza las convenciones estructurales de un lenguaje de programación real, pero está diseñado para facilitar y esclarecer la lectura humana ante todo y con independencia del lenguaje de programación utilizado. Normalmente, omite detalles que no son esenciales para la comprensión humana del algoritmo. Por ejemplo, aquí se muestra un prototipo en pleno detalle, propio de una aplicación comercial:

```
/*
    Nombre:      intercambiar(p, q)
    Descripción: intercambia dos números enteros
    Por:         Simón Galliano
    Variables:   p y q para intercambiar los valores
    Entrada:     dos enteros
    Salida:      están intercambiados
    Seudocódigo:
       intercambiar(p, q):
          tmp = p
          p   = q
          q   = tmp
       fin-intercambiar
  Ejemplo: si x = 3, y = 5, al ejecutar intercambiar(x,y) -> x = 5, y = 3
  Fecha: 21/8/2019
*/
void intercambiar(int &p, int &q);
```

Y aquí uno abreviado, propio de un desarrollo particular:

```
// intercambia dos enteros
// input: dos enteros
// output: están intercambiados
void intercambiar(int &p, int &q);
```

Seguidamente se presenta otro ejemplo comparando el código C++ de una función con su seudocódigo puesto al lado:

```
1 // seudocódigo: eleva la base al exponente
2 inicio: potencia(base, expo)
3 si expo es igual a cero
4     retorna 1
5 otrosí
6     retorna base x potencia(base, expo – 1)
7 fin-potencia
```

```
1 // C++: eleva la base al exponente
2 double potencia(double base, int expo) {
3   (0 == expo)
4      ? return 1;
5      : return base*potencia(base, expo-1);
6 }
```

Ilustración 4. Seudocódigo y código

Parámetros

Constituyen la interfaz —el medio de comunicación— entre el programa y sus funciones, y van entre paréntesis tanto en el prototipo como en el cuerpo. Por ejemplo, el prototipo **`double potenciar(double base, int exponente)`** define dos parámetros: uno de tipo **`double`**, y otro de tipo **`int`**. La función es llamada desde **`main`** como **`potenciar(base, exponente);`** y devuelve un valor **`double`**.

En C++ todas las funciones llevan paréntesis. Omitirlo es un error de novato

Si una función no lleva parámetros, los paréntesis se dejan vacíos. Si lleva uno, se pone entre paréntesis. Si lleva más de uno, todos van entre paréntesis y separados por comas. Cada parámetro tiene que especificar su propio tipo y en el caso de la declaración, aunque no es obligatorio, es una buena costumbre el identificar su nombre, tratando de que las interfaces de la declaración y la definición se lean de igual modo.

El tamaño de la interfaz (cantidad de parámetros empleados) es una medida cualitativa de la generalidad de una función, su independencia y la fortaleza de su diseño: a menor interfaz más general e independiente es la función y mayor fortaleza tiene en su diseño, características muy deseables. Por ello se prefieren las interfaces lo más pequeñas posible.

Pase por valor

Al ser pasados los parámetros, por lo común el sistema lo hace enviando copias a la función que los usa, y si esta los cambia, entonces los verdaderos valores no se alteran. A este mecanismo se le llama *Pase por Valor*. El pase por valor a veces es fuente de ineficiencia, pues en caso de un parámetro muy grande, o muy complejo, o llamado muchas veces, o una mala combinación de lo antes dicho, la acción de copiar el dato a la función y destruir las copias al término, tomará un tiempo que puede llegar a ser perceptible.

Sea el prototipo **`void respuesta(int x)`** que necesita del entero **`x`** para dar una respuesta (¿cuál? a los efectos de esta discusión, no importa), pero si la función usa ese número en su cuerpo y lo cambia, la cuantía original queda intacta en el programa, porque fue pasada por valor.

Pase por referencia

Otras veces se pasa la dirección del parámetro mediante un puntero, y el valor es tratado directamente. A este otro mecanismo se le llama *Pase por Referencia*. Es más eficiente y como valía añadida, permite a una función retornar más de un valor a través de su interfaz, pero tiene la desventaja que el valor pasado puede ser alterado, obteniendo como resultado un posible efecto colateral en el trabajo de la función.

Si en una determinada función la variable cambia su valor y se quiere que también cambie el valor original, esto sugiere un pase por referencia. Y aunque puede usarse un puntero, como se dijo en el párrafo anterior, el mecanismo más cómodo, simple, preferido y mayoritariamente empleado en C++ es pasar la variable afectada del operador de referencia directa **`(&)`** que toma la dirección de memoria de la variable que afecta.

- ✓ Una referencia es un apodo, un alias. Una referencia es constante y al crearla se inicializa con un nombre y desde ese instante, la referencia se “refiere” a ese nombre y todo lo que se la haga a la referencia, se le hace al nombre.

La referencia se declara escribiendo el operador inmediatamente afectando al elemento afectado, e iniciándola al instante de su creación. Por ej.:

```
int edad;
int &rEdad = edad;
```

Al usar una referencia como parámetro, se antepone el operador de dirección al nombre del parámetro. Al usar dicho valor, la función lo toma directamente del lugar de la memoria donde el programa lo puso. Si lo altera, lo está haciendo sobre la misma variable. Por ej.:

```
void respuesta(int &x);
```

¿Y cómo evitar que al pasar una referencia, un puntero o un arreglo, la función altere el valor original en el cuerpo del programa, si no se desea? Sencillo. Las referencias siempre son constantes, por lo tanto ¡hacer constante la variable!

```
void respuesta(const int &x);
```

Si ahora se intenta cambiar el valor de **x**, el compilador dará error.

La regla práctica cuando se escribe la interfaz de una función es:

1. Si hay que alterar a cualquiera de los valores originales, se pasa por referencia.
2. Los tipos de datos complejos siempre se pasan por referencias y si no deben ser alterados, haciendo constante la referencia.
3. Los datos simples, nativos del sistema (**int**, **float**, **char**, etc.) siempre se pasan por valor y si no hay que alterarlos, el programador decidirá si se cualifican como valores constantes, o no.

Nunca se ha de reasignar una referencia, ¡porque no se puede! Sea el siguiente código:

```
int edad = 25;      // años
int peso = 83;      // kg
int &rEdad = edad;  // rEdad vale 25
&rEdad = peso;      // ¡edad ahora vale 83!
```

Nota: La referencia es de naturaleza constante. Declarada y definida al mismo tiempo, ya no puede ser cambiada para referir a otra variable.

¿Referencias o punteros?

La buena codificación en C++ favorece el uso de referencias, más claras de leer, más simples de usar y mejor ocultadoras de información, pero hay un problema si es necesario usar primero un elemento y luego otro en el mismo módulo: las referencias no pueden ser reasignadas y entonces se impone el uso de punteros.

Por otra parte, es un grave error hacer una referencia nula. Si existe una remota posibilidad de que el elemento sea nulo, no puede usarse directamente una referencia. Hay que usar un puntero o codificar por el error. Por ej.:

```
int *pInt = new[20](std::nothrow) int;
if (pInt != nullptr) {
    int &rInt = *pInt;
} // if
```

Valores por omisión

Ahora bien, a partir de la extrema derecha de la lista de parámetros y retrocediendo hacia la izquierda, pueden darse valores por omisión a los parámetros de la declaración de la función siempre que se vayan dando los valores de derecha a izquierdas a cada parámetro, sin saltarse ninguno:

- No se tienen que dar valores a todos; puede detenerse donde desee.
- El valor dado y el tipo de parámetro deben coincidir plenamente.

[20] Ver el uso de **new** en el tema 13 - MANEJO DINÁMICO DE LA MEMORIA

Según (Schildt, C++. Guía de Autoenseñanza, pág. 140), "…*el uso de argumentos implícitos, básicamente, es una forma abreviada de sobrecarga de funciones*", porque si se omiten los valores, el compilador usa los que la función trae por omisión, aunque obedeciendo a una disposición rígida. Por ejemplo, sea una declaración de función:

```
void miFuncion(float par1 = 0.1, int par2 = 1, char par3 = 'z');
```

cuya definición podría ser:

```
void miFuncion (float par1, int par2, char par3) {
    cout  << "parámetro 1 = " << par1
          << "parámetro 2 = " << par2
          << "parámetro 3 = " << par3 << '\n';
}
```

Dando valores en ese orden a las variables: `par1 = 9.9, par2 = 8, par3 = 'A'`, sus posibles llamadas son:

Tabla 17. Uso de argumentos implícitos

Nº	Llamada	Valores salientes	Pase
1	`miFuncion(par1, par2, par3);`	`par1 = 9.9, par2 = 8, par3 = A`	Los 3 parámetros son pasados.
2	`miFuncion(par1, par2);`	`par1 = 9.9, par2 = 8, par3 = z`	Los 2 primeros son pasados.
3	`miFuncion(par1);`	`par1 = 9.9, par2 = 1, par3 = z`	El primero es pasado.
4	`miFuncion();`	`par1 = 0.1, par2 = 1, par3 = z`	Ningún parámetro es pasado.

Sólo se pueden omitir los argumentos de derecha a izquierda, y solamente en el orden dado en la declaración, como muestra la tabla; cualquier otra forma de pasar los parámetros es ilegal y da error de compilación:

```
main.cpp:nn expected primary-expression before ',' token
```

Esta característica trabaja muy bien, ofreciendo funciones adaptables para tareas típicas, simplificando código y eliminando la necesidad de sobrecargarlas (ver en la página 82), pero les resta flexibilidad porque hay que respetar un orden estricto de pase de parámetros. Por otra parte, los valores por omisión posibilitan la creación de interfaces intuitivas y fáciles de usar. Hay que pensar muy bien en cada caso cuál es la vía más adecuada.

Funciones matemáticas

Si hay cálculos en su código, entonces debe asegurarse que las matemáticas están bien. Hay centenares de ejemplos en que aplicaciones han ofrecido información incorrecta basada en una mala codificación matemática, lo cual puede traer efectos desastrosos en dependencia de lo que se quiere resolver. Consejo: compruebe una y otra vez las ecuaciones que ha codificado.[21] (Linux Tricks and Tips, pág. 65)

Es común en muchos problemas tener que aplicar alguna función matemática a una variable numérica. Traducir una cierta función matemática a código C++ puede ser algo no trivial que requiera programación de altos vuelos. Sin embargo, en la mayoría de las veces esto se puede obviar usando funciones predefinidas, tanto de C++ como de terceros especializados. En la tabla que sigue se muestran las funciones matemáticas más comunes, que el sistema ofrece en una biblioteca de fácil uso por cualquiera y avalada por años de aplicación: `<cmath>`

Tabla 18. (simplificada) Funciones de <cmath>

Función	Resultado
`n = abs(v)`	Devuelve el valor absoluto de **v**
`n = exp(x)`	Devuelve e^x `(e ≈ 2.7183)`
`n = sqrt(x)`	Raíz cuadrada de **x**.

[21] El humilde programador no tiene por qué ser un matemático de excelencia. Ante la duda, buscar ayuda profesional.

	Si (**x < 0**) hay error de dominio
n = pow(x, y)	Devuelve x^y. Si **(x = 0)** & **(y ≤ 0)** o si **(x < 0)** & (**y** no es entero) hay error de dominio Si el resultado es un desbordamiento (overflow) hay error de rango
n = log(x)	Logaritmo natural (base **e**) de **x**. Si **x** es negativo hay error de dominio; si **x** es cero hay error de rango
n = log10(x)	Logaritmo (base **10**) de **x**. Si **x** es negativo hay error de dominio; si **x** es cero hay error de rango
n = ceil(x)	El menor entero no menor que **x**
n = floor(x)	El mayor entero no mayor que **x**
n = fmod(x, y)	Devuelve el resto de **x/y**
n = modf(x, y)	Divide **x** en dos partes; la parte fraccional va a **n**, y carga en **y** la parte entera
n = sin(x)	Seno de **x** (**x** en radianes)
n = cos(x)	Coseno de **x** (**x** en radianes)
n = tan(x)	Tangente de **x** (**x** en radianes)
n = asin(x)	Arco seno de **x** (-1 ≤ x ≤ 1)
n = acos(x)	Arco coseno de **x** (-1 ≤ x ≤ 1)
n = atan(x)	Arco tangente de **x**
n = atan2(x, y)	Arco tangente de **x/y**
n = sinh(x)	Seno hiperbólico de **x**
n = cosh(x)	Coseno hiperbólico de **x**
n = tanh(x)	Tangente hiperbólica de **x**
Estas funciones numéricas están contenidas en la biblioteca <cstdlib>	
tmp = div(x, y)	**tmp.quot()** es el cociente; **tmp.rem()** es el resto de la división entera **x/y**
srand(time(0))	Nueva secuencia para la generación de números aleatorios
n = rand()	Próximo número de la secuencia

Notas:

- Todos los ángulos son dados u obtenidos en radianes.
- La sobrecarga para los valores enteros y de punto flotante unificó varias funciones. Por ejemplo, **abs(), fabs()** y **labs()** ahora ISO C++ las maneja como plantilla **abs()**.
- Si ocurre un sumidero (**underflow**), se lanza la macro **-HUGE_VAL**. Si ocurre un desbordamiento (**overflow**), se lanza la macro **HUGE_VAL**. En ambos casos la variable global de sistema **errno** toma el valor de **ERANGE**, indicando un error de alcance.
- La estructura **type div_t** tiene dos campos:

```
1.  int quot; // guarda el cociente
2.  int rem;  // guarda el resto de la división
```

Función inline

Cuando se define una función el compilador crea un conjunto de instrucciones en memoria para gestionarla. Cuando la función es llamada, el flujo del programa salta a esas instrucciones y cuando la función retorna, el flujo regresa a la instrucción que sigue al punto de llamado[22]. Todo eso en la computadora consume tiempo y memoria.

Si una función es muy pequeña (que no implica ser escrita en una sola línea, aunque ayuda) y adicionalmente se emplea pocas veces, es candidata a ser codificada afectándola antes de su declaración con el especificador **inline**, quien sugiere al compilador que la expanda en el punto de llamada, lo cual evita la sobrecarga de ida y vuelta a expensas de engrosar

[22] Es como ésta llamada a pie de página que le hace bajar sus ojos un tiempo acá, antes de volver al punto donde se quedó. Si la función es llamada varias veces, el proceso se repite y se consume tiempo.

exiguamente el tamaño del programa ejecutable. La pega es que hay que repetir la definición de una función **`inline`** en cada fichero donde se use.

Pero al final es el compilador quien tiene la última palabra: el programador propone, pero aquél dispone si convierte o no a la función. Es más, los buenos compiladores de hogaño pueden optimizar por su cuenta, convirtiendo funciones adecuadas al cambio sin que se les diga explícitamente, siempre que no se altere la semántica[23] del programa.

Algunos factores impiden que el compilador expanda una función a **`inline`**. Los más comunes se refieren a funciones recursivas, o muy complejas, o muy grandes, o con ciclos, o con **`switches`**, o con variables estáticas o una mala combinación de todo lo antes dicho.

En C++ se prefiere sustituir las macros por funciones **`inline`**

Optimizaciones

Si busca mejorar la eficiencia de alguna manera, siempre mire en primer lugar a sus algoritmos y estructuras de datos. Ellos marcan el orden de magnitud de la mejoría general, mientras que las optimizaciones de procesos, tal como el uso de funciones **`inline`**, generalmente…conllevan a resultados más discretos. Sólo diga "no por ahora." (Sutter, More Exceptional C++, pág. Item 12)

En los programadores comúnmente las ideas de la función **`inline`** y la optimización van en yunta, pero al hablar de optimizaciones, antes hay que aclarar muy bien que se entiende por ello: ¿se habla del tamaño del programa, del uso de la memoria, del tiempo de compilación y/o ejecución? ¿de algo más? Como una guía a seguir en estos textos de nivel elemental, aquí está un consejo inestimable frente al proceso de optimización: ¡ni lo piense! Eso debe hacerse aplicando un perfilador comercial a la aplicación terminada antes de comercializarla.

Codificación modular ideal

Durante la fase de diseño, típicamente se organizan las tareas básicas que dan solución al problema, y luego se las codifica como funciones o clases[24] que manejan dichas tareas. El concepto de modularidad se extiende a crear bibliotecas particulares y a dispersar los códigos de los módulos en otros ficheros integrados al programa.

Una codificación modular ideal de un programa sería llamar a las funciones básicas necesarias desde **`main()`**, que sólo publicaría la lógica, la estructura de primer nivel, pero siempre teniendo presente el Zen de Python: *lo práctico gana a lo puro*.

Tratar siempre de codificar los programas como módulos ideales

Programa 11: programa modular

Se requiere elevar un número cualquiera a una potencia entera. Se conoce que $a^n = \overbrace{a \times a \times a \ldots}^{\textbf{n veces}}$ En este caso a es la base real y n el exponente entero. Codificar un programa modular que realice el trabajo y lo siga haciendo mientras que el usuario lo desee.

[23] Parte de la lingüística que estudia el significado de las expresiones.

[24] El desarrollo por clases se estudiará en la tercera parte del texto.

Este es un caso verdaderamente simple, pero que de paso ilustra la aplicación del diseño modular Top-Down, que funcionó muy bien por alrededor de 20 años, hasta que la complejidad a manejar superó sus límites, dando paso a otras metodologías, entre ellas a la OOP.

Las funciones (módulos) definidas se adaptaron del código antes dado en la página 56. A partir de la tabla se crean los módulos y se enlazan entre sí. Generalmente la entrada de datos y la muestra de información se hacen mediante procedimientos, y los cálculos a través de funciones propiamente dichas.

Tabla 19. Comunicación entre módulos

Nombre del módulo	Parámetros	Tarea	Valor retornado
entrarDatos	**double &base** **uShort &exponente**	Los datos al programa.	Ninguno.
calcularRespuesta	**double base** **uShort exponente**	Calcular la respuesta.	(**double**) Respuesta.

Programa

```
#include <iostream>
#include <cstdlib>
#include <cassert>

typedef unsigned short uShort; // el nuevo nombre sólo es un apodo

//  prototipos de los dos módulos principales:
void entrarDatos( double & base, uShort & exponente );      // entrada de datos
double calcularRespuesta( double base, uShort exponente ); // función de cálculo

int main() {
  //  las variables se declaran fuera del ciclo
  double laBase;
  uShort elExponente;
  char sn;
  std::cout << "Programa 11: potencia un n\243mero.\n\n";

  while ( true ) {
    entrarDatos( laBase, elExponente );
    std::cout << "Respuesta = " << calcularRespuesta( laBase, elExponente )
        << "\n\n\250Una vez m\240s? (s/n) > ";
    std::cin >> sn;//  ¿seguimos?

    if ( 'N' == toupper(sn) ) {
      return EXIT_SUCCESS;
    } // if
  } // while
}

// entra la base y el exponente
// se asegura de que el exponente sea mayor o igual a cero
void entrarDatos ( double & base, uShort & exponente ) {
  // entra la base
  std::cout << "Escriba el n\243mero: ";
  std::cin >> base;
  // entra el exponente
  std::cout << "Escriba el exponente entero: ";
  std::cin >> exponente;
  // se asegura de que el exponente sea mayor o igual a cero
  assert ( exponente >= 0 );
}
```

```
// hace el ciclo y cálcula la potencia
double calcularRespuesta ( double base, uShort exponente ) {
  double respuesta = 1; // neutro del producto

  for ( uShort i = 0; i < exponente; ++i ) {
    respuesta *= base;
  } // for

  return respuesta;
}
```

Salida

```
Programa 11: potencia un número.

Escriba el número: 5.01
Escriba el exponente entero: 2
Respuesta = 25.1001

¿Una vez más? (s/n) > n
```

Funciones recursivas

Recursión o recursivo es la forma en la cual se especifica un proceso basado en su propia definición. Si un problema puede ser definido en función de su tamaño, entonces puede ser dividido en instancias más pequeñas a las cuales se les conoce la solución explícita (casos base) y se puede suponer que quedan resueltas. (Wikipedia en español)

C++ permite la recursividad. Al igual que los ciclos, una función recursiva no puede llamarse *ad infinitum*: o se encuentra una condición válida de terminación, o se agotan los recursos que el sistema tiene a mano y el programa termina con un fallo catastrófico.

Comparándolo contra el secuencial, el código recursivo es más simple de escribir, pero a la vez, más difícil de concebir y en muchas ocasiones, más arduo de entender. En resumen, es a veces muy ofuscado y se recomienda cuidado y discreción en su empleo. Por lo demás, es otra herramienta en el arsenal del programador, elegante de codificar, rápida de escribir (que no de concebir) y adecuadamente eficiente si la recursividad es sencilla.

El modelo de la programación recursiva simple es llamar a la misma función cada vez de forma más simple, hasta llegar a un caso límite de salida.

Existen programaciones recursivas que no son simples y no siguen al modelo antes dado. Ejemplos clásicos son el cálculo doblemente recursivo del número de Fibonacci, o el cálculo recursivo múltiple de las Torres de Hanói. En cada caso las llamadas se multiplican y eso es un verdadero ataque a la pila de control del programa, haciendo de esta técnica una mala elección si muchos niveles de recursión son necesarios.

- ✓ Pila de control del programa. Área de memoria donde se guardan las variables automáticas que son construidas en su punto de definición dentro de un bloque y destruidas inmediatamente que salen de ámbito.

Tampoco se debe perder de vista que la recursión nunca es más eficiente que los procesos cíclicos porque consume más recursos de espacio (más memoria) y de tiempo (más llamadas a la pila.) Por otra parte:

1. Existen escenarios donde se requiere visitar ciertas estructuras de datos (colas, listas, pilas, etc.) y como no hay un número fijo de elementos a procesar, hacerlo se torna dificultoso, por decirlo así.

- ✓ Visitar una estructura significa que dentro de un ciclo se aplicará a todos y cada uno de sus componentes una o más operaciones.

2. Se deben ordenar o buscar valores en cantidades masivas de datos y precisamente, en estos casos los métodos de divide-y-vencerás, normalmente basados en la recursión, son más eficaces que los cíclicos.

Los métodos recursivos se adaptan muy bien a estos dos tipos de tareas

Ante el dilema, recordar la filosofía del Zen de Python: *explícito es mejor que implícito; simple es mejor que complejo; plano es mejor que anidado; la legibilidad cuenta; lo práctico gana a lo puro;* y hacer un balance de esas recomendaciones.

Programa 12: programa recursivo

Se requiere elevar un número cualquiera a una potencia entera. Se conoce que $a^n = \overbrace{a \times a \times a \ldots}^{\text{n veces}}$ En este caso a es la base real y n el exponente entero. Codificar un programa modular que realice el trabajo recursivamente y lo siga haciendo mientras que el usuario lo desee. Se parte de repetir la llamada a la función de exponenciación, pero disminuyendo cada vez el valor del exponente, hasta que al llegar a cero se sabe con certeza que el valor de la función es uno, ya que toda base elevada a 0 vale 1, es decir, $x^0 = 1$.

En este caso, solamente se actúa en el módulo de cálculo, cambiando el algoritmo, y es lo que acá se refleja. Ventajas de la programación estructurada (modular), que la OOP eleva a límites superiores, como se verá en el tomo 3 de la serie.

```
// cálcula la potencia recursivamente
double calcularRespuesta ( double base, uShort exponente ) {

  return ( 0 == exponente )
     ? 1
     : base * calcularRespuesta( base, exponente );
  } // for
}
```

Y salvo por el encabezamiento, con los mismos datos la salida es la misma del programa anterior.

Sobrecarga de funciones

La sobrecarga de funciones proporciona un tipo básico de polimorfismo que se resuelve en tiempo de compilación utilizando sus firmas. Cuando dos o más funciones comparten el mismo nombre, en el mismo ámbito, pero tienen diferente firma, se dice que están sobrecargadas pues comparten interfaz única, pero se aplican en ocasiones diferentes.

- ✓ Sobrecarga en esta área se refiere a tener dos o más funciones en el mismo ámbito, con el mismo nombre, pero con diferentes interfaces.
- ✓ Firma de la función es el conjunto formado por el nombre y la lista de sus argumentos.
- ✓ Polimorfismo, es la cualidad representada por una interfaz, pero muchos métodos.

Las funciones sobrecargadas pueden ayudar a reducir la complejidad de un programa permitiendo, mediante un mismo nombre, que se referencien diferentes acciones relacionadas de alguna suerte entre sí, cada una de ellas asociada a una firma distinta. Una interfaz (un mismo nombre) y muchas funciones aplicadas en acciones similares (muchos métodos), usadas conjuntamente con la técnica de los valores por omisión, posibilita la creación de interfaces intuitivas, fáciles de usar y aportadoras de estandarización. Nuevamente, hay que pensar muy bien en cada caso cuál es la vía más adecuada.

¿Cuándo usarla?

Uno de los principales usos de la sobrecarga de funciones es conseguir polimorfismo en tiempo de compilación, que se ajusta a la filosofía de una interfaz, muchos métodos. (Schildt, C++. Guía de Autoenseñanza, pág. 32) Al emplear la sobrecarga, el programador debe limitarse a funciones que hacen básicamente la misma tarea o tareas similares, pero que necesitan diferentes parámetros o diferentes algoritmos, o no tienen uno o varios valores por omisión razonables o claros.

Como metodología de diseño y técnica de programación, si se emplea, debe reservarse para enfrentar una tarea verdaderamente polimórfica, es decir, que requiera varios juegos de datos distintos para cumplir un mismo objetivo u objetivos similares.

Corolario: si dos funciones no hacen tareas similares, no debe de aplicárseles sobrecarga, pues eso es fuente segura de ofuscación y punto de partida inequívoco para la generación de errores.

¿Cómo usarla?

Es muy fácil sobrecargar una función: simplemente hay que declarar y definir todas las versiones requeridas. Cuando aparece una en el flujo del programa, el compilador C++ seleccionará automáticamente la versión correcta a llamar, en base a su firma. En un mismo espacio de nombres (mismo ámbito), dos o más funciones pueden compartir el mismo nombre si difieren en su firma, es decir, en el tipo de sus argumentos, en el número de sus argumentos, o en la posición de sus argumentos en la lista, o en una combinación de todas esas condiciones.

a) Al comprobar la firma de la función, se considera que un tipo de dato dado, por ejemplo, **`(x)`** y su referencia **`(&x)`** son lo mismo: la llamada es ambigua y hay error de compilación.
b) El tipo devuelto tampoco representa diferencia. Si dos funciones difieren solamente en el tipo de datos que devuelven, el compilador no puede distinguir entre sus firmas; no puede elegir a quién llamar: la llamada es ambigua y hay error de compilación.

Programa 13: sobrecarga de funciones

La hora juega un papel importante en documentos públicos. Puede pedirse como una cadena ANSI C/C++ **`("20:22")`**, como números individuales **`(20, 22)`**, o incluso con un patrón de entrada (que no será visto aquí) pero al final se desea ver su representación unificada **`("8:22 pm")`** Codificar un programa que resuelva esta tarea.

Desafortunadamente, para resolver el problema con facilidad y sencillez, EODA se debe usar la función **`strtok()`** que extrae de una cadena ANSI C palabras separadas por punteros delimitadores. Acá serán gentilmente mostrados, pero estudiados con detenimiento en el tópico destinado a los punteros. Y se recomienda al aplicado lector que cuando los domine, regrese a asimilar detenidamente el programa.

Ver en la Tabla 8 el prototipo de la función **`atoi`**, que toma una cadena de números y retorna su valor entero. Ver en la misma tabla el prototipo de la función **`strtok`**, que desmiembra una expresión en sus partes constituyentes, o tokens, y devuelve un valor que contiene una palabra extraída. Para descomponer la frase en palabras aisladas hay que llamar a **`strtok`** varias veces. La primera pasando la cadena como el primer parámetro:

```
tokenPtr = strtok(hora, delimitador);  // se fija el primer token
hh = atoi(tokenPtr);                   // la palabra se guarda como el entero hh
```

y el resto de las veces, pasando como primer parámetro al puntero nulo. Aunque este no es el caso, usualmente eso se hace dentro de un ciclo, que cuando termina la descomposición total de la frase, retorna un puntero nulo como centinela

de término. Además, en la biblioteca **<ctime>** se define el tipo de datos **clock_t**, que se usa para representar una hora dada.

Para vencer el problema serán aplicadas varias técnicas interesantes que vale la pena estudiar con detenimiento, recordar y utilizar donde quiera que se den las condiciones, pues simplifican y acortan la escritura, aportando robustez, fiabilidad y precisión al código:

1. Se hará una función principal que toma dos números enteros y se encarga del cálculo final. Se le pasará el dato desde las otras, encadenándolas. Esto concuerda con la filosofía del Zen de Python: *simple es mejor que complejo; y plano es mejor que anidado*. Ahí se hará algo simple de programación defensiva que no es ni intolerante, ni correctora, pero justo en la línea de la OOP, para evitar datos ilegales, lo cual concuerda con la filosofía Zen de Python: *nunca debería dejarse pasar un error silenciosamente*.
2. Seguidamente se codificará la cadena clásica, con mucho la función más compleja de las tres. Se tomará el dato y con **strtok** se le extraerán los tokens apropiados; cada uno de ellos será convertido en un número entero con la función **atoi** y será puesto en un valor numérico, lo cual concuerda con la filosofía Zen de Python: *lo práctico gana a lo puro; y complejo es mejor que complicado*. Luego éstos serán pasados a la primera función, lo cual está de acuerdo con la filosofía del lenguaje C: *dividir una gran tarea en subtareas, y repetir el proceso hasta conseguir funciones pequeñas y sencillas*; y del C++: *crear varias unidades o clases y construir el programa con ellas*.
3. Por último, la cadena de C++ se convertirá a clásica mediante la función miembro **c_str**, mostrada en la Tabla 10, y será pasada a la segunda función. Pero hay que remover su "constancia" antes de pasarla al puntero, por lo que hay que encasillar con **const_cast** antes de pasarlo, siguiendo filosofía Zen de Python: *simple es mejor que complejo.*

Este ejemplo es con mucho el más complicado hasta ahora visto, y es una muestra de la no-linealidad al enseñar/aprender computación: se aplican elementos dispersos que serán propiamente explicados más adelante, aun cuando se centre en el uso de funciones sobrecargadas, que es lo que hay que dominar en este momento. Se recomienda al lector volver a repasarla de cuando en cuando…

Programa

```
#include <iostream>
#include <cstdlib>
#include <string>
#include <ctime>    //  para clock_t
#include <cstring>  //  para strcpy(), strtok()

//  imprime(): imprime la hora
//  input: dos enteros hora y minuto
//  output: la hora a consola
void imprime ( clock_t hora, clock_t minuto );

//  imprime(): imprime la hora
//  input: una cadena clásica
//  output: la hora a consola
void imprime ( char * hora );

//  imprime(): imprime la hora
//  input: una cadena C++
//  output: la hora a consola
void imprime ( std::string hora );

int main() {
  //  información:
  std::cout <<
  "Programa 13: sobrecargando funciones.\n\n"
```

```
  "Este programa muestra 3 formas de tomar\n"
  "la hora para ponerla en consola:\n\n"
  "1) cadena C:\t25:44 (mal formato)\n"
  "2) cadena C++:\t 9:45 (por la ma\244ana)\n"
  "3) n\243meros:\t21,46 (por la tarde)\n\n";

  //  hora clásica;
  const short SIZE = 6;           // tamaño máximo
  char hora1[SIZE];               // variable
  std::strcpy ( hora1, "25:44" ); // valor erróneo: no hay hora 25
  std::cout << "1) cadena C:\t";  // información
  imprime( hora1 );               // respuesta

  //  cadena del C++
  std::string hora2( "9:45" );     // variable + valor
  std::cout << "2) cadena C++:\t"; // información
  imprime( hora2 );                // respuesta

  //  enteros en directo
  std::cout << "3) n\243meros:\t";
  imprime( 21, 46 ); // variable + respuesta
  std::cout << '\n';

  system( "pause" );
  return EXIT_SUCCESS;
}

// esta función imprime la hora en formato h:m y cuenta con programación
// defensiva tipo OOP: cualquier valor extraño pone la hora a cero
void imprime( clock_t h, clock_t m ) {
  //  se comprueban los límites
  if ( (h < 0 or h > 24) or (m < 0 or m > 60) ) { // mal hora o mal minuto ...
    h = m = 0;                                    // ... todo a cero
  } // if

  std::string meridiano(" am"); // fija el meridiano en la mañana

  // si la hora es más de 12, le quita 12 y arregla el meridiano
  if ( h > 12 ) {
    h -= 12;
    meridiano = " pm";
  } // if

  //  saca a consola
  std::cout << h << ":" << m << meridiano << "\n";
}

// se usa la función strtok() que extrae 'tokens', para poder separar
// las horas de los minutos; cada uno de esos tokens se pasa a un
//  valor con atoi() antes de enviarlo a la impresión numérica
void imprime( char * hora ) {
  // así se asignan los dos punteros que usa strtok (fíjese en el *):
  const char * delim = ":"; // un puntero al delimitador; pueden ser varios
  char * tokenPtr;          // un puntero a la palabra

  // se desmiembra manualmente la palabra en caracteres numéricos:
  // se fija el primer token, se pasa a atoi() y se guarda como hora
  clock_t hh = atoi(strtok(hora, delim ));
  // se fija el otro token, se pasa a atoi() y se guarda como minuto
```

```
  clock_t mm = atoi(strtok(nullptr, delim));

  imprime( hh, mm ); // los números se pasan a la impresión numérica
}

//  se pasa la cadena de C++ para una ANSI C constante con la función
//  cadena.c_str() y se le remueve la constancia con el encasillamiento
//  y se pasa a la impresión de cadena clásica
void imprime( std::string hora ) {
  imprime( const_cast<char *>(hora.c_str()) );
}
```

Salida

```
Programa 13: sobrecargando funciones.

Este programa muestra 3 formas de tomar
la hora para ponerla en consola:

1) cadena C:    25:44 (mal formato)
2) cadena C++:   9:45 (por la mañana)
3) números:     21,46 (por la tarde)

1) cadena C:    0:0 am
2) cadena C++:  9:45 am
3) números:     9:46 pm
```

10 - COMPILACIÓN SEPARADA

> *A principios de los '80, el software se desarrollaba usando programación estructurada, establecida para ayudar a dividir los programas en pequeñas partes o módulos, haciendo más fácil el desarrollo cuando la aplicación crecía. (Wikipedia en español)*

Hasta ahora los programas de ejemplo han sido presentados en codificación monolítica, pero eso no es para nada lo que sucede en la vida real. Al contrario, un proyecto serio estará hecho en codificación separada, con su código distribuido en cientos, miles y quizás más ficheros. Por ejemplo, *el SO Windows Vista™ se codificó con C++, partiendo de unos 40 millones de líneas de código.* (Wikipedia en español) Sin compilación separada, eso sería... ¡imposible!

- ✓ Codificación monolítica: codificación de un programa que está escrita totalmente en un fichero que, desde luego, también contiene a **`main`**.
- ✓ Codificación separada: codificación de un programa que está escrita en múltiples ficheros, que serán compilados aparte y le darán soporte. Por supuesto, uno y sólo uno de ellos contiene a **`main`**.

Tipos de codificación

Son dos: el estilo monolítico y el de la compilación separada. El código monolítico, sencillamente no es una opción viable para solucionar un problema serio, realmente complicado, grande o todas esas cosas, debido a su dificultad inherente fundamentada sobre todo en tres simples razones basadas en el entrecruzamiento de sus instrucciones, presentes porque son necesarias, si no, no estarían allí.

Ese tipo de código es difícil de:

1. Escribir si ocupa más de 150–200 líneas en su editor. Entre más líneas haya, se hace más necesario controlar correctamente el código ya hecho, antes de seguir añadiendo, convirtiéndole en un sumidero de tiempo.
2. Depurar. Un error lógico (*bug*) puede ser codificado ahora, pero aparecer cuando se usa la aplicación mucho más tarde, o en muchos otros lugares, etc., reclamando entonces, tiempo extra de mantenimiento.
3. Escalar. Al estar todo junto y aun habiendo sido escrito con extrema claridad, pronto se hace ininteligible para el lector humano la lógica global, y la producción del código punto menos que imposible, obligando a buscar otras soluciones que necesariamente consumirán tiempo y recursos adicionales.

Los módulos que formarán el proyecto en compilación separada, vendrán casi siempre por parejas de ficheros[25]. En uno de ellos, llamado cabecera (*header* en inglés), con extensión estándar **`.h`**, se pondrán los prototipos o declaraciones. En el otro, con extensión típica **`.cpp`**, y que se recomienda escribir preferiblemente con el mismo nombre del anterior, se pondrán las definiciones, o sea, el código de los susodichos prototipos.

En un escenario comercial el fabricante del producto distribuye las definiciones pre-compiladas y las cabeceras en formato de texto, de modo que el comprador del producto pueda saber cómo usar las funciones y hasta formarse una idea de cómo trabajan, aunque no exactamente cómo ellas hacen su tarea. Este mecanismo forma parte de la ocultación de información y del principio del privilegio mínimo, características de diseño deseables en los programas contemporáneos y muy presentes en la OOP.

- ✓ Ocultación de información: es el concepto que conduce a la ocultación de decisiones de diseño en un programa, con el objetivo de proteger a otras partes del código cuando ocurren los inevitables cambios.
- ✓ Principio del privilegio mínimo: disciplina de diseño que requiere que cada módulo sólo pueda acceder a la información y recursos que le son necesarios para realizar su tarea y nada más.

Ambas características no están presentes *per se*. Al ser C++ un lenguaje híbrido, es obligación del codificador mantener su presencia en el trabajo, lo cual depende mucho de la destreza y conocimientos personales.

Los módulos que contribuyen a solucionar una tarea lo más general posible se concentran en paquetes que a su vez, agrupados por categorías forman bibliotecas. Tanto los ficheros que definen, como el programa que los emplea, incluirán dichas cabeceras para poder usar las funciones que necesiten.

En lo adelante, excepto en casos muy simples, se trabajará en compilación separada

Las extensiones ya vistas (**`.h`** y **`.cpp`**) son el estándar en C++, aunque existen otras comúnmente aceptadas por diferentes compiladores. Para Code::Blocks están:

- Declaraciones .....**`*.hh, *.hpp`** y **`*.hxx`**
- Definiciones .....**`*.cc`** y **`*.cxx`**

Se recomienda apegarse al estándar, pero ante la duda, consultar el manual de su IDE

Ficheros múltiples

Como ya se ha dicho, la típica programación en C/C++ no se codifica en un sólo fichero, antes bien, se distribuye en muchos. Los mínimos son al menos dos, usualmente tres, el principal y los dos que recogen declaraciones y definiciones:

1. Un fichero principal con la función **`main()`**, para el arranque del programa.

[25] Un caso es el de las plantillas que tienen que venir completas en un mismo fichero. Otros casos son que a veces sólo se requiere de una cabecera declarando valores que serán usados globalmente, o códigos de error/diálogos, o clases abstractas, etc.

2. Un fichero de declaración o de cabecera (*header file* en inglés), donde van los prototipos de las estructuras, uniones, clases, funciones definidas por el usuario y constantes globales.
3. Un fichero de definición, pareja del anterior, donde va la implementación de lo antes declarado.

La función es considerada unidad atómica de código en C++. Se puede —y se debe— tener su declaración en un fichero de cabecera, aparte, pero no se puede tener una parte de la implementación de una función en un fichero y la otra parte en otro u otros.

Otra implementación similar en su firma, dada por terceras personas puede venir replicada en ficheros pre-compilados si estos se tratan con la palabra-clave **`extern`**. Está claro que al final, la cantidad requerida de ficheros está en función de la agrupación lógica de los elementos que componen el programa y que en proyectos comerciales pueden llegar ser muy, muy numerosos, agrupados en muchas bibliotecas.

En la programación de C++ hay dos términos que se prestan a confusión: declaración y definición. He aquí lo que significan exactamente:

- ✓ Una *declaración* de función le especifica al compilador su nombre, su o sus parámetros y el tipo de retorno, si lo tiene.
- ✓ Una *definición* de función está constituida por el código completo de la función y el compilador le separa espacio en la memoria.

Según la ODR, en C++ sólo se puede definir la función una vez. Cuando el enlazador vincule todos los módulos, emitirá error de compilación si encuentra más de una definición para el mismo elemento.

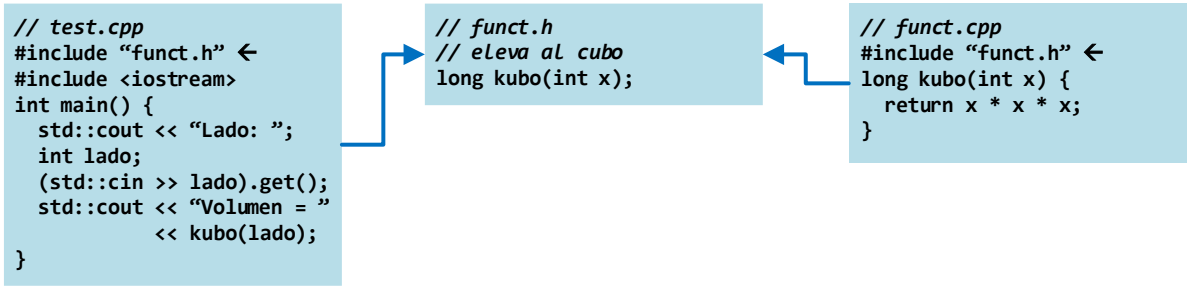


Ilustración 5. Compilación Separada

Arriba se muestra un esquema de programación en múltiples ficheros que serán compilados por separado. El IDE usado compila y enlaza automáticamente los tres ficheros produciendo un programa ejecutable, que en los SO de la familia Windows™ se llamará **`test.exe`**

Al colocar las declaraciones de funciones en un fichero de cabecera, e incluyendo ese fichero allí donde se usen —generalmente en la función **`main`**, pero puede ser en otro fichero— se asegurará una declaración consistente a través del sistema. Al incluir el fichero de cabecera en el fichero de definición, también se asegurará de que la declaración y la definición se correspondan.

El enlace entre cada cabecera y su respectiva definición es obligatorio. El enlace entre cada cabecera y el fichero que la requiera, también es obligatorio.

La forma en que se manejan los ficheros múltiples depende del entorno de programación utilizado. Embarcadero tiene su manera, Microsoft Visual Studio tiene la suya, así como Code::Blocks, que es el que aquí se usa, y en cada caso hay diferencias. No queda otra: hay que consultar el manual del IDE.

Añadiendo ficheros al proyecto

¿Cómo añadir al proyecto los ficheros necesarios desde el IDE? Cada uno tiene su forma. De nuevo aparece el caso en que no queda de otra: hay que consultar el manual del IDE. La ilustración que sigue enseña cómo se hace en Code::Blocks.

Para éste IDE basta ir al menú y pulsar **`[File]->[New]->File`**… Aparece una caja de diálogos donde se seleccionan los que se deseen, ya sea ***C/C++ header*** para las declaraciones, ya ***C/C++ source*** para las definiciones. Si esto no es así en el caso particular del lector, debe consultar el manual de su IDE.

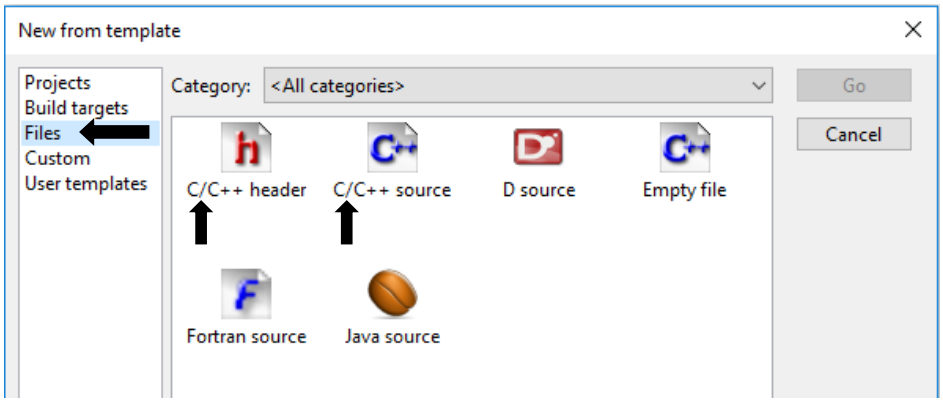


Ilustración 6. Añadiendo nuevos ficheros al proyecto

Las cabeceras

Muy a menudo los profesionales del ambiente se refieren a las cabeceras como ficheros de inclusión porque son implementados mayoritariamente como ficheros-fuente con el mismo nombre. Como otras implementaciones son permitidas, la norma es cuidadosa de no referirlos como ficheros. De toda suerte se entiende que encabezados, ficheros de encabezamiento y ficheros de inclusión son la misma cosa. (Saks)

El desarrollo de programas grandes, serios, está atado a los procesos de compilación separada y tratamiento en ficheros múltiples. Los ficheros cabecera son…inevitables. Sólo deben llevar información para el compilador (declaraciones y constantes), pero no deben incluir nada que sea aludiendo a **`namespaces`** declarados en otros sitios, o creando variables, o haciendo definiciones. Si hay que definir algo, se debe utilizar exclusivamente la cualificación con cambios de nombres hechos con **`typedef`**.

Esto es así porque un fichero de cabecera de un gran proyecto normalmente se incluye en varias unidades y si el almacenamiento para un identificador se pide en más de un sitio, el enlazador indicará en tiempo de compilación, un error de definición múltiple que conlleva a la violación de alguna de las reglas ODR.

Según (Jaeschke), los ficheros de cabecera vienen en cuatro categorías:

1. De aplicación: son codificados por el programador para resolver un problema puntual. ISO requiere que estos ficheros sean incluidos mediante la notación **`#include "nombre.h"`** y deben ser puestos separados de los anteriores, por lo general en el mismo lugar donde residen los ficheros-fuente, o en alguno de sus subdirectorios.
2. Típicos: son los definidos por el estándar vigente en su compilador, tales como **`cstdlib`**, **`iostream`** o **`cstring`**. ISO C++11 requiere que estos ficheros sean incluidos siempre mediante la notación **`#include <nombre>`**, aunque aparentemente funcione el haber puesto **`#include "nombre"`**

3. De sistemas: son dados por el IDE instalado y se usan como interfaz de comunicación con el SO, o como medio de explotar sus bondades particulares, o las del hardware presente. Por ejemplo, si el programa se gestiona desde Microsoft Visual C++, para trabajar en modo consola, éste incluirá automáticamente cabeceras y pragmas propios de la tarea.
4. De bibliotecas dadas por terceras personas, tales como los proyectos gratuitos STLPort, Boost o Blitz++. Es común que se pongan junto a las dos categorías anteriores.

Advertencia: es posible que durante el desarrollo de una aplicación se cree un módulo cuya utilidad sobrepase la vida útil del proyecto; entonces debería separarse, moviéndolo para una biblioteca miscelánea.

Se considera un mal estilo de codificación el especificar información relativa al lugar donde se ponen los ficheros incluidos. Eso se deja al sistema de desarrollo que se está utilizando, quien lo hace a través de directivas y especificidades de su entorno IDE, aunque a veces haya que quebrar la sugerencia[26]. Recordar aquí el Zen de Python: *lo práctico gana a lo puro*.

¿Por qué un envoltorio protector?

Un envoltorio protector o *header guard-word*, en inglés, es necesario porque la cabecera podría ser incluida en múltiples ocasiones, aun sin saberlo el desarrollador y entonces, cada vez que se incluya intentará definir su contenido, violando la ODR y dando error de compilación. En un proyecto grande, serio, no hay garantías de que eso no vaya a suceder. Para evitarlo, en el fichero de cabecera se usan tres directivas de preprocesador que crean un envoltorio protector de la definición de los datos. Se declara una constante específica, y si ya está definida, el preprocesador ignora la inclusión del fichero. Se termina con la directiva de finalización.

Tabla 20. El envoltorio protector de cabeceras

Directiva	Acción a tomar
`#ifndef CONSTANTE`	si no está definida la constante especificada...
`#define CONSTANTE`	...defínala aquí y finalmente...
`#endif`	...codifique el fin de este `if`

Proteger siempre los ficheros cabecera con envolturas apropiadas

Un envoltorio muy eficaz, simple y casi respetado como norma, consiste en formar su nombre tomando el del fichero cabecera, sustituyendo el punto que lo separa de la extensión por un carácter de subrayado y adicionándole la extensión, todo en mayúsculas. Por ejemplo, si el nombre del fichero es `menu.h`, el de la constante sería `MENU_H`.

Code::Blocks suministra y emplaza automáticamente un envoltorio protector y no hay razón alguna para no usarlo. En este caso el proporcionado sería `MENU_H_INCLUDED`.

Las definiciones

Al rehacer el Programa 13: sobrecarga de funciones, pero ahora en múltiples ficheros, se observa que el fichero de las definiciones lleva el mismo nombre que el de la cabecera, pero con extensión `.cpp` e incluye al primero. El que ambos ficheros lleven el mismo nombre, aunque no es de obligatorio cumplimiento, es una convención que se sugiere fuertemente porque ayuda a identificarlos por pares.

Para cada declaración de prototipos en la cabecera, en el fichero de definiciones habrá declarada una función con la interfaz exactamente igual, pero con el código apropiado. También es una convención fuertemente sugerida porque elimina ofuscación y aclara la lectura humana del código. Así se hará en el texto.

[26] Lo cual se hizo por ejemplo en los programas 28 y 29, a ver más adelante.

Notas:

- Si falta alguna declaración o definición *puede haber* error de compilación o de enlace;
- Si las interfaces que casan no son iguales, *habrá* error de compilación;
- Es lícito usar la calificación **`using`** en los ficheros de definiciones, *porque sólo afecta hasta el final de la compilación del fichero*, allí termina y no afecta a ningún otro.
- Una función **`inline`** debe ser declarada en su fichero cabecera, *pero las definiciones se hacen en los ficheros que la llaman y todas deben ser idénticas*. Si un fichero dado no la necesita, no tiene por qué definirla. Por ejemplo, si tres ficheros la requieren, se declara una vez en la cabecera, pero hay que definirla idénticamente en los tres lugares. Esto no es una violación de la ODR, es una característica del lenguaje.

Programa 14: compilación separada

Rehacer el programa anterior, pero ahora en compilación separada.

Declaraciones

- **`hora.h`**: se pasan para acá los prototipos y se ajustan los **`includes`**.

```
#ifndef HORA_H_INCLUDED
#define HORA_H_INCLUDED

#include <ctime>
#include <string>

//  imprime(): imprime la hora
//  input: dos enteros hora y minuto
//  output: la hora a consola
void imprime ( clock_t hora, clock_t minuto );

//  imprime(): imprime la hora
//  input: una cadena clásica
//  output: la hora a consola
void imprime ( char * hora );

//  imprime(): imprime la hora
//  input: una cadena C++
//  output: la hora a consola
void imprime ( std::string hora );

#endif // HORA_H_INCLUDED
```

Definiciones

- **`hora.cpp`**: se pasan las definiciones para acá y se ajustan los **`includes`**.

```
#include "hora.h"
#include <cstring>  //  para strcpy(), strtok()
#include <cstdlib>  //  para atoi()
#include <iostream> // para cout

// esta función imprime la hora en formato h:m y cuenta con programación
// defensiva tipo OOP: cualquier valor extraño pone la hora a cero
void imprime( clock_t h, clock_t m ) {
```

```
    //  se comprueban los límites
    if ( (h < 0 or h > 24) or // mala hora o ...
         (m < 0 or m > 60) ) {  // ... mal minuto ...
      h = m = 0;               // ... todo a cero
    } // if

    std::string meridiano(" am"); // fija el meridiano en la mañana

    // si la hora es más de 12, le quita 12 y arregla el meridiano
    if ( h > 12 ) {
      h -= 12;
      meridiano = " pm";
    } // if

    //  saca a consola
    std::cout << h << ":" << m << meridiano << "\n";
  }

  // se usa la función strtok() que extrae 'tokens', para poder separar
  // las horas de los minutos; cada uno de esos tokens se pasa a un
  // valor con atoi() antes de enviarlo a la impresión numérica
  void imprime( char * hora ) {
    clock_t hh, mm; // dos valores numéricos

    // así se asignan los dos punteros que usa strtok (fíjese en el *):
    const char * delim = ":"; // un puntero al delimitador
    char * tokenPtr;          // un puntero a la palabra

    // se desmiembra la palabra en caracteres numéricos:
    tokenPtr = strtok( hora, delim );     // se fija el primer token
    hh = atoi( tokenPtr );                // se guarda como hora
    tokenPtr = strtok( nullptr, delim ); // se fija el otro token
    mm = atoi( tokenPtr );                // se guarda como minuto

    imprime( hh, mm ); // los números a la impresión numérica
  }

  //  se pasa la cadena de C++ para una ANSI C constante con la función
  //  cadena.c_str() y se le remueve la constancia con el encasillamiento
  void imprime( std::string hora ) {
    imprime( const_cast<char *>(hora.c_str()) );
  }
```

Programa

```
#include "hora.h"
#include <iostream>
#include <cstdlib>
#include <cstring>

int main() {
  //  información:
  std::cout <<
  "Programa 14: sobrecargando funciones. Este programa muestra\n"
  "tres formas de tomar la hora para ponerla a consola:\n\n"
  "1) cadena C:   25:44 (mal formato)\n"
  "2) cadena C++:  9:45 (por la ma\244ana)\n"
  "3) n\243meros: 21,46 (por la tarde)\n\n";
```

```
  //  cadena clásica;
  const short SIZE = 10;           // tamaño máximo
  char hora1[SIZE];                // variable
  std::strcpy ( hora1, "25:44" ); // valor erróneo: no hay hora 25
  std::cout << "1) cadena C:\t";  // información
  imprime( hora1 );                // respuesta

  //  cadena del C++
  std::string hora2( "9:45" );      // variable + valor
  std::cout << "2) cadena C++:\t"; // información
  imprime( hora2 );                // respuesta

  //  enteros en directo
  std::cout << "3) n\243meros:\t";
  imprime( 21, 46 ); // variable + respuesta
  std::cout << '\n';

  system( "pause" );
  return EXIT_SUCCESS;
}
```

Salida

```
Programa 14: sobrecargando funciones. Este programa muestra
tres formas de tomar la hora para ponerla a consola:

1) cadena C:   25:44 (mal formato)
2) cadena C++:  9:45 (por la mañana)
3) números:    21,46 (por la tarde)

1) cadena C:    0:0 am
2) cadena C++:  9:45 am
3) números:     9:46 pm
```

TERCER BLOQUE DE EJERCICIOS

Considerar todos los tópicos anteriormente estudiados y aplicar lo necesario en cada ejercicio, pero utilizando la técnica de compilación separada.

12. En matemáticas la sucesión de Fibonacci tiene numerosas aplicaciones en ciencias de la computación, matemáticas y teoría de juegos, así como en configuraciones biológicas. Es una sucesión infinita de números naturales, donde los primeros ocho son: 0 1 1 2 3 5 8 13

    Como se ve, empieza con los números 0 y 1, y a partir de estos, la relación de recurrencia que la define es: cada término es la suma de los dos anteriores. Puesta en notación matemática:

    $$x_0 = 0;\ x_1 = 1;\ x_i = x_{i-2} + x_{i-1}$$

    Codificar un programa que pida un número límite y muestra la sucesión hasta él. Si se hace recursivamente, consume muchos recursos y en este sistema, ya para la sucesión de 35 se observa una ligera demora en la salida a consola y los números comienzan a apiñarse, por lo que se sugiere limitar la entrada hasta no más de 32, pero en este caso…es el lector quien decide.

```
Problema 12: cálculo de la serie de Fibonacci
para enteros positivos, hasta no más de 32.

0 para terminar, ¿cuántos números? 24

0       1       1       2       3       5       8       13
21      34      55      89      144     233     377     610
987     1597    2584    4181    6765    10946   17711   28657
```

13. Codificar un programa que cuente los caracteres de una cadena cualquiera (de C, o de C++) con una sola función llamada **`lenstr()`**. Al sobrecargar la función se recomienda usar las facilidades de cada tipo de cadena, ya probadas y garantizadas por el tiempo, pero es el lector quien decide. La salida debiera verse así:

```
Problema 13: cuenta los caracteres en una cadena.
Primero una cadena clásica y luego una de C++

ANSI C: ¿cadena? > mariposita
Esta cadena tiene 10 caracteres.

C++: ¿cadena? > de primavera
Esta cadena tiene 12 caracteres.
```

14. En geometría un anillo o corona circular es una figura plana delimitada por dos circunferencias concéntricas. Se sabe que el área de un circunferencia es $A = \frac{\pi}{4}r^2$ y su perímetro es $\mathcal{P} = 2\pi r$, siendo r su radio. Codificar un programa que calcule el perímetro y el área de un anillo si ambos diámetros son conocidos. Seguidamente se muestra la salida. Una comprobación QAD estriba en que, si para unos valores dados y manteniendo el valor del diámetro interior se aumenta una unidad al exterior, la diferencia entra este perímetro y el original es de ≈π unidades.

```
Problema 14: cálculo del área
y del perímetro de un anillo.

Diámetro exterior: 12.5
Diámetro interior: 8.25
Área del anillo = 69.2623 u2
Perímetro = 65.1881 u
```

15. Codificar un programa que pida mediante un menú, una de las cuatro operaciones aritméticas simples (suma, resta, multiplicación o división) y las aplique a dos números cualesquiera. Hacerlo correr hasta que el usuario pida terminar, recordando que la división por cero está prohibida. Sus salidas serían algo así como:
    - Intentando dividir por cero:

```
Problema 15: simula una calculadora simple.

Calculadora
1) Suma
2) Resta
3) Multiplicación
4) División
0) Terminar
Entre una opción: 4

x = 12.5
y = 0
Error: no puedo dividir por cero.

Presione una tecla para continuar . . .
```

- Operación normal:

```
Problema 15: simula una calculadora simple.

Calculadora
1) Suma
2) Resta
3) Multiplicación
4) División
0) Terminar
Entre una opción: 4

x = 12.8
y = 3.45
División: 12.8 / 3.45 = 3.71014

Presione una tecla para continuar . . .
```

16. La Criba de Eratóstenes es un algoritmo que permite hallar todos los números primos menores que un número natural dado. Una simplificación QAD es la siguiente: se visitan todos los números comprendidos entre dos y el límite superior, y si el número visitado es primo, se cuenta y se pone en consola. Si se piden los primos hasta 120 la respuesta puede comprobarse en la (Wikipedia en español), artículo *Criba de Eratóstenes*. La salida se debiera ver más o menos así:

```
Problema 16: criba de Eratóstenes.
Muestra y cuenta los primos entre dos y un límite.

¿Número? 100

2       3       5       7       11      13      17      19      23      29
31      37      41      43      47      53      59      61      67      71
73      79      83      89      97

Son 25 primos.
```

17. Rehacer en compilación separada el ejercicio de la CPA Santos García, dado en la página 66. Con los mismos datos, salvo el encabezamiento, la salida será como en el programa 6 del segundo bloque:

```
Problema 17: cálculo del salario mensual.

-1 para terminar. Horas trabajadas: 167
Salario $2131.25

-1 para terminar. Horas trabajadas: -1
```

18. El teorema de Pitágoras es la proposición más conocida entre las que tienen nombre propio en las matemáticas y establece que: "*en todo triángulo rectángulo el cuadrado del valor de la hipotenusa es igual a la suma de los cuadrados de los valores de los catetos*", donde la hipotenusa es el lado mayor de todos.

    Codificar un programa que tome tres lados enteros y decida si con ellos se puede construir un triángulo (si la suma de los lados menores es igual al lado mayor, no se puede construir un triángulo con ellos) y si ese triángulo es rectángulo. Dos ternas pitagóricas serían: (3,4,5) —la más conocida— y (15,112,113) La salida muestra otra:

```
Problema 18: clasifica un triángulo.

Lado 1 = 17
Lado 2 = 8
Lado 3 = 15

Se puede hacer un triángulo rectángulo.
  Hipotenusa = 17
Cateto mayor = 15
Cateto menor = 8
```

19. El test comparativo o *benchmark* en inglés, es a grandes rasgos una técnica utilizada para medir el rendimiento de una computadora. Antaño se aplicaba la criba de Eratóstenes a 10,000 números, por 1,000 veces consecutivas, midiendo el tiempo que tardaba todo el proceso. Por ejemplo, en 1984 una IBM AT tardaba alrededor de 3 min y una IBM PC, unos 8 min (Microsoft Co.)

    Calcular el rendimiento de su computadora aplicándole un test similar, teniendo presente que (**a**) el tiempo depende del hardware del equipo; (**b**) como también depende del algoritmo utilizado — vpara tipificación se usará el básico, sin optimizaciones; (**c**) se está probando el hardware, por lo que las operaciones de E/S quedan fuera; y (**d**) los tiempos **`clock_t`** están representados por números enteros, pero recordar que la eficiencia es un número real menor que cero, que puede y debe ser dado en porciento. Sugerencia: para la criba ver el ejercicio 6 del segundo bloque; para la precisión decimal, ver en la página 98; para el cronometraje, ver en la página 180. Si desea puede retrasar la solución hasta llegar a dichas páginas. Y aunque la eficiencia pudiera ser diferente, su salida debiera verse más o menos así:

```
Problema 19: prueba la eficiencia de este sistema, aplicando
la criba de Eratóstenes a 10000 números, 1000 veces.

Tiempos de 1984 para una AT = 3 min; PC = 8 min.

Tenga paciencia, estoy trabajando. . .tardé 0.707567 min.

Este sistema es:
76.4144% mejor que la AT.
91.1554% mejor que la PC.
```

20. La función factorial se aplica exclusivamente a números enteros y se usa con bastante frecuencia en problemas de probabilidades y de convergencia de series. Se define como: la factorial de cero y de uno es uno y la de n es igual al producto de todos los números desde uno hasta n. En notación matemática:

    $$0! = 1;\ \ 1! = 1;\ \ \ n! = 1 \times 2 \times 3 \times \cdots \times n_i$$

    El valor de la factorial crece con suma rapidez y para números bastante discretos puede desbordar la capacidad de la PC de manejar la respuesta. Codificar una factorial que pida un entero y le calcule su valor. Tratar que acepte números de hasta 1500 (escoger un tipo adecuado de dato. Su salida debiera verse más o menos así:

```
Problema 20: cálculo de la factorial.
n es positivo.

-1 para terminar; n = 8
n! = 40320

-1 para terminar; n = 12
n! = 4.79002e+08

-1 para terminar; n = 750
n! = 2.5808e+1832

-1 para terminar; n = -1
```

    El cálculo de la factorial en Excel™ es:

n	1	8	12	750
n!	1	40320	4.79E+08	#¡NUM!

11 - TIPOS ESTRUCTURADOS DE DATOS

Se dice que C++ es un lenguaje estructurado, porque posee todos los tipos de datos básicos que tienen estructura propia. El lenguaje presenta los cinco tipos clásicos de estructuras de datos, heredados directamente de ANSI C: las enumeraciones, los arreglos, las uniones, las estructuras como tal, y los apuntadores o punteros. Además, C++ presenta como sexto tipo básico a las clases, historia para otro tomo…El Autor

Encasillamiento

Muchas veces se necesita tener un tipo de dato a partir de otro y el sistema acepta conversión automática, pero cuando hay peligro de pérdida de datos, el compilador lanza una advertencia. Hay dos tipos de conversión automática:

1. Narrowing (estrechamiento, en español). Permite al programa cambiar el tipo de un entero por el de uno de punto flotante sin perder precisión.
2. Widening (ensanchamiento, en español). En general no garantiza mantener la precisión, porque si el número de punto flotante trae decimales, al pasar a entero se trunca y si el número es tipo **`double`** o mayor, al pasar a **`float`** también se trunca.

Para el siguiente programa de ejemplo, la nota promedio es tipo **`double`**, calculada mediante la división de dos enteros, que arroja un resultado entero. Hay que convertir uno de los factores de la división a formato punto flotante, pues rara-

mente un promedio es representado por un entero. En este caso en particular puede promoverse un **`int`** a **`double`** multiplicándolo por 1.0, pero cuando se necesita un cambio seguro de tipo, se recurre al encasillamiento, que trata de forzar la operación de cambio y si se hizo correctamente, suprime la advertencia del compilador, otrosí, avisa en tiempo de compilación. Se pueden encasillar datos y punteros.

- ✓ Encasillamiento (*casting* en inglés) en C++ significa convertir un tipo de dato a otro.

Contando con la herencia del C, en C++ hay seis tipos de castings, donde los dos primeros son inseguros y su aplicación en código nuevo está totalmente desaprobada.

Tabla 21. Formas de casting o encasillamiento

Casting	Significado
`(type) expr`	Forma desaprobada, herencia del lenguaje ANSI C. Simplemente, no usar.
`type(expr)`	Forma desaprobada, pero usada a veces en C++ por ser más rápida de escribir y más clara que la anterior. Evitar en lo posible.
`static_cast<type>(expr)`	Conversiones normales y sin comprobación entre tipos o punteros compatibles.
`const_cast<type>(expr)`	Para manejar una constancia entre tipos o punteros iguales, pero que uno de ellos es constante.
`reinterpret_cast<type>(expr)`	Cambia un tipo para otro diferente y usualmente se aplica a punteros.
`dynamic_cast<type>(expr)`	Tema para el tercer tomo de esta serie, su mayor y mejor uso es aplicándole a clases polimórficas. Realiza una comprobación durante la ejecución, para asegurar la validez de la operación.

Notas:

- Los dos encasillamientos directos ANSI C/ISO C++ se pueden aplicar en situaciones que pueden y deben manejar las tres siguientes formas nuevas. Son fáciles de escribir y funcionan...a veces, pero sépase que es peligroso, pues sin quererlo podrían ser usados en una operación desastrosa, dando lugar a un error de ejecución muy difícil de encontrar.

Si se debe o se quiere aplicar el encasillamiento estilo antiguo, preferir siempre el estilo C++

- El **`dynamic_cast`** no puede ser emulada por ninguna de las anteriores, y se aplica especialmente a punteros de clases, *permitiendo encasillar una clase derivada a uno de sus ancestros*. Requiere de un compilador que tenga habilitada la información sobre tipos en tiempo de ejecución (RTTI: siglas de *Run-Time Type Information* en inglés), característica que casi siempre viene deshabilitada de fábrica. Hay que consultar el manual de su IDE y tener cuidado.

Los nuevos operadores, quizás largos y fastidiosos de escribir, deben ser puestos en código nuevo, ya que facilitan al compilador las razones exactas de la conversión, permitiéndole emitir un error si el encasillamiento convierte más allá de lo que debería, o no llega hasta donde convendría, y son mejor interpretados en una lectura humana.

Siempre usar los nuevos castings en el código nuevo

Las advertencias no son errores, pero sí que son la antesala de los mismos. EODA, siempre que se vea una advertencia, de ser posible tratar de arreglar el código para eliminarla, pero sin olvidar el Zen de Python: *lo práctico gana a lo puro.*

Salida formateada

Ud. puede controlar cómo un programa formatea la salida usando métodos de la clase ios_base y manipuladores (funciones que pueden ponerse en el flujo de inserción) definidos en las cabeceras iostream e iomanip. Estos métodos y manipuladores le permitirán controlar la base numérica, la anchura del campo de salida, el número de lugares decimales exhibidos, el sistema empleado en mostrar valores en punto flotante, y otros elementos. (Prata, C++ Primer Plus)

En muchas ocasiones se desea poner la salida en un estado especial. Por ejemplo, salidas financieras que deben poner los números alineados, columnas nítidas de datos o de nombres, tablas visualmente placenteras, etc. El sistema de E/S de C++ funciona mediante flujos.

- ✓ Un flujo es un dispositivo lógico que consume o produce información.

Un flujo se enlaza con un dispositivo físico mediante el sistema de E/S de C++, permitiendo que se pueda operar virtualmente sobre cualquier tipo de dispositivo: consola, discos duros, cintas de respaldo, etc., utilizando un mismo método para escribir o leer. Cuando comienza la ejecución de un programa en C++, se abren automáticamente cuatro flujos:

Tabla 22. Flujos implícitos

Flujos	Significado	Dispositivo implícito
cin	Entrada estándar	Teclado
cout	Salida estándar	Pantalla
cerr	Error estándar	Pantalla
clog	Búfer de **cerr**	Pantalla

Las clases con la que se trabajan normalmente derivan de **ios**, clase de E/S de alto nivel que proporciona formateado, comprobación de errores e información del estado del flujo y se usa como base para varias clases derivadas, incluyendo **istream** para entrada, **ostream** para salida, e **iostream** para E/S.

Tabla 23. (simplificada) Manipuladores de flujo

Manipulador	Propósito
boolalpha	Activa la salida booleana en inglés: **true** o **false**
nobooalpha	Desactiva la salida booleana; salen números: **0** ó **1**
showpoint	Activa la salida del punto decimal en números de punto flotante.
noshowpoint	Desactiva la salida del punto decimal en números de punto flotante.
showpos	Activa la salida del signo **±** antes del número.
noshowpos	Desactiva la salida del signo **±** antes del número
noskipws	Tiene presente los caracteres en blanco —por omisión.
skipws	Ignora los caracteres en blanco
dec	Activa la salida en formato decimal —por omisión.
oct	Activa la salida en formato octal.
hex	Activa la salida en formato hexadecimal.
endl	Da salida a una nueva línea y limpia el búfer.
fixed	Valores mostrados en notación "normal" —por omisión.
scientific	Valores mostrados en notación científica.
right	En la salida, números alineados a la derecha —por omisión.
left	En la salida, números alineados a la izquierda.
internal	El sistema escoge el formato de alineación.
endl	Saca una nueva línea y limpia el flujo.
setiosflags(flags f)	Activa las banderas de formateo especificadas en **f**
resetiosflags(flags f)	Desactiva las banderas de formateo especificadas en **f**
nouppercase	Saca los símbolos de los formatos octal y hexadecimal en mimúsculas —por omisión.
uppercase	Saca los símbolos de los formatos octal y hexadecimal en mayúsculas
Para usar los manipuladores set* hay que incluir a la biblioteca <iomanip>	
setfill(c)	Llena el ancho del campo de salida con el carácter **c**; espacio en blanco, por omisión
setprecision(d)	Fija la cantidad **d** de dígitos decimales mostrados en los números de punto flotante.
setw(e)	Fija el ancho de salida en **e** espacios.

Algunos manipuladores una vez fijados, están en acción hasta que se les desactiva; por ejemplo, **showpoint** y **noshowpoint**. A **setw** hay que llamarlo cada vez que se hace una operación de salida. Otros accionan hasta que son sustituidos por uno

similar, por ejemplo, **scientific** da la salida en formato científico hasta que es sustituido por **fixed**, que da la salida normal. Algunos de ellos también se pueden controlar con los métodos **setf** y **unsetf** de **cout**. Por ejemplo:

```
cout.setf(ios_base::scientific);   // fija el manipulador
cout.unsetf(ios_base::scientific); // libera el manipulador
```

Un consejo

A veces se hacen salidas en distintos puntos del programa y aparece un efecto colateral específico traducido en que un formateo anterior está afectando adversamente a otra salida. Para evitar esto se recomienda aplicar la técnica del ladrón: *al salir, poner todo como estaba al entrar*. Suponga que desea su salida muestre el punto decimal en los valores y tenga precisión hasta las milésimas. Entonces:

```
// formato original
ios::fmtflags oldFlags = cout.flags();
// formato deseado
cout << setiosflags(ios::fixed | ios::showpoint) << std::setprecision(3);
```

Luego, al salir del módulo, restaura los valores originales, para que no influyan en salidas subsiguientes:

```
// repone el formato original
cout.flags(oldFlags);
```

Programa auxiliar a1: manipuladores de salida

```
#include <ios>
#include <iostream>
#include <cstdlib>
#include <iomanip>
#include <cmath>

using namespace std;

int main() {
    ios::fmtflags oldFlags = cout.flags();
    cout << "Programa auxiliar 1: uso de los manipuladores.\n\n";
    cout << "Bool con n\243meros: Verdadero vale " << true << ", Falso vale " << false << '\n';
    cout.setf( ios::boolalpha );
    cout << "Bool en ingl\202s: Verdadero vale " << true << ", Falso vale " << false << "\n\n";
    cout.unsetf( ios::boolalpha );

    double d = 4.0 * atan( 1.0 );
    double e = -6.78;
    char c   = 'Q';
    int i    = 56;

    cout << "Variables: double " << d << ", " << e
         << "; char " << c << "; int " << i << "\n\n";
    cout << "Por omisi\242n los valores van a la derecha del campo:\n"
         << "|" << setw( 8 ) << d << "|\n"
         << "|" << setw( 8 ) << e << "|\n"
         << "|" << setw( 8 ) << c << "|\n"
         << "|" << setw( 8 ) << i << "|\n\n";

    cout.setf( ios::left );
    cout << "Para moverlos al inicio hay que fijar o llamar a left:\n"
         << "|" << setw( 8 ) << d << "|\n"
         << "|" << setw( 8 ) << e << "|\n"
         << "|" << setw( 8 ) << c << "|\n"
```

```
        << "|" << setw( 8 ) << i << "|\n\n";
    cout.unsetf( ios::left );

    cout.setf( ios::showpoint | ios::fixed );
    cout << "Valores con punto decimal fijo:\n";
    cout << "Entero: " << i << '\n';
    cout << "Real  : " << d << "\n\n";
    cout.unsetf( ios::showpoint | ios::fixed );

    cout.setf( ios::scientific );
    cout << "Valores en notaci\242n cient\241fica:\n";
    cout << "Entero: " << i << '\n';
    cout << "Real  : " << d << "\n\n";
    cout.unsetf( ios::scientific );

    cout << setprecision( 1 ) << fixed << showpos << right;
    cout << "Valores con signo forzado y un decimal:\n";
    cout << "Entero: " << setw( 6 ) << static_cast<double>( i ) << '\n';
    cout << "Real  : " << setw( 6 ) << d << '\n';
    cout << "Real  : " << setw( 6 ) << e << "\n\n";

    cout << "Mostrando una suma de dinero:";
    cout << setfill( '.' ) << setprecision( 2 ) << noshowpos;
    cout << setw( 12 ) << left << "\n Item 1" << right
         << setw( 6 ) << d
         << setw( 12 ) << left << "\n Item 2" << right
         << setw( 6 ) << abs( e )
         << setw( 12 ) << left << "\n Item 3" << right
         << setw( 6 ) << static_cast<double>( i )
         << setw( 12 ) << left << "\n Total"  << right
         << setw( 6 ) << ( d + abs( e ) + i )    << "$\n\n";

    cout.flags( oldFlags );
    system( "pause" );
    return EXIT_SUCCESS;
}
```

Salida

```
Programa auxiliar 1: uso de los manipuladores.

Bool con números: Verdadero vale 1, Falso vale 0
Bool en inglés: Verdadero vale true, Falso vale false

Variables: double 3.14159, -6.78; char Q; int 56

Por omisión los valores van a la derecha del campo:
| 3.14159|
|   -6.78|
|       Q|
|      56|

Para moverlos al inicio hay que fijar o llamar a left:
|3.14159 |
|-6.78   |
|Q       |
|56      |
```

```
Valores con punto decimal fijo:
Entero: 56
Real  : 3.141593

Valores en notación científica:
Entero: 56
Real  : 3.141593e+000

Valores con signo forzado y un decimal:
Entero:  +56.0
Real  :   +3.1
Real  :   -6.8

Mostrando una suma de dinero:
 Item 1......3.14
 Item 2......6.78
 Item 3.....56.00
 Total......65.92$
```

Arreglos

El arreglo o *array* en inglés, es un tipo de dato estructurado homogéneo declarado con un tamaño específico puesto dentro de corchetes. Por ej., **`int punto[20]`** declara un arreglo de 20 números enteros con el nombre común de **`punto`**. Este tipo de dato, subyacente en el diseño básico del lenguaje ANSI C, responde a las siguientes características:

- Sus elementos son todos del mismo tipo, están escritos en memoria uno a continuación del otro y son referenciados por un nombre común.
- Sus accesos por índice son muy rápidos.
- El primer índice siempre es cero, propiedad llamada indización a cero y siendo herencia del ANSI C, no es negociable.
- Usualmente no puede asignarse un arreglo completo a otro; hay que hacerlo elemento por elemento.
- El tamaño del arreglo estático se define en tiempo de compilación y no se puede cambiar en tiempo de ejecución.
  - ✓ Arreglo estático es el definido como se muestra en este acápite. Va en la pila, donde se guardan las variables automáticas y comparte espacio físico con la tienda gratis; a medida que se usa, crece hacia esa área.
  - ✓ Arreglo dinámico es el definido con el operador **`new`** en la tienda gratis; área de memoria que comparte el mismo espacio físico de la pila y crece hacia ella.
- Si se va más allá del final de cualquier arreglo se pierde la seguridad: se entra en otras áreas de la memoria que no le fueron reservadas. El arreglo se dice que está "fuera de límites." Si este es el caso, se leerá "basura" o se escribirá fuera del arreglo, acabando con lo ya estaba ahí, lo cual es un error en tiempo de ejecución que puede ir desde "nada pasó…todavía" hasta "¡la catástrofe!". Y mucho peor, el error puede salir a flote en algún momento, durante alguna de las ejecuciones continuadas del programa, delante de quien no debería estar presente, etc.

Es responsabilidad total del codificador que el error fuera de límites no ocurra

Recomendación: cuando se vaya a pasar un arreglo como parámetro, pasarle también su tamaño, para que el módulo que lo reciba sea autocontenido y no tenga que calcularlo o ir a buscarlo a otros lugares, aunque sin olvidar el Zen de Python: *lo práctico gana a lo puro*.

Debe tenerse presente que el nombre de un arreglo ya trae consigo la dirección de su primer elemento y anteponerle el operador de referencia es un error de sintaxis. Si sus valores no deben ser alterados, entonces hacer su pase como parámetro constante.

Arreglo numérico simple

La potencia del arreglo se ve muy bien durante los ciclos. Si se piensa por un momento en cómo entrar 100 números manualmente por consola, cómo sacarlos manualmente a consola, o como visitarlos uno por uno, inmediatamente se comprende que serían tareas sumamente engorrosas, grandes consumidoras de tiempo y espacio, y con altas probabilidades de salir erróneas en los primeros intentos.

Declaración

El número entre corchetes establece la cantidad de elementos en el arreglo y por ahora tiene que ser constante, porque esta estructura se guarda en una parte que está fuera de la memoria dinámica, y su tamaño debe ser determinado en tiempos de compilación. Un arreglo estático puede ser declarado de dos formas, como:

```
1.  int nota[5];         // un literal o...
2.  const int NOTAS = 5; // ...una constante
    int nota[NOTAS];     // ...entre corchetes
```

Inicialización

Si un arreglo es relativamente corto, puede ser inicializado en su declaración de cualquiera de las dos siguientes maneras:

```
1.  int nota[5] = { 63, 97, 88, 77, 53 };
2.  int nota[]  = { 63, 97, 88, 77, 93 };
```

De la última forma sería el compilador quien contaría el tamaño (opción preferida.) Actualmente, según C++11 también podría haberse inicializado omitiendo el operador de asignación:

```
1.  int nota[5] { 63, 97, 88, 77, 93 };
2.  int nota[]  { 63, 97, 88, 77, 93 };
```

A veces un arreglo grande requiere de comenzar con todos sus elementos puestos a cero y se codifica así:

```
int nota[100] { 0 };
```

pero si se hubiera codificado así:

```
int nota[100] = { 63, 97, 88, 77, 53 };
```

los cinco primeros elementos tomarían esos valores y los 95 restantes tomarían el valor cero.

Acceso a sus elementos

Los elementos de un arreglo llevan entre corchetes un índice (valor entero y positivo, ya sea un literal, una constante simbólica o el producto de un cálculo), que permite operar con uno de sus valores individuales. Del corchete se dice que es un operador por compensación, o también por acceso directo. En el ejemplo antes dado y como muchos elementos del lenguaje, el corchete tiene dos usos diferentes con el arreglo: en la declaración especifica su tamaño y como operador, define dónde está uno cualquiera de sus valores. No se deben confundir:

```
int nota[5] { 63, 97, 88, 77, 93 }; // indica un tamaño de cinco elementos
cout << nota[1];                    // imprime el número 97
```

El corchete en la segunda forma es un operador binario de indirección, que recibe el nombre del arreglo a su izquierda y el índice en su interior. También es llamado operador de compensación (*offset operator* en inglés) puesto que su valor puede ser dado por una operación cualquiera que entregue un número entero positivo, "compensando" así la dirección. Devuelve el valor del elemento del arreglo que está en ese lugar.

Programa 15: una tabla de notas

Se pide calcular y mostrar la nota promedio (de cero a 100 puntos) de un alumno que cursa 5 asignaturas. La salida deberá ser agradable a la vista. Sean $\bar{x}$ el promedio, x_i el número iésimo de un conjunto de valores y n su potencia (cantidad de valores del conjunto.) El promedio se define como la suma de todos los valores x_i del conjunto, desde $i = 1$ hasta n dividida por su potencia:

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n} = \frac{\sum x_i}{n}$$

Datos del problema

	Matemáticas	Español	Física	Química	Inglés
Alumno	63	97	88	77	93

Declaraciones

```
#ifndef PROMEDIO_H_INCLUDED
#define PROMEDIO_H_INCLUDED

const int NOTAS = 5; // columnas

//  pone una línea doble o simple
//  input: si es simple o doble
//  output: una línea simple o doble por omisión
void ponerLinea(bool doble = true);

//  muestra la tabla y su promedio
//  input: el arreglo
//  output: la tabla en consola
void mostrar(const int ary[], int n);

#endif // PROMEDIO_H_INCLUDED
```

Definiciones

```
#include "promedio.h"
#include <iostream>
#include <iomanip>
using std::cout;

//  pone una línea doble o simple
void ponerLinea(bool doble) {
  ( true == doble )
    ? cout << std::setfill( '=' )
    : cout << std::setfill( '-' );
  cout << std::setw(8 * (NOTAS + 2)) << '\n' << std::setfill(' ');
}

//  muestra las notas
void mostrar(const int ary[], int n) {
  cout << std::setprecision(1) << std::fixed << "Tabla de resultados\n";
  ponerLinea();
```

```
  cout << "Asig.\tMat.\tEsp.\tF\241s.\tQu\241.\tIng.\tProm.\n";
  ponerLinea(false);

  double total = 0;
  cout << "Alum.\t" ;

  for ( int i = 0; i < n; ++i ) {
    cout << ary[i] << '\t';
    total += ary[i];
  } // for i

  cout << total / n << '\n'; // su promedio
  ponerLinea();
}
```

Programa

Apreciar como todo el trabajo lo hace la función mostrar. ¿Eso es correcto? Prácticamente sí más teóricamente no, pues viola el espíritu de la programación moderna y las directivas para una función bien formada— si algún día hubiera que dar mantenimiento, el diseño tomado lo dificultaría. Más adelante se hará una revisión sobre este tema. Ahora, lo importante es observar cómo se declara el arreglo en la lista de parámetros tanto del prototipo como de la definición, cómo se pasa el parámetro a la función dentro del cuerpo de **main** y notar la diferencia.

```
#include "promedio.h"
#include <iostream>
#include <cstdlib>

int main() {

  std::cout << //  información
  "Programa 15: muestra las " << NOTAS << " notas de " <<
  "un alumno y calcula su promedio.\n\n";

  int notas[] { 63, 97, 88, 77, 93 };//  vector de notas
  int n = sizeof(notas) / sizeof(notas[0]); // cantidad de elementos

  //  la tabla
  mostrar(notas, n);

  //  fin
  system("pause");
  return EXIT_SUCCESS;
}
```

Salida

```
Programa 15: muestra las 5 notas de un alumno y calcula su promedio.

Tabla de resultados
=========================================================
Asig.    Mat.     Esp.     Fís.     Quí.     Ing.     Prom.
---------------------------------------------------------
Alum.    63       97       88       77       93       83.6
=========================================================
```

Arreglo bidimensional

Recordar que por definición los elementos de un arreglo son puestos en memoria consecutivamente. Entonces, un arreglo bidimensional (arreglo 2D) es uno simple de cualquier tamaño, donde cada uno de sus elementos es otro arreglo simple que necesariamente es de tamaño fijo, para que sus valores puedan entrar uno a continuación del otro, como se muestra en la figura.

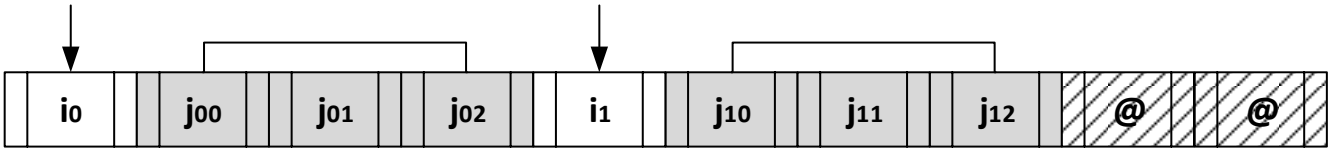


Ilustración 7. Arreglo bidimensional de 2 x 3 en memoria

Se ilustra un arreglo de 2 filas, de un arreglo fijo de 3 columnas o a_{ij}. Claramente se ve que luego de entrar el primer elemento (i_0) hay que poner los elementos fijos de su segunda dimensión (j_{00}, j_{01}, j_{02}). Luego entra el segundo elemento (i_1) y siguen los elementos fijos de la segunda dimensión (j_{10}, j_{11}, j_{12}) Quedan elementos espurios que estaban allí antes de la operación.

Una función que reciba un arreglo 2D como parámetro puede trabajar con arreglos de cualquier cantidad de filas, ¡pero atención!, la cantidad de columnas será siempre la misma, porque son fijas. Eso es lo que C++ permite para estos casos.

Determinación de las dimensiones de un arreglo

Las dimensiones de un arreglo son hasta cierto punto construcciones de conveniencia, porque los valores van escritos a continuación uno del otro, pero la reserva de memoria responde a la declaración. Ellas están dadas por la cantidad de índices para los que se reservó memoria. Por ejemplo, **`float miVector[5]`** (sólo se declaró un índice) es un arreglo lineal de 5 valores reales, pero **`int miTabla[5][25]`** (dos índices reservados) es un arreglo bidimensional de 5 filas, o registros, o *records* en inglés y 25 columnas, o campos, o *fields* en inglés, el cual se dice es un arreglo de 5×25, y así sucesivamente.

Cuando sólo se trabaja con una dimensión se suele hablar de arreglos lineales, listas, o vectores. Cuando se usan dos la referencia es a arreglos 2D, tablas, o vectores bidimensionales. Cuando se usan tres se dicen arreglos 3D, cubos, o vectores tridimensionales. Más dimensiones son raramente utilizadas en la programación típica, aunque la norma permite hasta doce de ellas.

Lazos anidados

A los elementos de arreglos de múltiples dimensiones usualmente se les accede mediante lazos anidados, o sea, contenidos totalmente uno dentro de los otros. C++ no permite un contenido parcial, por lo que el compilador automáticamente cerrará el lazo con las llaves más cercanas entre sí.

Usualmente no es lo buscado por el programador y el error puede ir desde una salida caótica a consola hasta un error de permisividad (error de compilación.) Por ejemplo:

```
for (int i = 0; i < ESTUDS; ++i) {    // i pertenece al lazo exterior
  for (int j = 0; j < NOTAS; ++j) {  // j pertenece al lazo interior
    std::cout << notas[i][j] << '\t';
  } // for j
  nota += notas[i][j];          // aquí j cruza el lazo al cual pertenece
  std::cout << static_cast<double>(nota)/EXAMS << '\n';
} //for i
```

y el compilador emite un error de permisividad:

```
…notas.cpp:24 name lookup of ‘j’ changed for ISO ‘for’ scoping [-fpermissive] (if you use ‘-fpermis-
sive’ G++ will accept your code)
```

que en español dice algo así como: *el ámbito de j cambió respecto a la norma ISO del ámbito para el for, si usa el parámetro –fpermissive, [el compilador] G++ aceptará su código.*

Eso no es lo que se desea; en realidad, lo correcto sería:

```
for (int i = 0; i < ESTUDS; ++i) {
    for (int j = 0; j < NOTAS; ++j) {
        std::cout << notas[i][j] << ‘\t’;  // j está dentro del ámbito de i
        nota += notas[i][j];               // y aquí ambos son visibles
    } // for j
    std::cout << static_cast<double>(nota)/NOTAS << ‘\n’;
} //for i
```

Inicialización de arreglos bidimensionales

Al crear tablas cortas, se les puede inicializar cada uno de sus elementos, ampliando el modo de hacerlo a un vector lineal. En un arreglo 2D cada elemento puede verse como otro arreglo de tamaño fijo, por tanto, se inicializa cada fila separada de la otra con una coma (los paréntesis interiores son ignorados por el compilador, pero ayudan a la lectura humana del código); las columnas de cada fila se inicializan con sus valores entre llaves, separados por coma entre ellos, como muestra. Aun cuando el compilador acepta en su declaración **`int notas[3][6]`**, se prefiere como se muestra, dejando libre la primera dimensión, la cual es contada automáticamente por el compilador.

```
//  notas: arreglo de 3 x 6 - (C = columna; F = fila)
int notas [][6] = {
//   C1  C2  C3  C4  C5  C6
    {63, 97, 88, 77, 93, 95}, // F1
    {96, 87, 89, 78, 80, 89}, // F2
    {70, 90, 86, 81, 90, 82}  // F3
}; // cerrar con punto y coma
```

La segunda dimensión y las superiores tienen que dar su tamaño explícitamente, ya que sus elementos ocupan una posición fija en memoria. Como se muestra arriba, cada fila entra secuencialmente a sus seis columnas. Si se modifica la declaración así:

```
//  notas ⬇
int notas [3][] = { ... };
```

o así:

```
//  notas ⬇⬇
int notas [][] = { ... };
```

se obtiene un error de compilación:

```
...declarations of ‘notas’ as multidimensional array must have bounds for all dimensions except the first.
```

Que en español se lee como: *la declaración de ‘notas’ como arreglo multidimensional deberá tener límites para todas las dimensiones, excepto la primera.*

Programa 16: otra tabla de notas

Se pide mostrar una tabla con las notas (de cero a 100 puntos) y sus promedios, de 3 alumnos que cursan 6 asignaturas. Las puntuaciones son dadas en la tabla que sigue. En la cabecera se declara constante la segunda dimensión, requisito para el manejo correcto del arreglo estático multidimensional. La salida deberá ser agradable a la vista.

Datos del problema

	Matemáticas	Español	Física	Química	Inglés	Biología
Alumno 1	63	97	88	77	93	95
Alumno 2	96	87	89	78	80	89
Alumno 3	70	90	86	81	90	82

El atento lector inmediatamente advierte que este programa es una ampliación del anterior. Cuando a un problema ya resuelto se le pide una ampliación de sus prestaciones originales, se está en presencia de un fenómeno llamado escalada o escalamiento.

- ✓ Escalamiento en este sentido aparece a partir de un programa hecho y funcional, al que se añaden nuevas complejidades, producto de nuevas demandas.

Cuando son pocas las alteraciones, basta con integrarlas a lo ya hecho, pero llega muy, muy pronto el momento en que aparecen advertencias del compilador, efectos colaterales —molestos cuando menos, y bugs de todo tipo. Entonces se dice que el programa no escaló bien y hay que readaptarlo o sustituirlo por una versión avanzada.

- ✓ Readaptación o rehúso en este sentido es usar el programa de partida, pero añadiéndole los elementos deseados, manteniendo lo más posible la forma original.

Al readaptar para un problema dado, las interfaces de los módulos usualmente crecen, dando una idea sobre el fuerte acoplamiento de los mismos con el programa. A mayor fortaleza de esa ligadura, menor reusabilidad, portabilidad y facilidad de mantenimiento, por eso, siempre que se pueda se prefiere programar para un entorno abstracto, lo que facilita la readaptación y hace las interfaces lo más breves posibles, lo cual se verá con más detalle en el tomo siguiente.

Declaraciones

```
#ifndef PROMEDIO_H_INCLUDED
#define PROMEDIO_H_INCLUDED

const int NOTAS = 5; // columnas

//  pone una línea doble o simple
//  input: si es simple o doble
//  output: una línea simple o doble por omisión
void ponerLinea(bool doble = true);

//  muestra la tabla y su promedio
//  input: el arreglo
//  output: la tabla en consola
void mostrar(const int ary[], int n);

#endif // PROMEDIO_H_INCLUDED
```

Definiciones

```
#include "promedio.h"
#include <iostream>
#include <iomanip>
using std::cout;

//  pone una línea doble o simple
void ponerLinea(bool doble) {
  ( true == doble )
    ? cout << std::setfill( '=' )
    : cout << std::setfill( '-' );
  cout << std::setw(8 * (NOTAS + 2)) << '\n' << std::setfill(' ');
}
```

```
//  muestra las notas
void mostrar(const int ary[], int n) {
  cout << std::setprecision(1) << std::fixed << "Tabla de resultados\n";
  ponerLinea();
  cout << "Asig.\tMat.\tEsp.\tF\241s.\tQu\241.\tIng.\tProm.\n";
  ponerLinea(false);

  double total = 0;
  cout << "Alum.\t" ;
  for ( int i = 0; i < n; ++i ) {
    cout << ary[i] << '\t';
    total += ary[i];
  } // for i

  cout << total / n << '\n'; // su promedio
  ponerLinea();
}
```

Programa

De nuevo todo el trabajo lo hace la función mostrar y ya se sabe que no es teóricamente correcto, pero *lo práctico gana a lo puro*. Más adelante también se hará una revisión sobre este tema. Ahora, lo importante es estudiar bien la adaptación, observar cómo se declara el arreglo en la lista de parámetros tanto del prototipo como de la definición, cómo se pasa el parámetro a la función dentro del cuerpo de **main** y notar la diferencia.

```
#include "promedio.h"
#include <iostream>
#include <cstdlib>

int main() {

  std::cout << //  información
  "Programa 16: muestra las " << NOTAS << " notas de " <<
  "un alumno y calcula su promedio.\n\n";

  int notas[] { 63, 97, 88, 77, 93 };//  vector de notas
  int n = sizeof(notas) / sizeof(notas[0]); // cantidad de elementos

  //  la tabla
  mostrar(notas, n);

  //  fin
  system("pause");
  return EXIT_SUCCESS;
}
```

Comprobar con cuidado ambos programas y analizar las partes readaptadas o añadidas. Tratar de comprender la anidación de ciclos en el cuerpo de la tabla y el ciclo de cómo se obtuvieron los valores de los promedios por nota.

Salida

```
Programa 16: muestra la tabla de 6 notas de 3 alumnos y calcula sus promedios.

Tabla de resultados.
================================================================
Asig.   Mat.    Esp.    Fís.    Quí.    Ing.    Bio.    Prom.
----------------------------------------------------------------
Al. 1   63      97      88      77      93      95      85.5
Al. 2   96      87      89      78      80      89      86.5
Al. 3   70      90      86      81      90      82      83.2
----------------------------------------------------------------
Prom.   76.3    91.3    87.7    78.7    87.7    88.7
================================================================
```

Enumeradores

El enumerador (**enum**) es un tipo estructurado de dato, opcionalmente declarado con un nombre y un conjunto de identificadores mapeados a enteros en forma semiautomática.

```
enum Logico { no, quizas, si };
Logico sePermite = quizas;
```

- **enum** puede ser anónimo, o pueden declararse variables **enum**, como en el ejemplo anterior. En cualquier caso, los nombres se representan como valores enteros, ya que el compilador enumera automáticamente cualquier lista de identificadores que se le pase, siempre asignándoles los valores 0, 1, 2, etc.
- Si el modo automático no es adecuado, se puede mapearlos manualmente. Por ejemplo:

  ```
  enum Figura { circulo = 10, cuadrado = 20, rectangulo = 50 };
  ```

- Si se dan valores a algunos nombres y a otros no, para los no mapeados el compilador utilizará el valor entero que sigue al último que se mapeó. Por ejemplo, si declara:

  ```
  enum { tin = 25, marin, cucara = 35, macara }
  ```

- el compilador le da a **marín** el valor 26 y a **mácara** el valor 36.

Independientemente de las diversas situaciones que se le presentan al programador, al solucionar un problema, existen dos de ellas donde el dato enumerado es útil:

1. Con valores secuenciales puede usarse para gobernar los ciclos sobre arreglos, sin necesidad de saber o utilizar explícitamente sus tamaños.
2. Cuando se quiere seguir la pista de alguna característica en un programa. El código es más legible cuando se utilizan tipos de datos enumerados, aclarando a quien lo lea sobre ciertos significados usados con la instrucción **switch**. Este último caso se presenta en el texto, siendo adaptado a los programas manejados por menús.

Programa 17: enumeradores

Los procesadores modernos de texto usan acciones (cambio de mayúsculas, minúsculas, comparaciones insensibles al tamaño de la letra, etc.) Éstos presentan al usuario una opción que consiste en cambiar el tamaño de las letras de una frase, o de un párrafo. Generalmente esas opciones son (entre otras) poner todas las letras en mayúsculas, o en minúsculas, o como una oración –esto último no es para nada trivial, pero acá sólo se considerará poner la primera letra en mayúsculas. Codificar un programa que mediante un menú emule las opciones descritas.

Declaraciones

- El menú:

```
#ifndef MENU_H_INCLUDED
#define MENU_H_INCLUDED

#include <string>
typedef std::string Str;

//  enumera lo que pide el menú
enum { terminar, mayusculas, minusculas, oracion };

//  un menú en consola
//  entrada: la información al usuario
//  salida: menú a consola
short menu(Str info);

#endif // MENU_H_INCLUDED
```

- Los cambios:

```
#ifndef CAMBIO_H_INCLUDED
#define CAMBIO_H_INCLUDED

#include <string>
typedef std::string Str;

//  pone la frase en mayúsculas
//  input: una cadena C++
//  output: la cadena a mayúsculas
void aMayusculas(Str &frase);

//  pone la frase en minúsculas
//  input: una cadena C++
//  output: la cadena a minúsculas
void aMinusculas(Str &frase);

//  pone la frase como una oración
//  input: una cadena C++
//  output: la cadena a oración
void aOracion(Str &frase);

#endif // CAMBIO_H_INCLUDED
```

Definiciones

- El menú:

```
#include "menu.h"
#include <iostream>
#include <cstdlib>

//  un menú en consola
short menu(std::string info) {
  Str conjunto = "0123";
  Str opcion;
  Str::size_type idx;

  while ( true ) {
```

```
      system("cls");
      std::cout << info <<
      "1) May\243sculas, 2) Min\243sculas, 3) Oraci\242n\n"
      "0) Terminar. Entre una opci\242n: ";
      ( std::cin >> opcion ).get();
      idx = conjunto.find(opcion);

      if ( Str::npos == idx ) {
        std::cerr << "\nError: opci\242n ilegal.\n\n";
        system("pause");
      } else {
        return idx;
      } // if-else
    } // while
  }
```

- El cambio de tamaño:

```
#include "cambio.h"

//  pone la frase en mayúsculas
void aMayusculas( Str &frase ) {
  for ( unsigned i = 0; i < frase.length(); ++i ) {
    frase[i] = toupper( frase[i] );
  } // for
}

//  pone la frase en minúsculas
void aMinusculas ( Str &frase ) {
  for ( unsigned i = 0; i < frase.length(); ++i ) {
    frase[i] = tolower( frase[i] );
  } // for
}

//  pone la frase en oración
void aOracion( Str &frase ) {
  aMinusculas( frase );
  toupper( frase[0] );
}
```

Programa

```
#include "menu.h"
#include "cambio.h"
#include <iostream>
#include <cstdlib>

const std::string PROMPT( "\n   Entre una frase: " );

int main() {
  std::string info( //  información
    "Programa 17: un men\243 para el cambio de tama\244o de letras.\n"
    "No trabaja bien con el espa\244ol.\n\n" );

  //  ciclo
  std::string frase;
  unsigned opcion;

  while ( true ) {
    opcion = menu(info);
```

```
    if ( terminar == opcion ) {
      return EXIT_SUCCESS;
    } //  if

    std::cout << PROMPT;
    std::getline(std::cin, frase);

    switch ( opcion ) {
    case mayusculas:
      aMayusculas(frase);
      std::cout << "Frase a may\243sculas: " << frase << "\n\n";
      break;
    case minusculas:
      aMinusculas(frase);
      std::cout << "Frase a min\243sculas: " << frase << "\n\n";
      break;
    case oracion:
      aOracion(frase);
      std::cout << "   Frase a oraci\242n: " << frase << "\n\n";
      break;
    } // switch

    system("pause");
  } //  while
}
```

Salida

```
Programa 17: un menú para el cambio de tamaño de letras.
No trabaja bien con el español.

1) Mayúsculas, 2) Minúsculas, 3) Oración
0) Terminar. Entre una opción: 1

   Entre una frase: una tarde de mayo.
Frase a mayúsculas: UNA TARDE DE MAYO.

Presione una tecla para continuar . . .
```

Uniones

En un lenguaje orientado a objetos, es importante preservar la encapsulación. Así, la capacidad de las uniones para enlazar código y datos permite crear tipos de clases en los que todos los datos usan una posición compartida. Esto es algo que no se puede hacer usando una clase como tal. (Schildt, C++. Guía de Autoenseñanza, pág. 58)

En C++ la unión (**`union`** en inglés) es una clase compuesta por un conjunto de valores heterogéneos, que pueden ser cada uno de ellos de cualquier tipo de dato, pero que están almacenados todos en el mismo espacio de memoria. En un momento dado de la ejecución, sólo uno de esos valores puede estar activo, otorgándole una altísima capacidad de encapsulación. Sus miembros son públicos por defecto, significando que cualquier instrucción del programa puede acceder a cualquiera de sus miembros o métodos.

Su variante POD (sin funciones-miembro) fue concebida en ANSI C para actuar sobre las operaciones directas con bits y para manejar óptimamente recursos, por la gran compactación de su almacenamiento y su rapidez de ejecución, pero como ahora no faltan la rapidez de la CPU, la capacidad de almacenaje externo, o el tamaño de la memoria de trabajo, EODA la unión ya no es tan globalmente importante. Como su manejo correcto no es elemental, no será vista en la serie.

Dicho esto, existen programas C++ escritos para estar incrustados en un sistema donde el ahorro de memoria y tamaño es primordial, que pueden hacer buen uso de las uniones. Desde hornos microondas, pasando por aviónica civil, militar o espacial, hasta sistemas gigantes de climatización, están presentes dondequiera en este mundo moderno. También se emplean en la creación de datos estructurados para interactuar directamente con el hardware.

Estructuras C++

Aunque las estructuras tienen las mismas capacidades que las clases, muchos programadores restringen su uso para ceñirse a su forma de C y no las usan para incluir funciones-miembro. (Schildt, C++. Guía de Autoenseñanza, pág. 58)

En C++ la estructura *es verdaderamente una clase*[27] que abarca **(a)** un conjunto de valores heterogéneos llamados miembros, que pueden ser de cualquier tipo, aunque generalmente los componentes están relacionados de alguna suerte entre sí, formando un todo coherente, y **(b)** las funciones que trabajan sobre dichos miembros y son llamadas métodos.

- ✓ Miembro es todo dato que es parte de una unión, estructura o clase.
- ✓ Método es una función que es parte de una unión, estructura o clase, se declara dentro de ellas y trabaja en estrecho vínculo con sus respectivos miembros. El conjunto de métodos constituye la interfaz de la estructura.

```
struct Punto {
  // métodos
  int getX();
  int getY();
  void putX(int val);
  void putY(int val);
  // miembros
  int x, y;
};
```

Hay tres grados de acceso a los componentes de una estructura o clase: **`public`**, **`protected`** y **`private`**. En las estructuras, los miembros son públicos (**`public`**) por defecto, significando que cualquier instrucción del programa puede acceder a cualquiera de sus miembros o métodos, haciéndolas vulnerables, desprotegidas y fuentes de errores lógicos difíciles de encontrar y erradicar. En C++ esto se remedia usando las clases, cuyos miembros son privados (**`private`**) por defecto, lo cual se deja para la tercera y última parte de la serie. Ahora sólo se codificará con cuidado.

Una estructura **(a)** no puede contener un miembro de su mismo tipo; **(b)** pero puede contener un puntero a su mismo tipo, permitiendo la creación de datos autoreferenciados complejos tales como listas enlazadas, árboles y grafos; y **(c)** puede contener a otras estructuras, lo cual se llama composición por agregación, permitiendo la creación de estructuras complejas.

Puesto que en C++ una estructura es un tipo básico tratado como si fuera uno nativo (**`int`**, **`float`**, etc.), una vez definida su organización (identificación de sus miembros y métodos), ya se pueden crear directamente variables (objetos) para uso del programa. Además, el lector debe conocer que:

- Dos estructuras iguales permiten asignación directa mediante un pase de datos “miembro-a-miembro”, aun cuando alguno de ellos sea un arreglo, que comúnmente no puede pasarse así.
- Los operadores binarios de membresía —punto y flecha— se usan para acceder a sus miembros.

Tabla 24. Operadores de membresía

Op.	Operador	Aplicación	Se usa con
.	Punto	Sobre el objeto como tal	Miembros de estructuras, uniones y clases

[27] En C++ las estructuras son tratadas por la programación en súper C como clases que, aunque desprotegidas, son más simples de codificar, dejando la programación OOP tradicional para las “verdaderas” clases.

`->`	Flecha	Sobre un puntero al objeto	Punteros a miembros de estructuras, uniones y clases.

En C++, las estructuras tipo POD son totalmente compatibles con las del C y se emplean sobre todo para operaciones de mantenimiento retroactivo sobre código antiguo. Sin ahondar en el tópico por ahora, el texto aplicará la potencialidad de las estructuras actuando como clases muy simplificadas para manejar con soltura las complejidades de los ejemplos, y se recomienda al lector lo haga si se da el caso con los problemas propuestos que siguen, allanando el paso de la programación imperativa de esta primera parte, a la parte de la OOP que culmina esta serie.

Aquí es de observar **(a)** la ligadura de datos (miembros) y funciones (métodos), **(b)** la extrema fortaleza y tersura de la interfaz; y **(c)** la nueva sintaxis introducida, especialmente el uso del operador de ámbito *dobles-dos-puntos* (**`::`**)

Quizás no sea así en casos sencillos como los programas y ejercicios que se presentan en el texto, pero se advierte que normalmente el código OOP lleva más tiempo de concebir que el tradicional, es más largo de escribir y usualmente hay que definirlo en el ámbito del problema. Una vez hecho esto, es más simple y claro de programar y usar, aporta una mejor seguridad al manejo de las complejidades de la solución al problema y eleva el grado de mantenibilidad.

La introducción del diseño orientado a objetos OOD (*Object Oriented Design*, en inglés) y la programación OOP, han permitido extender el universo de respuestas a problemas que en su momento ya bordeaban la imposibilidad de ser solucionados por los medios digitales de antaño. Por eso se considera la orientación a objetos como un verdadero paradigma en este campo.

La variante POD

Puesto que class es una entidad sintácticamente separada de struct, la definición…es libre de evolucionar de una forma que puede no ser compatible con una definición…al estilo de C. Puesto que…son distintas, la futura dirección de C++ no está restringida por términos de compatibilidad. (Schildt, C++. Guía de Autoenseñanza, pág. 58)

En general, la variante POD de C++ es la estructura de ANSI C que sólo lleva miembros, ya que el lenguaje no “conoce” nada acerca de las clases y sus métodos. Como tipo estructurado es muy apropiada para crear datos complejos, ajustados para una aplicación en particular. Es común en C++ denominar a las estructuras POD como registros y sus miembros como campos. Un arreglo de este tipo de estructuras es el ABC de las bases de datos relacionales clásicas.

Restringir la potencia de la estructura es un asunto casi de preferencias. Es decir, el programador veterano de C++ se siente más seguro con las clases y prefiere trabajar directamente con ellas, el acceso de cuyos miembros es privado por omisión, permitiendo un encapsulamiento seguro, una protección más fehaciente y una terminología más ajustada a los conceptos de la OOP, dejando el uso de las estructuras del tipo ANSI C solamente para realizar compatibilidad retroactiva, por razones de mantenimiento.

Incluso cuando hasta hoy (11/12/2020) no han sido marcadas como desaprobadas, nada ni nadie garantiza que las estructuras C++ se queden en el estado actual de desarrollo y la línea de implantación de las clases se les separe totalmente.

¿Por qué uniones, estructuras y clases?

En C++, el tipo estructurado struct es redundante: es heredado del C. Pero conservando la palabra-clave struct sus programas pueden asimilar bibliotecas de C que la emplean. No obstante, cualquier programa C++ que espera usarla debe restringir sus definiciones de modo que no tengan funciones-miembro, constructores, partes privadas, etc. (Halterman, pág. 465)

En C++ una unión, estructura o clase es un tipo básico con características similares. Entonces, ¿por qué existen uniones y estructuras con los súper poderes de las clases? Una explicación dada por (Schildt, C++. Guía de Autoenseñanza, pág. 58) y compartida por este autor es que siempre se quiso mantener compatibilidad retroactiva, dada la cantidad de código ANSI C existente, material con potencialidad para pasar a ser rehecho en C++, y así ha sido.

Estructuras dentro de estructuras

Ya se vio que la estructura es una clase compuesta por un conjunto de valores heterogéneos de cualquier tipo, que no pueden ser de su mismo tipo, pero si pueden ser punteros a ellas o a otras estructuras. Poner una estructura POD dentro de otra es común en ANSI C, pero en algunos entornos de desarrollo el compilador C++ emite varias advertencias, porque las "ve" como clases y espera constructores, métodos, etc. Si es el caso, se pueden ignorar.

Constructores, destructores y el puntero *this

Hay tres elementos —que serán ahondados adecuadamente en la tercera y última parte de esta serie— a los que es necesario ahora mismo hacerles un breve examen: el constructor, el destructor y el puntero **`*this`**.

1. Constructor. Ahora mismo puede verse como una forma más de declarar la variable de una estructura.

Una estructura C++ posee un constructor que lleva su mismo nombre y no puede retornar valores, porque su única misión es inicializar los miembros con datos apropiados a la tarea, pero si no se usa la tienda gratis, casi nunca hace falta explicitarlo: C++ lo pone por omisión. No obstante, como en este caso, si el constructor lleva parámetros usados para pasarle valores a los miembros de la estructura en el instante de ser creados, hay que codificarlo y casi siempre se hace dándole valores por omisión que le permitan hacer una tarea dual: actuar como constructor por omisión y actuar como constructor con parámetros.

El constructor puede ser sobrecargado tal y como se hace con una función normal

2. Destructor. Ahora mismo puede verse como una función de limpieza.

Una estructura C++ posee un destructor que lleva su mismo nombre con una tilde antepuesta (**~**) y no puede tomar parámetros o retornar valores, porque su única misión es destruir un objeto, pero si no se usa la tienda gratis, no hace falta explicitarlo: C++ lo pone por omisión.

El destructor no puede ser sobrecargado

3. **`*this`**. Este puntero está presente, aunque oculto en cada estructura y se usa para conseguir el valor del elemento al cual se aplica. Aunque se abordará apropiadamente en la tercera parte, hay una somera explicación de su función en la página 130, en el tópico dedicado a los punteros.

Programa 18: estructura C++

En la cafetería El Jardín se ofertan a la población tres productos cuyos precios son los siguientes: refresco a $1.00, dulce de rollito a $1.20, y bola de helado a $0.55. Al informático ubicado en ese establecimiento se le ha dado la tarea de elaborar un programa que, mediante un menú, permita mostrar las ofertas (nombre y precio de cada producto), pueda entrar la venta del día y determinar la recaudación total al final de cada jornada.

He aquí un problema simple, pero a la vez complejo, porque es abarcador de muchos de los tópicos vistos hasta el momento. Presenta algo de análisis utilizable por el lector novel, aunque el OOD es tópico para libros de texto independientes del lenguaje. A estas alturas del curso y para no tornarlo más denso de lo debido, no se controlan mucho los errores en las entradas.

El análisis y el diseño en general

Aunque es grandioso levantarse una mañana y decidir crear un juego clásico de aventuras, nunca será práctico si no existe un buen plan. Pequeños módulos pueden crearse sin mucho planeamiento, pero entre más larga y compleja la tarea, más se requiere de una buena planificación a seguir, lo cual de paso ayuda a erradicar muchos errores. (Linux Tricks and Tips, pág. 64)

Se advierte que al comenzar un proyecto serio en la vida real la cuestión no estriba en enrollarse las mangas, flexionarse los dedos y comenzar a teclear. En las grandes empresas de desarrollo de software, siempre se aplica previamente el análisis y diseño orientado a objetos a la creación de nuevos productos y en aquellas que no utilizan para su creación esos conceptos, se emplean otros modelos, pero ninguna aplicación seria se gesta sin un análisis y diseño previo, que normalmente consume bastante tiempo del fijado según plan para toda la tarea.

El enfoque OOD organiza la solución como un sistema de objetos en interacción. Con este método se crea un conjunto de vistas de símbolos especializados utilizando una notación prestablecida que actúa como un sistema de planos de construcción al que se atendrán todos los desarrolladores. El lenguaje unificado de modelado o ULM (*unified modeling language* en inglés) es la notación prevaleciente hoy día. El curioso lector deberá buscar apoyo bibliográfico en textos como *The Unified Modeling Language User Guide* y *The Unified Modeling Language Reference Manual*, ambos de Booch, Rumbaugh & Jacobson (el trio creador del UML) y publicados por la editorial Addison-Wesley, USA.

Esos tópicos rebasan ampliamente los objetivos del texto introductorio, no obstante, en casos tan simples como éste, se puede hacer un diseño OOD muy QAD con lápiz y papel, que sirva de guía de desarrollo.

Análisis y diseño del problema

Las relaciones entre clases básicamente son de dos tipos:

1. La relación *tiene-un-o-unos/tiene-una-o-unas*, indica que a una estructura se le pueden agregar varias instancias de otra u otras. Se dice que este tipo de relación es modelada por la composición por agregación.
2. La relación *es-un/es-una* indica la formación básica de la estructura y de sus descendientes, e involucra la herencia, tópico que se verá en el tercer tomo de esta serie.

Si se analiza el entorno más global del problema inmediatamente se ve que hay dos estructuras involucradas: la entidad "El Jardín" *es-una* cafetería (estructura básica) que *tiene-unos* productos (estructuras agregadas.) El producto está dado en una estructura, pero instanciado tres veces, una para cada producto en particular: se dice que son tres objetos. La cafetería está dada en una estructura que tiene entre sus miembros a tres objetos instanciados a partir del producto.

Una nueva sintaxis

Primero ver el prototipo de un método declarado dentro de la estructura **`Producto`**:

```
void mostrarProducto();
```

Simplemente permite a cada instancia de la estructura **`Producto`** mostrar sus datos. En su definición:

```
void Producto::mostrarProducto() {...}
```

la sintaxis emplea el operador de alcance para ligar la estructura **`Producto`** con su método **`mostrarProducto()`** y no lleva parámetros porque los datos ya se conocen: están dentro de la misma estructura. Ahora véase el prototipo de un método declarado dentro de la estructura **`Cafeteria`**:

```
void mostrarMenu(); // muestra todos los productos
```

Permite mostrar todos los productos en venta una instancia de la estructura **`Cafeteria`**. En su definición:

```
void Cafeteria::mostrarMenu() {...}
```

la sintaxis emplea el operador de alcance para ligar la estructura **`Cafeteria`** con su método **`mostrarMenu()`** y no lleva parámetros porque los datos ya se conocen: están dentro de la misma estructura.

Cuando se declara una estructura, el compilador C++ especifica nombre, e identifica miembros y métodos. Cuando se define una o más variables de ese tipo, entonces es que se separa memoria para alojar completamente sus funciones y el tipo de dato se concreta en un objeto: se crean éstos y se pueden dar valores a sus miembros y usar sus métodos dentro de un programa.

La protección de los datos

Estas son las características que garantizan la protección de los datos:

- **`jardín`** es un objeto que pertenece a la clase **`Cafeteria`**. **`helado`** es un objeto que pertenece a la clase **`Producto`**. Los objetos son variables que pertenecen a sus respectivas clases. Los métodos discutidos, al pertenecer a estructuras diferentes, son llamados por sus respectivos objetos. ¿Resultado? No hay pérdida ni equivocaciones en su empleo.
- Al conocer cada objeto sus datos, usualmente las interfaces de sus métodos están vacías, o son muy breves. Se aprecia en el programa ejemplo que el código es más fácil de escribir e insensible a los errores de pase de parámetros. ¿Resultado? Las funciones están atadas a los miembros de la estructura, que no al programa.

  En la práctica es muy difícil que pueda haber algún tipo de error al llamarlas con parámetros cambiados o erróneos, porque sus interfaces son tersas o están vacías. Por otro lado, también hay que generar más código para crear métodos que satisfagan las demandas de cada clase involucrada, aunque son bastantes cortos y van dentro de módulos muy bien encapsulados.

- Cuando el programa (el cliente) llama al objeto **`jardin`** (el servidor) a través de la instrucción **`jardin.mostrarMenu()`**, está enviando un mensaje que dice: *objeto jardín, muestra tu menú.* Cuando el objeto **`jardin`** (que ahora deviene en cliente), llama al objeto **`refresco`** (que pasa ahora a ser el servidor) a través de la instrucción **`refresco.mostrarProducto()`**, está enviando un mensaje que dice: *objeto refresco, muestra tu producto*. Entonces, en ningún caso se puede tomar cualquier otro método del objeto por equivocación, o por descuido tomar el método con datos erróneos, o tomar un método perteneciente a otro objeto. ¿Resultado? Los datos ahora están mucho más protegidos.

La seguridad de los datos

Ahora bien, dado que todos los miembros de la estructura son públicos[28], o sea, pueden ser llamados desde cualquier lugar del programa y por cualquiera de sus instrucciones, la encapsulación (propiedad básica de una clase tradicional, que limita y controla esos accesos) no existe, no les protege del mal uso, y en ese sentido la estructura C++ tal y como se muestra no es más segura que la estructura ANSI C. Sólo es más cómoda de usar, más eficaz que la clásica y bastante eficiente a la hora del manejo de complejidades. Y así se usa en esta parte de la serie: programando en súper C.

El producto a vender

Haciendo una abstracción básica, para el problema un producto posee un nombre, un precio y la cantidad que del mismo se vendió en el día, y debe saber cómo mostrarse en consola. La recaudación puede ser parte del producto o ser calculada por demanda de la cafetería, siendo una decisión de diseño, que aquí se hizo parte del producto en aras de una codificación más sencilla. La interfaz de su estructura debe reflejar estos hechos.

---

[28] Se puede aplicar la encapsulación en una estructura C++, pero acá no se contempla.

Para construir un objeto del producto lo que se hace en su constructor es pasar el nombre de la oferta y su precio en su interfaz y el método pone a todos los datos en un estado inicial conocido. Por ejemplo:

```
//  constructor (nomb: su nombre, prec: su precio)
Producto(Str nomb = "", float prec = 0);
```

La cafetería

En la misma línea, una cafetería posee una razón social (su nombre), tiene varios productos a la venta, debe saber cómo recaudar las ventas del día, calcularlas y saber cuándo terminó el trabajo. La interfaz de su estructura debe reflejar estos hechos. Para construir un objeto de la cafetería lo que se hace en su constructor para resolver este problema, es pasar sus tres ofertas y su nombre en la interfaz y el método pone a todos los datos en un estado inicial conocido. Por ejemplo:

```
//  constructor: n es nombre, p es precio -- Ref es refresco, Hel es helado y Dul es dulce
//  nomb es el nombre de la cafetería
Cafeteria(Str nRef, float pRef, Str nHel, float pHel, Str nDul, float pDul, Str nomb);
```

Declaraciones

- El menú

```
#ifndef MENU_H_INCLUDED
#define MENU_H_INCLUDED

#include <string>
typedef std::string Str;

//  enumera lo que pide el menú
enum { mostrar, vender, recaudar };

//  un menú en consola
//  entrada: la información al usuario
//  salida: menú a consola
short menu(Str info);

#endif // MENU_H_INCLUDED
```

- El producto

```
#ifndef PRODUCTO_H_INCLUDED
#define PRODUCTO_H_INCLUDED

#include <string>
typedef std::string Str;

struct Producto {
  //  constructor (nomb: su nombre, prec: su precio)
  Producto(Str nomb = "", float prec = 0);
  //  método
  void mostrarProducto();
  //  miembros
  Str nombre;
  int cantidad;
  float precio;
  float recaudo;
};
```

```
#endif // PRODUCTO_H_INCLUDED
```

- La cafetería

```
#ifndef CAFETERIA_H_INCLUDED
#define CAFETERIA_H_INCLUDED

#include "producto.h"

struct Cafeteria {
  //  constructor: n es nombre, p es precio
  //  Ref es refresco, Hel es helado y Dul es dulce
  //  nomb es el nombre de la cafetería
  Cafeteria(
    Str nRef, float pRef,
    Str nHel, float pHel,
    Str nDul, float pDul,
    Str nomb
  );
  //  métodos
  void mostrarMenu();
  void tomarVenta();
  void mostrarRecaudo();
  //  miembros
  Str nombre;
  bool estaHecho;
  //  miembros agregados
  Producto refresco;
  Producto dulce;
  Producto helado;
};

#endif // CAFETERIA_H_INCLUDED
```

Definiciones

- El menú

```
#include "menu.h"
#include <iostream>
#include <cstdlib>
#include <cctype>

//  un menú en consola
short menu ( Str info ) {
  Str conjunto = "OVR";
  Str opcion;
  Str::size_type idx;

  while ( true ) {
    system("cls");
    std::cout << info <<
    "Sistema de venta diaria\n\n"
    "(O)fertas, (V)entas, (R)ecaudaci\242n\n"
    "Entre una opci\242n: ";
    (std::cin >> opcion).get();
    idx = conjunto.find(toupper(opcion[0]));

    if ( Str::npos == idx ) {
      std::cerr << "\nError: opci\242n ilegal.\n\n";
```

```
      system("pause");
    } else {
      return idx;
    } // if-else
  } // while
}
```

- El producto

```
#include "producto.h"
#include <iostream>
#include <iomanip>
using std::cout;
using std::setw;

//  construye una instancia del producto
Producto::Producto(Str nomb, float prec) {
  nombre   = nomb;
  precio   = prec;
  cantidad = 0;
  recaudo  = 0;
}

//  muestra el producto
void Producto::mostrarProducto() {
  cout << std::left <<
  setw(10) << nombre << std::setprecision(2) << std::fixed <<
  "$ " << precio << '\n' << std::setprecision(0) << std::right;
}
```

- La cafetería

```
#include "cafeteria.h"
#include <iostream>
#include <iomanip>
using std::cout;
using std::cin;
using std::setw;

//  construye una instancia de la cafeteria, con tres productos
Cafeteria::Cafeteria( Str nRef, float pRef,
                      Str nHel, float pHel,
                      Str nDul, float pDul,
                      Str nomb ) {
  // primero los de la clase
  nombre = nomb;
  estaHecho = false;
  // luego los valores agregados
  refresco.nombre = nRef; refresco.precio = pRef;
  helado.nombre   = nHel; helado.precio   = pHel;
  dulce.nombre    = nDul; dulce.precio    = pDul;
}

//  muestra las ofertas
void Cafeteria::mostrarMenu() {
  cout << '\n' << nombre << " - Men\243.\n\n" << std::left <<
  setw(10) << "Oferta" << "Precio\n";
  refresco.mostrarProducto();
  helado.mostrarProducto();
  dulce.mostrarProducto();
  cout << '\n';
}
```

```
//  toma las ventas y la recaudación del día
void Cafeteria::tomarVenta() {
  cout << '\n' << nombre << "  -  Ventas del d\241a:\n";
  cout << refresco.nombre << "? ";
  cin  >> refresco.cantidad;
  refresco.recaudo = refresco.cantidad * refresco.precio;
  cout << helado.nombre << "? ";
  cin  >> helado.cantidad;
  helado.recaudo = helado.cantidad * helado.precio;
  cout << dulce.nombre << "? ";
  cin  >> dulce.cantidad;
  dulce.recaudo = dulce.cantidad * dulce.precio;

  //  ya puede salir
  estaHecho = true;
  cout << '\n';
}

//  muestra la recaudación calculada por demanda
void Cafeteria::mostrarRecaudo()  {
  //  encabezamiento
  cout << '\n' << nombre << " - Total ventas del d\241a:\n\n" <<
  std::fixed << std::setprecision(2) <<
  setw(10) << std::left  << "Oferta" <<
  setw(9)  << std::right << "Cant."  <<
  setw(8)  << "Precio"   <<
  setw(9)  << "Venta"    << '\n';

  //  tabla
  double total = 0, importe = 0;
  if ( refresco.recaudo > 0 ) {
    importe = refresco.recaudo;
    total += importe;
    cout <<
    setw(10) << std::left  << refresco.nombre <<
    setw(8)  << std::right << refresco.cantidad <<
    setw(9)  << refresco.precio <<
    setw(9)  << importe << '\n';
  } // if

  if ( helado.recaudo > 0 ) {
    importe = helado.recaudo;
    total += importe;
    cout <<
    setw(10) << std::left  << helado.nombre <<
    setw(8)  << std::right << helado.cantidad <<
    setw(9)  << helado.precio <<
    setw(9)  << importe << '\n';
  } // if

  if ( importe > 0 ) {
    importe = dulce.recaudo;
    total += importe;
    cout <<
    setw(10) << std::left  << dulce.nombre <<
    setw( 8) << std::right << dulce.cantidad <<
    setw( 9) << dulce.precio <<
    setw( 9) << importe << '\n';
  } // if
```

```
    // pie
    cout << "Total " << std::setfill('.') << setw(30) << total  << "$\n";
  }
```

Programa

```
#include "menu.h"
#include "cafeteria.h"
#include <iostream>
#include <cstdlib>

int main() {
  // información
  Str info = "Programa 18: cafeter\241a El Jard\241n\n";

  // estructura C++
  Cafeteria jardin( "Refresco", 1.00, "Helado", 0.55, "Rollito", 1.20, "El Jard\241n" );

  // el programa
  unsigned opcion;
  while ( true ) {
    opcion = menu( info );
    switch ( opcion ) {
    case mostrar:
      jardin.mostrarMenu();
      break;
    case vender:
      jardin.tomarVenta();
      break;
    case recaudar:
      if ( jardin.estaHecho ) {
        jardin.mostrarRecaudo();
        return EXIT_SUCCESS;
      } else {
        std::cerr << "\nPrimero entrar las ventas.\n\n";
      } // if-else
      break;
    } // switch

    system("pause");
  } // while
}
```

Una posible salida

El menú acepta letras mayúsculas, minúsculas o comandos.

```
Programa 18: cafetería El Jardín
Sistema de venta diaria

(O)fertas, (V)entas, (R)ecaudación
Entre una opción: o

El Jardín - Menú.

Oferta    Precio
Refresco  $ 1.00
Helado    $ 0.55
Rollito   $ 1.20

Presione una tecla para continuar . . .
```

CUARTO BLOQUE DE EJERCICIOS

Considerar todos los tópicos anteriormente estudiados. No hay obligación de usar ninguna instrucción en especial, pero se deben aprovechar los nuevos conocimientos tratando de aplicarlos a cada ejercicio, empleando el buen juicio.

21. El programa 17 pedía darle al usuario la opción de escoger entre poner su cadena en mayúsculas, minúsculas o tipo oración, pero no trabajaba bien con el español. Rehacerlo para que acepte frases en español y realice los cambios correctamente. Sugerencia: crear las funciones **`aMayusculas`** y **`aMinusculas`** para procesar una frase en español. Se estipula que la oración debe ser en minúsculas con la primera letra en mayúsculas, que puede estar precedida de una apertura de interrogación o exclamación y que además puede ser ASCII o española. La salida debiera así:

```
Problema 8: cambio de tamaño

(M)ayúsculas, mi(N)úsculas, (O)ración, (T)erminar.
Entre una opción: m

   Entre una frase: una puñetera tade de mayo.
Frase a mayúsculas: UNA PUÑETERA TADE DE MAYO.

Presione una tecla para continuar . . .
```

22. La baraja española clásica consiste en un mazo de 40 naipes, divididos en cuatro palos que son: oros, copas, espadas y bastos; y los números son As, 2, 3, 4, 5, 6, 7, Sota, Caballo y Rey, todo lo cual facilita el control de los ciclos por simple enumeración. Se pide barajar el mazo.

    El algoritmo de Fisher & Yates *barajado del sombrero* (Wikipedia en español), describe la acción de mezcla que se lleva a cabo cuando se hace una rifa. La lista siempre contiene los mismos valores, simplemente es reordenada de otra manera cambiando las posiciones, que son dependientes del algoritmo de aleatoriedad. Su seudocódigo en C++ es:

```
Algoritmo barajado:
   int az;
   int tmp;
   para i (MAZO-1, i mayor que cero, disminuyendo a i en uno)
      az       ← un valor azaroso entre 0 y MAZO-1
      tmp      ← mazo[az]
      mazo[az] ← mazo[i]
      mazo[i]  ← tmp
   fin-para
fin-barajado
```

    Una posible salida sería algo como:

```
Problema 22: un barajado de cartas españolas.

Jugador 1          Jugador 2          Jugador 3         Jugador 4
--------------------------------------------------------------------------
As de oros         Cuatro de oros     Dos de oros       As de copas
Tres de copas      Cuatro de bastos   Tres de oros      Siete de bastos
Sota de espadas    Caballo de espadas Dos de espadas    Sota de bastos
Rey de bastos      Siete de oros      Cinco de oros     As de espadas
Dos de bastos      Tres de bastos     Dos de copas      Sota de oros
Caballo de oros    Rey de copas       Caballo de bastos Seis de oros
Cuatro de copas    Siete de copas     Cinco de copas    Sota de copas
As de bastos       Seis de copas      Seis de bastos    Caballo de copas
Cuatro de espadas  Rey de oros        Siete de espadas  Tres de espadas
Seis de espadas    Cinco de espadas   Cinco de bastos   Rey de espadas
--------------------------------------------------------------------------
```

23. El producto escalar de vectores o producto-punto permite determinar ángulos y distancias en el plano real $(\mathbb{R}^2)$ de una forma fácil y directa. Si los vectores $\boldsymbol{A}$ y $\boldsymbol{B}$ se expresan en función de sus componentes cartesianas rectangulares, el producto punto se realiza como un producto matricial de la siguiente forma:
    - $\boldsymbol{A} \cdot \boldsymbol{B} = [a_1, b_1] \begin{bmatrix} a_2 \\ b_2 \end{bmatrix} = a_1 a_2 + b_1 b_2$
    - La norma o módulo de un vector se define como su tamaño y se calcula como: $|\boldsymbol{A}| = \sqrt{a^2 + b^2}$
    - El ángulo entre dos vectores en el plano se calcula como: $cos\theta = \frac{\boldsymbol{A} \cdot \boldsymbol{B}}{|A||B|}$ radianes
    - $\boldsymbol{A}$ y $\boldsymbol{B}$ son perpendiculares si su coseno es cero, paralelos si es 1 y anti-paralelos si es -1

    Codificar un programa que pida las coordenadas de dos vectores, calcule el tamaño de cada uno, su producto punto y su posición respectiva: paralelos, anti-paralelos, perpendiculares, o el ángulo entre ellos, en radianes y grados. El valor de la conversión se tomó de la calculadora HEXelon MAX6, de Jerzy Znamirowski. La salida debe verse más o menos así:

```
Problema 23: cálculo del producto-punto de dos vectores

Vector A en formato (x,y): (9.8,6.5)
Vector B en formato (x,y): (5.4,2.1)

Vector A (9.80,6.50)
Vector B (5.40,2.10)

El producto-punto de A y B vale 66.57

Norma de A = 11.76 u
Norma de B = 5.79 u

El coseno entre A y B vale 0.017 rad o 55.98 grados.
```

24. En el pañol de la universidad pedagógica Blas Roca, donde se guardan las herramientas de la carrera Educación Laboral, existen las siguientes herramientas automáticas y su precio por unidad: Sierra de calar a $6.80, Sierra de trocear a $8.99, Atornillador a $3.55 y Lijadora plana a $11.22 - Mediante el llenado de un modelo, un docente pide al pañol la cantidad de estas herramientas que necesita para una clase práctica y el pañolero las registra y además el costo del préstamo, quién hizo el pedido y en qué fecha lo hizo. Diseñar una estructura adecuada al problema y escribir un pequeño programa para probarla.

    Observar que las herramientas pueden ser consideradas una estructura que lleva su nombre, costo y cantidad del pedido, y se puede crear por agregación una estructura pañol que lo maneje todo. Con lo ya conocido del formateo, use los manipuladores y trate de conseguir algo así:

```
Problema 24: Pedido del pañol del Blas Roca.

Stock del pañol:
Herramienta        Precio
Sierra de calar.......6.80
Sierra de trocear.....8.99
Atornillador..........3.55
Lijadora plana.......11.22

Nombre completo del docente: Salvador Aldorso
Fecha del pedido - día/mes/año: 4/9/2020
   Sierra de calar = 6
 Sierra de trocear = 6
      Atornillador = 15
    Lijadora plana = 3

Hoja de pedidos.
Salvador Aldorso - Fecha: 4/9/2020
Herramienta         Precio   Cant.      Costo
Sierra de calar       6.80       6       40.80
Sierra de trocear     8.99       6       53.94
Atornillador          3.55      15       53.25
Lijadora plana       11.22       3       33.66
Costo total:............................181.65$
```

25. El problema 7 pedía codificar una tabla de potencias de números enteros para el cuadrado, cubo e inverso, del uno al diez. Con lo ya conocido del formateo de las salidas, usar los manipuladores y conseguir números alineados correctamente con precisión de 4 decimales:

```
Problema 25: tabla formateada.

  N   Cuad.    Cubo    Inverso
------------------------------
  1       1       1     1.0000
  2       4       8     0.5000
  3       9      27     0.3333
  4      16      64     0.2500
  5      25     125     0.2000
  6      36     216     0.1667
  7      49     343     0.1429
  8      64     512     0.1250
  9      81     729     0.1111
 10     100    1000     0.1000
------------------------------
```

26. En las calculadoras modernas se puede buscar la equivalencia en el cambio de unidades entre magnitudes físicas. Codificar un programa que emule parte de la función de conversión de la calculadora HEXelon MAX, hecha por Jerzy Znamirowski (www.HEXelon.com) y buscar la equivalencia para: galón(gl)/litro(l); pie cúbico(ft^3)/metro cúbico(m^3); libra(lb)/kilogramo(kg); acre(ac)/hectárea(ha); grado Fahrenheit(F)/grado Celsius(C); radián(rad)/grado(grado) –el radián se debe dar al menor ángulo posible. La salida debería verse más o menos así:

```
Problema 26: calculadora de conversión.

1) Galón (gal) y Litro (l)
2) Pie cúbico (ft3) y metro cúbico (m3)
3) Libra (lb) y Kilogramo (kg)
4) Acre (ac) y Hectárea (ha)
5) Milla (mi) y Kilómetro (km)
6) Grados Fahrenheit (F) y grados Celsius (C)
7) Radián (rad) y grado(grado)
0) Terminar

Entre una opción: 6
Unidades a convertir: 14.72

 14.7200 F -->  -9.6000 C
 14.7200 C -->  58.4960 F

Presione una tecla para continuar . . .
```

12 - PUNTEROS

> *Los punteros (**`pointers`**) son variables especializadas que sólo reciben direcciones de memoria de cualquier otro tipo de variables, incluyéndolos a ellos mismos, y permiten realizar gestiones oportunas sobre esas direcciones. Un puntero es como el número de página que un índice asocia con una palabra; si va al número de la página, encuentra la palabra que busca. El Autor.*

Tema de comprensión nada fácil, que EODA *requiere de esfuerzo extra para ser asimilado a cabalidad*, los punteros conforman una de las herramientas más potentes de C++, principalmente porque dan acceso y control sobre la tienda gratis, pero también son muy peligrosos, porque permiten realizar acciones hostiles al programa: los programadores pueden acceder directamente a las direcciones de memoria mediante el empleo de punteros y si se equivocan, casi siempre cometen errores irrecuperables en la ejecución, y dificultosos de encontrar en una acción de depuración.

El atento lector deberá tener cuatro elementos bien presentes:

1. *Un puntero en manos inexpertas es como una granada sin espoleta.* El problema está al declarar los punteros y apuntarlos: ellos manejan directamente la memoria. Mal declarados o mal encaminados pueden apuntar dondequiera, y ahí puede haber cualquier tipo de dato, literalmente cualquiera…la granada puede estallar y tarde o temprano lo hace. Leer ahí es leer "basura" para el programa. Escribir ahí es al menos temerario…Frente a punteros no cabe otra; hay que pensárselo bien.
2. *En C++ el uso estándar del operador asterisco está sobrecargado en cuatro ocasiones, y eso es una fuente real de ofuscación.* Con el mismo operador asterisco el programador novel puede confundirse…y muchas veces lo hace, pero el compilador "sabe" perfectamente **(a)** cuando se usa para multiplicar; **(b)** cuándo se usa para declarar un puntero; **(c)** cuando se usa para indireccionar su valor; y **(d)** cuando se usa para reservar memoria en la tienda gratis.
    - ✓ Indirección o desreferencia es tomar el valor de una variable mediante un puntero.
3. *La lectura humana del código se torna difícil y más aún cuando se trabaja con punteros que apuntan a otros punteros, que a su vez apuntan a otros*…Bueno, esa es la idea, ya que la norma admite hasta 12 niveles seguidos de llamadas a punteros.
4. *Los punteros en C++ son la única vía del manejo de la memoria residente en la tienda gratis*. No pueden ser ignorados pues sin su empleo se limita severamente el tamaño de las aplicaciones y la cantidad de variables usadas en problemas

serios, de calidad profesional que pudieran ser realizados en este lenguaje. No queda de otra: hay que aprender a trabajarlos.

El trabajo intensivo con punteros tiende a producir código ofuscado

Modelos de memoria

Según sea la mecánica de reservar memoria para las variables, C++ la administra de tres formas y los elementos del código que residen en estos bloques de memoria se diferencian en su tiempo de vida, y en el lugar (número) que ocupan sus direcciones.

1. Almacenaje estático: es el que usan las variables que existen durante toda la ejecución del programa. Una variable es estática si se declara fuera de **`main()`**, o dentro de cualquier otra función, si se le antepone el modificador **`static`**
2. Almacenaje automático: es el que usan por omisión las variables definidas dentro de una función o un bloque de instrucciones. Vienen a existir automáticamente, cuando el bloque o la función que las contiene son llamados, y expiran automáticamente cuando el bloque o la función terminan. Se les puede anteponer el modificador **`auto`**, que prácticamente nunca se usa porque así son por omisión. A medida que el programa las emplea, van a la pila de control, quien crece en la dirección de la tienda gratis, o se reduce en cuanto se sale del bloque que las usa, en dirección opuesta como si fuera una parte de un acordeón.
3. Almacenaje dinámico: es la otra parte de la memoria compartida con la pila. A medida que el programador la emplea, la tienda gratis crece en la dirección de la pila, o cuando la libera, se reduce en dirección opuesta como si fuera la otra parte del acordeón.

Puntero nulo

Algunas veces los programadores usan ***`(void*)0`*** para identificar el puntero nulo. (El puntero nulo mismo puede ser una representación interna diferente de cero.) Otros programadores usan la macro **`NULL`**, definida en C para representar el puntero nulo. Sin embargo, esto resultó ser una solución incompleta. C++11 provee una mejor solución introduciendo una palabra clave nueva, ***`nullptr`***, para denotar el puntero nulo. Todavía Ud. puede usar ***`0`*** como antes —de otra manera una cantidad enorme de código existente quedaría deshabilitada— pero de ahora en adelante, la recomendación es usar ***`nullptr`*** en su lugar. (Prata, p. 648)

El puntero nulo es aquél cuyo valor es una dirección que garantiza no apuntar a ningún dato válido. Esto puede lograrse asignando el valor nulo correcto, que puede ser de tres maneras: usar **`nullptr`**, forma que C++11 recomienda emplear siempre en código nuevo; usar la macro **`NULL`**, de la biblioteca **`<cstdlib>`** o ante la duda, poner simplemente el valor cero**.**

Puntero void

Es posible declarar un puntero genérico, sin un tipo concreto, llamado **`void`**. Por ejemplo, cualquiera de las tres formas mostradas indica un puntero **`void`**. En la declaración de cualquier puntero, el asterisco puede "flotar" entre el tipo y su nombre, como se muestra; el lector usará la forma que mejor le plazca: la primera es propia de C++, la que sigue es neutral y la que cierra es tipo ANSI C, pero todas significan lo mismo: **`voidPtr`** es un puntero vacío.

```
void* voidPtr;  void * voidPtr;  void *voidPtr;
```

Hay que tener presente que un puntero **`void`**:

- Puede contener cualquier dirección arbitraria y en momentos diferentes podría apuntar a cualquier tipo de dato: un **`char`**, un **`double`**, una estructura de algún tipo, etc.

- No registra ninguna información acerca de lo que apunta; es responsabilidad del programador seguirle la pista.
- No se puede indireccionar, hay que encasillarlo apropiadamente.
- No permite hacer directamente aritmética de punteros, hay que encasillarlo apropiadamente.

Los punteros **void** pueden apuntar a objetos de cualquier tipo, siendo ideales para ser usados de comodines, pero se advierte que para indireccionarlos o hacer con ellos aritmética de punteros, hay que aplicarles el encasillamiento adecuado. Ver a (Jaeschke, Void Pointers) o a la

Tabla 21

Tabla 21. Formas de casting o encasillamiento.

No confundir el puntero nulo con el **void**. El primero tiene el valor nulo, cualquiera que aquel sea en su sistema, mientras que el segundo puede apuntar a cualquier dato en concreto sin tener que especificar su tipo.

El puntero **void** siempre tiene la dirección de algún tipo de dato. El puntero nulo siempre tiene el valor nulo

Puntero constante

Hay cuatro maneras de hacer constante a un puntero: haciendo constante al tipo al cual apunta el puntero, haciendo constante al puntero, haciendo constantes ambos o haciendo constante al puntero cuando es pasado como parámetro a una función:

- ✓ Puntero constante es aquel que ha sido modificado de alguna forma con el calificador **const** el cual afecta lo que va a su derecha.

1. Poniendo constante al tipo del puntero:

```
const int *pConst = &intVar; // const afecta al tipo int
```

const afecta a **int**, haciendo constante el contenido del valor al cual se apunta, y aunque el puntero puede cambiar la dirección a la que apunta, el valor que esté presente a dondequiera que apunte, lo ve constante, convirtiéndole en un puntero de sólo lectura (*r/o pointer* o *read only pointer* en inglés.) La variable en sí no es constante y por lo mismo su valor se puede cambiar, pero sobre la misma variable. A este tipo de puntero también se puede asignar el valor de una constante:

```
const float pulgada = 25.4;
const float *pMM = &pulgada;
```

2. Poniendo constante al puntero:

```
int* const pHead = &tope; // const afecta al puntero
```

const afecta a **pHead**, haciendo constante la dirección a la que apunta el puntero, el cual no puede moverse de esa dirección. Es un puntero fijo (*fixed pointer* en inglés) y aunque el valor apuntado puede ser modificado, la dirección a la que apunta es fija, por lo que al declararlo hay que inicializarlo.

3. Poniendo constantes a ambos:

```
const float* const pMM = &pulgada; // const afecta al tipo y al puntero
```

Ahora **pMM** sólo puede apuntar al valor de la variable **pulgada** y no se puede ni mover, ni usarlo para cambiar el valor.

4. Poniendo **const** al puntero pasado como parámetro a una función, en la interfaz de un procedimiento o función.

```
void miProc(const int *pVal); // la función no puede cambiar al valor que apunta el puntero
```

Al declarar a **pVal** como parámetro constante, la función puede usar su valor, pero no puede modificarlo bajo ninguna forma de codificación. Esto protege el valor al que apunta el puntero para que no sea modificado por el trabajo de la rutina.

Puntero extraviado

Si a un puntero que ha sido usado para reclamar memoria en la tienda gratis se le borró con **delete** la dirección que le fue asignada con **new** *y todavía tiene tiempo de vida*, entonces sigue apuntando a la vieja dirección y si es llamado, la regresa. Está extraviado, porque esa área de memoria ya se había liberado y en el ínterin, el compilador pudo haber escrito otra cosa allí.

- ✓ Puntero extraviado, también conocido como puntero colgante (*wild pointer* o *dangling pointer* en inglés), es aquél al que se la aplicó el operador **delete** y no fue reasignado.

Es un error de ejecución muy desagradable y difícil de localizar, porque puede ir desde "nada pasó…todavía" hasta "¡la inesperada catástrofe!". Y mucho peor, el error puede salir a flote en algún momento, durante alguna de las ejecuciones continuadas del programa, delante de quien no debería estar presente, etc. Entonces, ¿cómo encaminar a este puntero para que no se extravíe? Fácil. Cada vez que se le aplique el operador **delete** a un puntero con tiempo de vida, inmediatamente se le pone a nulo.

Puntero *this

Si por alguna razón se necesita la dirección de memoria de una estructura o clase, o de cualquiera de sus elementos, ella lleva oculto, un puntero llamado **this** que produce la dirección del componente deseado. Por ejemplo, para devolver la dirección de una estructura dada, se podría escribir un código parecido a éste:

```
//  devuelve el conjugado de z
Complejo Complejo::conjugado() {
    imZ = -imZ;
    return *this;
}
```

o también:

```
//  suma dos complejos
Complejo Complejo::sumadoCon(Complejo z) {
    reZ += z.reZ;
    imZ += z.imZ;
    return *this;
}
```

El problema con estas formas es que altera directamente las variables de la estructura que llama y los resultados globales dependerán de las condiciones del problema. Es por eso que generalmente ***this** no se usa mucho, pero siempre estará allí. Aunque se dan dos ejemplos donde se utiliza correctamente desde el punto de vista sintáctico, debe tenerse presente que su aplicación todo el tiempo, es indicio de código pobremente programado y lenguaje mal entendido.

Declaración de punteros

Un puntero "normal" en el sentido que no es usado para acceder a la tienda gratis, se declara uniendo un tipo de dato con un identificador de tipo por medio del operador de punteros (*) Por ejemplo se declara un puntero a entero, a un número real, o a un carácter: `int *intPtr; float *fltPtr;  char *chrPtr;`

Al declarar un puntero normal, éste apunta a una dirección de memoria usada por el código, fuera de la tienda gratis y tomada al azar: se dice que apunta a "basura" y hay peligro en usarlo así. Para estar del lado seguro, al declarar un puntero y asignarle la dirección de memoria de una variable, es conveniente hacerlo de este modo:

```
int unEntero = 5;        // primero se declara e inicializa la variable
int *intPtr = &unEntero; // luego se apunta a la variable
```

o también usar el operador coma y hacerlo mejor y de una sola vez, así:

```
int unEntero = 5, *intPtr = &unEntero; // declaración y asignación juntas
```

Si un puntero global apunta a una variable local perteneciente a una función, método o bloque, cuando la función o el método terminen, o el flujo del código salga del bloque y la variable desaparezca, el puntero retiene una dirección de memoria que no es válida. En esa situación, se advierte se ponga a nulo, o quedará extraviado.

Operadores de punteros

Son dos: el ampersand o *y-comercial* (**&**) y el asterisco (*****)

Tabla 25. Operadores de punteros

Op.	Significado	Acción	Ejemplo
&	Operación de dirección	Toma una dirección y la pone en el puntero.	`ptr = &x;`
*****	Operación de indirección	Muestra el valor de la variable a que apunta.	`cout << *ptr;`

Programa 19: una primera mirada

Codificar un programa que declare un puntero (puede ser local o global) y dos variables: una local y otra global. Mostrar sus relaciones. Para fines didácticos, se hará monolíticamente. Notar cómo las variables residen en segmentos de memoria diferentes, y cómo se pueden cambiar esos valores a través del puntero. El puntero, local o global, residirá en el segmento adecuado.

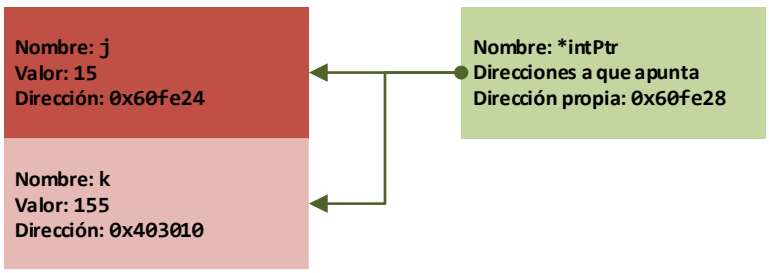
Programa

```
#include <iostream>
#include <cstdlib>

int k = 155; //  variable declarada globalmente

int main() {
  int j = 15;          // variable declarada localmente
  int * ptr = nullptr; // puntero declarado localmente

  //  operaciones locales
  ptr = &j; // ptr apunta a j
  std::cout <<
  "Programa 19: una mirada a un puntero.\n\n"
  "Apuntando a la direcci\242n local:\n"
  "La variable LOCAL j vale "            <<  j   <<
  " y su direcci\242n es "                << &j   << '\n' <<
  "El puntero (&j) apunta a "            <<  ptr << '\n' <<
  " su indirecci\242n (*ptr) es "         << *ptr << '\n' <<
  " y su propia direcci\242n (&ptr) es " << &ptr << " <--\n\n";

  *ptr = 18;  // cambiando el valor a través del puntero
```

```
  std::cout << "Si hacemos *ptr = 18, ahora j vale " << j << "\n\n";

  //  operaciones globales
  ptr = &k; // ahora ptr apunta a k
  std::cout <<
  "Apuntando ahora a la direcci\242n global:\n"
  "La variable GLOBAL k vale "                  <<  k   <<
  " y su direcci\242n es "                      << &k   << '\n' <<
  "El puntero (&k) ahora apunta a "             <<  ptr << '\n' <<
  " su indirecci\242n (*ptr) ahora es "         << *ptr << '\n' <<
  " pero su direcci\242n (&ptr) sigue siendo " << &ptr << " <--\n\n";

  *ptr = 158; // cambiando el valor a través del puntero

  std::cout <<
  "Si hacemos *ptr = 158, ahora k vale " << k << "\n\n"
  "El mismo puntero apunta a dos variables distintas\n"
  "en dos ocasiones diferentes.\n\n";

  //  fin
  system ( "pause" );
  return EXIT_SUCCESS;
}
```

Salida

```
Programa 19: una mirada a un puntero.

Apuntando a la dirección local:
La variable LOCAL j vale 15 y su dirección es 0x60fe24
El puntero (&j) apunta a 0x60fe24
 su indirección (*ptr) es 15
 y su propia dirección (&ptr) es 0x60fe28 <--

Si hacemos *ptr = 18, ahora j vale 18

Apuntando ahora a la dirección global:
La variable GLOBAL k vale 155 y su dirección es 0x403010
El puntero (&k) ahora apunta a 0x403010
 su indirección (*ptr) ahora es 155
 pero su dirección (&ptr) sigue siendo 0x60fe28 <--

Si hacemos *ptr = 158, ahora k vale 158

El mismo puntero apunta a dos variables distintas
en dos ocasiones diferentes.
```



lustración 8. Un puntero apunta a 2 segmentos en 2 ocasiones diferentes

Programa 20: encasillamientos

Codificar un programa que declare una variable **float** y tres punteros a tipos de datos **float**, **char** y **void**. Maneje sus actuaciones con encasillamientos adecuados. Para fines didácticos, se hará monolíticamente. Notar cómo todos apuntan a la misma variable, y estudiar con detención sus encasillamientos:

- Como **void** es compatible con cualquier puntero, **static_cast** es apropiado.
- Cuando se encasilla a **float** un puntero de tipo **char**, se requiere del **reinterpret_cast** para poder cambiar de tipo.

Programa

```
#include <cstdlib>
#include <iostream>
#include <iomanip>
#include <cmath>

int main() {
    //  declaración e inicialización de una variable local
    float pi = 4 * atan(1);

    //  declaración e inicialización de punteros
    float * pFloat = nullptr;
    char  * pChar  = nullptr;
    void  * pVoid  = nullptr;

    std::cout << std::left <<
    "Programa 20: declara una variable y unos punteros, todos de\n"
    "distintos tipos, y los encasilla para que la representen.\n\n"
    "Variable local pi, de punto flotante:\n"
    << std::setw(12 ) << "  Valor:"
    << std::setw( 8 ) << pi  << " <--\n"
    << std::setw(12 ) << "  Direci\242n:"
    << std::setw( 8 ) << &pi << " <--\n\n"
    "Los punteros son a punto flotante (pFloat) a un car\240cter (pChar) y a\n"
    "void (pVoid). pChar debe ser encasillada, ya que apunta a un float.\n";
    pFloat = &pi;                           // pFloat apunta a pi
    pChar  = reinterpret_cast<char *>(&pi); // pChar  apunta a pi
    pVoid  = &pi;                           // pVoid  apunta a pi

    std::cout <<
    "Valor a que todos apuntan:\n\n" <<
    "  pFloat: " << pFloat                            << " <--\n"
    "  pChar:  " << reinterpret_cast<float *>(pChar) << " <--\n"
    "  pVoid:  " << pVoid                             << " <--\n\n"

    "Aqu\241 se desreferencian los punteros. char y void no son punteros\n"
    "a float, y deben ser encasillados. Valor que todos muestran:\n\n" <<
    "  pFloat = " << *pFloat                            << " <--\n"
    "  pChar  = " << *reinterpret_cast<float *>(pChar) << " <--\n"
    "  pVoid  = " << *reinterpret_cast<float *>(pVoid) << " <--\n\n";

    *pFloat = 3.1416;
    std::cout <<
    "Cambiando el valor de pi a 3.1416 con *pFloat. Solamente los\n"
    "punteros se desreferencian; como pi es una variable, no puede serlo.\n\n"
    "  pi    = " <<  pi                               << " <--\n"
    "  pChar = " << *reinterpret_cast<float *>(pChar) << " <--\n"
    "  pVoid = " << *reinterpret_cast<float *>(pVoid) << " <--\n\n";
```

```
    //  fin
    system ( "pause" );
    return EXIT_SUCCESS;
}
```

Salida

```
Programa 20: declara una variable y unos punteros, todos de
distintos tipos, y los encasilla para que la representen.

Variable local pi, de punto flotante:
  Valor:    3.14159  <--
  Direción: 0x6afe2c <--

Los punteros son a punto flotante (pFloat) a un carácter (pChar) y a
void (pVoid). pChar debe ser encasillada, ya que apunta a un float.
Valor a que todos apuntan:

  pFloat: 0x6afe2c <--
  pChar:  0x6afe2c <--
  pVoid:  0x6afe2c <--

Aquí se desreferencian los punteros. char y void no son punteros
a float, y deben ser encasillados. Valor que todos muestran:

  pFloat = 3.14159 <--
  pChar  = 3.14159 <--
  pVoid  = 3.14159 <--

Cambiando el valor de pi a 3.1416 con *pFloat. Solamente los
punteros se desreferencian; como pi es una variable, no puede serlo.

  pi    = 3.1416 <--
  pChar = 3.1416 <--
  pVoid = 3.1416 <--
```

Notas:

- En ambos programas, las salidas que se muestran son de este sistema. En otro posiblemente diferirán los valores, pero el significado no cambia.
- Al inicializar los punteros (ponerlos a apuntar), como el tipo de dato es **`float`**, ni el puntero a punto flotante, ni el puntero **`void`** necesitan del casting, ya que el primero es compatible con el dato y el segundo es compatible con cualquiera. Pero el puntero a **`char`** si necesita casting, y de reinterpretación, pues no es **`float`**.
- Al mostrar los valores, hay que indireccionar los punteros, pero como el **`void`** no puede serlo directamente, hay que encasillarlo, y se puede hacer con ambos tipos de castings, pero se prefiere el estático. Igualmente hay que encasillar al **`char`**, que ya se vio no es un **`float`**, pero ese sí que hay que reinterpretarlo.

Operaciones con punteros

Con los punteros se pueden hacer las siguientes operaciones y cada una de ellas tiene sus características propias:

1. Asignación de variable a puntero o de puntero a puntero.
2. Suma y resta de punteros con enteros.
3. Resta entre punteros del mismo tipo.
4. Comparación entre punteros del mismo tipo.

Asignaciones

En el apartado anterior se vio cómo asignar la dirección de una variable a un puntero. Por ejemplo:

```
float pi = 4 * atan(1), *pFloat = &pi; // declaración e inicialización de una variable
```

También se puede asignar un puntero a uno **`void`** o a otro del mismo tipo, y los dos apuntarán a la misma dirección. Por ejemplo:

```
int i = 5, *p1 = &i, *p2 = p1;
```

asigna la dirección de **`i`** a **`p1`** y la dirección de **`p1`** a **`p2`**, todo en una instrucción, usando el operador coma, quien se sabe, trabaja de izquierdas a derechas.

Sumas con enteros

La suma de punteros con enteros incrementa la dirección almacenada en una o varias veces el tamaño del tipo del puntero, según sea el tamaño del tipo de dato. La resta opera de modo similar, pero disminuyendo posiciones.

Esto simplemente es la famosa Aritmética de Punteros

Caso 1: suma y resta en arreglos

Sea el siguiente fragmento de código:

```
int ary[] = { 12, 7, 5, 3, 4 };
int *pAry = ary;
```

- El nombre del arreglo es *una especie de puntero disfrazado* que apunta al primer elemento, el que tiene el índice cero.
- La instrucción **`*pAry`** mostrará el valor 12.
- La instrucción de asignación **`pAry = pAry + 1`** (o aún mejor, **`pAry += 1`**) desplazará la dirección del puntero en **`1*sizeof(int)`** bytes; el puntero se desplazará a la dirección indicada (la del segundo valor, que pasa a ser la dirección actual) y la instrucción **`*pAry`** mostrará el valor 7.
- La instrucción **`*(pAry + 2)`** sin desplazar al puntero de su posición actual, mostrará el valor que está 2 posiciones después (**`2*sizeof(int)`** bytes): mostrará el valor 3.
- La instrucción **`*pAry + 2`** incrementará el valor mostrado por **`*pAry`** en dos unidades: mostrará el valor 7 + 2 = 9.
- La instrucción **`cout << pAry[i*col + j];`** multiplica el valor del índice **`i`** por el valor **`col`** y le suma el valor del índice **`j`**. El resultado es el índice de **`pAry`** que pasa a ser la dirección actual, y su valor sale a consola, si no se sale fuera de límites. Esta técnica se usa mayormente en ciclos que manejan arreglos 2D, donde usualmente **`i`** y **`j`** son valores variables, **`col`** es el número fijo de columnas a ver en consola y los ciclos están bien controlados.

a. Sea un puntero a un vector. Este programa muestra la forma de pasarlo a una función:

Programa auxiliar a2: pase de un puntero a una función

```
#include <iostream>
#include <cstdlib>
using namespace std;
```

```
const int N = 5;
void mostrar(int *pVect);

int main() {
  int vec[N];       // un vector de 5 columnas
  int *pVec = vec; // un puntero al vector
  //-------------------------------------------------
  //  int *pVec = &vec[0]; // también puede ser así
  //-------------------------------------------------
  cout << "Programa auxiliar 2:\nPase de un puntero a una funci\242n\n\n";
  mostrar(pVec);
  system("pause");
  return EXIT_SUCCESS;
}

void mostrar(int *pVect) {
  cout << "Vector:\n";

  for ( int i = 0; i < N; ++i ) {
    pVect[i] = (i + 1);
    cout << pVect[i] << ", ";
  } // for

  cout << "\b\b \n\n";
}
```

Y la salida es:

```
Programa auxiliar 2:
Pase de un puntero a una función

Vector:
1, 2, 3, 4, 5
```

b. Sea un puntero a una tabla. Este programa muestra la forma de pasarlo a una función: el nombre de un arreglo es tratado como la dirección de su primer elemento; entonces el parámetro es un puntero a **`int`**, pero en este caso, a un arreglo bidimensional y se vería así: **`int tabla[FILS][COLS]`** Como la tabla no se ha especificado antes de la llamada a la función, el compilador no sabe el tamaño de sus filas y poniendo **`tabla[][COLS]`** da un error más o menos así:

```
…main.cpp error: storage size of 'tabla' isn't known
```

que dice en español algo así como **`error: el tamaño del almacenaje de 'tabla' es desconocido`**, por lo que hay que declaralo completo. **`int *pTabla = tabla[FILS]`** declara un puntero al primer elemento de la tabla y lo trata como un vector plano, un valor a continuación del otro. Dentro de la función se acceden a los valores deseados a través de los índices **`i, j`** que recorren normalmente las filas y columnas.

Programa auxiliar a3: pase de una tabla a una función

```
#include <iostream>
#include <cstdlib>
using namespace std;

const int FILS = 2;
const int COLS = 5;

void mostrar(int *pTabla);
```

```
int main() {
  int tabla[FILS][COLS];      // una tabla de FILS x COLS
  int *pTabla = tabla[FILS]; // un puntero al 1er. elemento de la tabla
  cout << "Programa auxiliar 3: pase de una tabla\npor puntero a una funci\242n.\n\n";
  mostrar(pTabla);
  system("pause");
  return EXIT_SUCCESS;
}

void mostrar(int *pTabla) {
  cout << "Tabla de " << FILS << " x " << COLS << ":\n";

  for ( int i = 0; i < FILS; ++i ) {
    for ( int j = 0; j < COLS; ++j ) {
      pTabla[i] = (i + 1) * (j + 1);
      cout << pTabla[i] << "  ";
    } // for j

    cout << '\n';
  } // for i

  cout << '\n';
}
```

Y la salida es:

```
Programa auxiliar 3: pase de una tabla
por puntero a una función.

Tabla de 2 x 5:
1  2  3  4  5
2  4  6  8  10
```

Caso 2: suma en variables simples

Si se aplica esta operación a una variable común, al avanzar el puntero, no se sabe qué valor le sigue y el resultado, cualquiera que sea, carece de significación, lo cual es mostrado en la figura que sigue. No obstante, en el caso de las estructuras autoreferenciadas residentes en la tienda gratis, al avanzar el puntero, éste sí que apunta correctamente aun cuando ese valor no sea contiguo, porque ya está registrado en la estructura.

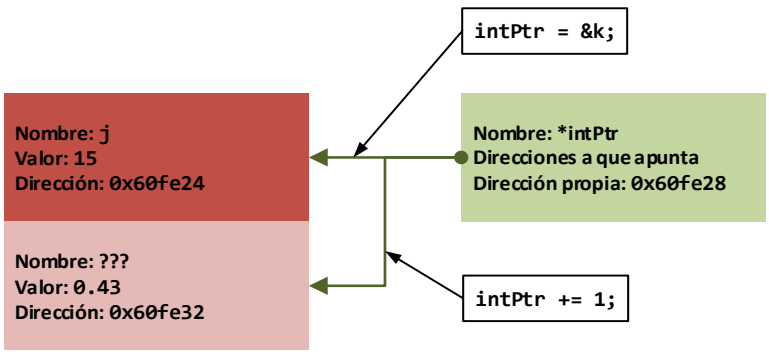


Ilustración 9. Mala aritmética de punteros

Resumiendo: la aritmética de punteros sólo tiene sentido cuando se efectúa sobre arreglos, porque por diseño sus elementos están escritos en memoria uno a continuación del otro.

Resta entre punteros del mismo arreglo

Esta operación devuelve un entero que marca la zancada (distancia) entre ambos valores. Las direcciones de memoria se incrementan desde cero en adelante por lo que, si se invierten los términos el resultado es negativo. Sea el siguiente fragmento de código:

```
int intAry[] = { 3, 6, 1, 9, 0, 2 };
int *pInt_A = &intAry[0]; // equivale a *pInt_A = intAry;
int *pInt_B = &intAry[4]; // apunta al 5to. elemento
cout   << "Enteros: la diferencia entre el 5to. item y "
          "el 1ro. es de " << pInt_B – pInt_A << " lugares.\n";
```

El resultado de 4 lugares está dado por la diferencia de desplazamiento entre los ítems 1ro. y 5to (índices 0 y 4) del arreglo. Si el arreglo fuera de **`float`** o de **`char`**, la diferencia sería la misma. Sea ahora el siguiente fragmento de código:

```
int k = 25, *pInt = &k;
float m = 2.5, *pFlt = &m;
cout << "La diferencia entre un entero y un real es de "
     << pInt - pFlt << " lugares.\n\n"; // línea ofensiva
```

al tratar de compilar, se comete el error:

```
...main.cpp: invalid operands of types 'int*' and 'float*' to binary 'operator-'
```

En español: *operandos no válidos de tipos '`float*`' e '`int*`' [enviados] al operador binario de resta '`operator-`'*

La resta sólo tiene sentido entre punteros del mismo arreglo. Si los punteros pertenecen a arreglos diferentes que son del mismo tipo, el resultado es irrelevante y si son de distintos tipos, hay error de compilación.

Comparación entre punteros

Esta operación busca posiciones relativas entre punteros que apuntan al mismo elemento. También se hace la comparación contra el puntero nulo, que como ya se sabe es compatible con cualquiera. Históricamente existen dos de estas comparaciones que dependen del valor actual del puntero y se realizan con frecuencia:

```
if (p) {...}     // igualdad del puntero p con cero
if (not p) {...} // desigualdad del puntero p con cero
```

aunque se prefiere estas otras, alineadas con el Zen de Python: *la legibilidad cuenta*; y *frente a la ambigüedad, rechaza la tentación de adivinar*.

```
if (nullptr == p) {...}     // igualdad del puntero p con cero
if (p not_eq nullptr) {...} // desigualdad del puntero p con cero
```

Manejo de excepciones

Hasta el momento se han manejado algunas técnicas defensivas tolerantes, intolerantes y las que en la línea de la OOP no son ni lo uno, ni lo otro. ¿Qué tal si el propio programa en tiempo de ejecución es quien decide su técnica correctora?

Idealmente, al desarrollar un algoritmo, el programador lo expresa de forma general y le incluye casos especiales, junto con la estrategia de adecuada, que pudiera tomar cualquier forma y aparecer en cualquier lugar, de modo que el algoritmo

en sí se enfoque en la solución, y las acciones para manejar las situaciones de excepción vayan en cualquier otro lugar. Eso es parte de la técnica de programación modular.

- ✓ Excepción: es un evento que no ocurre comúnmente durante la ejecución de un programa. Aunque tiene probabilidad de aparecer, ésta será baja, pues de otro modo sería un evento común, y debería ser procesado normalmente. Casi siempre es el resultado de un error aparecido en tiempo de ejecución.

La defensa se logra mediante la inspección a excepciones. Estas proveen un método para reaccionar frente a los errores en tiempo de ejecución, transfiriendo automáticamente el control del flujo del programa hacia funciones especiales llamadas manejadores de excepciones (*exception handlers* en inglés.)

El sistema de C++ que soporta la excepciones trabaja por completo ligado a dos clases especiales, pero puede emplearse desde la perspectiva del súper C, sin el uso de la OOP o de la biblioteca estándar de plantillas.

La técnica de excepciones aplicada a la programación modular se basa en el uso de tres palabras claves: **`try`** (trata), **`catch`** (recoge) y **`throw`** (tira.)

- **`try-catch`** es un bloque[29] especial que trata de proteger al código que lleva dentro e identifica la excepción prevista:

  ```
  try {...[trow]...} catch (T1) {...} [catch (T2) {...} ...];
  ```

- **`throw`** lanza una excepción cuando y donde surgen la o las acciones ofensivas previstas. Es el sustituto de **`assert`** o del tratamiento con **`if`**. Es una especie de **`goto`**, pero muy controlado pues, aunque se lanza desde cualquier parte del código, sólo llega hasta el primer **`catch`** que responda al valor de su parámetro **`T`** (una cadena, un número, etc.)
- **`catch`** es un manejador que opera sobre la excepción prevista por **`try`**. Parecida en su sintaxis a una función, indica el tipo de excepción al cual responde y en su cuerpo van las acciones a tomar, del tipo que sean.

Las instrucciones **`try-catch`** forman un bloque lógico y van juntas, pero lo interesante es que **`throw`** puede ejecutarse desde cualquier parte del código, ya sea directamente o emitido por una función u otro objeto, y cualquier excepción será manejada por una de las instrucciones **`catch`**, que le siguen inmediatamente escritas a **`try`**. Si se lanza una excepción, el bloque **`try-catch`** asume las acciones posteriores, ¡pero atención! la instrucción **`catch`** no es llamada explícitamente por código, antes bien, el control del programa salta automáticamente hacia la instrucción **`catch`** adecuada. El salto está gobernado por el tipo de excepción que lanza **`throw`**.

La infraestructura de C++ que maneja las excepciones permite separar al algoritmo de las situaciones excepcionales que pudieran surgir, posibilitando un código más modular, más claro de escribir, más fácil de depurar durante el desarrollo y más cómodo de mantener a lo largo de la vida útil del programa.

Para interceptar la excepción en tiempo de ejecución, usualmente se realizan dos pasos:

1. En un bloque **`try/catch`** se protege el código con potencial ofensivo, teniendo presente declarar los elementos adecuados fuera del bloque, siempre que se pueda, pues al salir, toda variable allí declarada, finaliza su tiempo de vida.
2. Se escribe la excepción **`throw`** donde y cuando pudiera ocurrir.

Este mecanismo permite el salto desde la parte del código que lanza la excepción hasta donde está el bloque que la intercepta. En C++ las dos diferencias importantísimas del **`throw`** frente al **`goto`** son:

- El salto del **`goto`** está limitado al módulo donde aparece, pero **`throw`** puede aparecer en cualquier parte del código.
- El **`goto`** saltará a una etiqueta que va dentro del módulo que le llama, pero **`throw`** saltará al bloque interceptor que le reciba la excepción lanzada, dondequiera que esté.

[29] Es un verdadero bloque –si se declara algo dentro, al salir de ámbito, lo declarado se pierde. Cuidado.

Punteros a funciones

Las funciones también poseen sus propias direcciones, que simplemente son sus nombres. La gran mayoría de las veces, una función llama a otra dentro de su código para que la ayude a resolver su tarea. En este caso se dice que la función llamada está encadenada a la que le llama. Pero hay ocasiones en que se desea aplicar de una vez diferentes acciones sobre un conjunto heterogéneo de datos.

Puede hacerse de dos maneras:

1. Un puntero a la función que va a realizar las acciones en diferentes momentos de la ejecución del programa. Puesto en un ciclo se dice que va a visitar los datos. Esta forma abre el paso al verdadero polimorfismo.
2. Una función lo va a hacer mediante otras que asumen esas acciones, pero tienen sus firmas iguales, lo cual es polimorfismo de funciones en tiempo de compilación. Esta función que usará otras se considera de orden superior y puesta en un ciclo se dice que va a visitar los datos. Entonces, se pasan las direcciones (nombres) de las otras como parámetros y se dice que van apuntadas.

   Esta función de orden superior puede llamar como si fuera un parámetro más a cualquier función conforme con la firma del prototipo desde distintas partes de su propio código, en diferentes momentos de la ejecución del programa, o frente a una combinación de ambas situaciones. Esta forma da flexibilidad a la apoyatura de varias funciones respecto a la de orden superior estando en la línea del Zen de Python: *disperso es mejor que denso*.

Hay dos formas de manejar una función: por encadenamiento y por apuntamiento, y de éstas hay dos formas de hacerlo, por puntero y por pase de parámetros. El encadenamiento y el apuntamiento tienen distintos objetivos, muchas veces nítidamente delimitados. Al programador se le dan las herramientas, pero es su responsabilidad el escogerlas adecuadamente y saber usarlas en el momento oportuno.

Puntero a función

Tal como el nombre de un arreglo se comporta como un puntero constante al primer elemento del conjunto, el nombre de una función también es como un puntero constante a la función. Se puede declarar un puntero a una función, e invocar la función usando ese puntero. Esto permite crear programas que deciden cual función a invocar basados en la entrada del usuario.

El problema sobre los punteros a función es entender el tipo de objeto que está siendo señalado. Un puntero a un **`int`** deberá apuntar a una variable entera y un puntero a una función debe apuntar a una función con el tipo apropiado de regreso y la firma correcta. Por ej., en la declaración:

```
double (*funcPtr)(int x);
```

`funcPtr` es un puntero (va un **`*`** delante del nombre) que apunta una función cualquiera que tome un entero como parámetro y devuelva un número de doble precisión. Los paréntesis alrededor de **`*funcPtr`** son necesarios porque los paréntesis alrededor de **`int`** se enlazan más fuertemente, ya que tienen precedencia sobre el operador de indirección (**`*`**).

Sin los paréntesis **`doublé * funcPtr(int x)`** declara una función que toma un entero y devuelve un puntero a un número de doble precisión.

La declaración de un puntero a función siempre incluirá el tipo de regreso y los paréntesis indicando el tipo de los parámetros, si los lleva.

Tabla 26. Precedencia de los operadores

Grupo	Asociación	Expresión

Primarios (la mayor precedencia)	Izquierda a derecha	`literal`
		`this(expr)`
		`nombre`
		`::nombre`
		`clase-o-namespace::nombre`

Postfijo	Izquierda a derecha	`puntero[expr]`
		`expr`
		`(expr, ...)`
		`type(expr, ...)`
		`objecto`
		`puntero->miembro`
		`cast_keyword<tipo >(expr)`
		`typeid(expr)`
		`typeid(tipo)`
		`lvalor++`
		`lvalor--`

Unarios	Derecha a izquierda	`++lvalor`
		`--lvalor`
		`~expr`
		`compl expr`
		`! expr`
		`not expr`
		`+ expr`
		`- expr`
		`*puntero`
		`&lvalor`
		`sizeof expr`
		`sizeof(tipo)`
		`new-expr`
		`delete-expr`

Encasillamiento ANSI C	Derecha a izquierda	`(tipo)expr`
Puntero a Miembro	Izquierda a derecha	`objeto.*expr` `puntero->*expr`

Multiplicativos	Izquierda a derecha	`expr * expr`
		`expr / expr`
		`expr % expr`

Aditivos	Izquierda a derecha	`expr + expr` `expr - expr`
De corrimiento	Izquierda a derecha	`expr << expr` `expr >> expr`

Relacionales	Izquierda a derecha	`expr < expr`
		`expr > expr`
		`expr <= expr`
		`expr >= expr`

De igualdad	Izquierda a derecha	`expr == expr`
		`expr != expr`
		`expr not_eq expr`

Y bit-a-bit	Izquierda a derecha	`expr & expr` `expr bitand expr`
O Exclusivo bit-a-bit	Izquierda a derecha	`expr ^ expr` `expr xor expr`
O Inclusivo bit-a-bit	Izquierda a derecha	`expr | expr` `expr bitor expr`
Y Lógico	Izquierda a derecha	`expr && expr` `expr and expr`

O Lógico	Izquierda a derecha	`expr || expr` `expr or expr`
Condicional	Derecha a izquierda	`expr ? expr : expr`

De asignación	Derecha a izquierda	`lvalue = expr`
		`lvalue op= expr`
		`throw expr`
		`throw`

Coma **(la menor precedencia)**	Izquierda a derecha	`expr, expr`

Programa 21-puntero a funciones

Codificar un programa que, dados dos números, mediante un puntero a funciones les aplique las cuatro operaciones aritméticas básicas: suma, resta, multiplicación y división. Proponer un ciclo que pregunte por dos valores y ponga en consola la salida de sus cuatro resultados. La división por cero no está definida en aritmética, por lo tanto, para evitar un fallo catastrófico en ese caso, deberá aplicarse programación defensiva.

Declaración

```
#ifndef ARITMETICA_H_INCLUDED
#define ARITMETICA_H_INCLUDED

#include <string>
const std::string ERR("No puedo dividir por cero.\n");

//  las cuatro operaciones aritméticas básicas; x, y son los parámetros
//  de las funciones y todas ellas llevan los mismos parámetros, en la
//  misma posición.

//  entrada: dos números reales
//  salida: el resultado según la función aplicada
double mas  ( double x, double y );
double menos( double x, double y );
double por  ( double x, double y );
double entre( double x, double y );

#endif // ARITMETICA_H_INCLUDED
```

Definición

```
#include "aritmetica.h"

//  las cuatro funciones aritméticas
double mas  ( double x, double y ) { return x + y; }
double menos( double x, double y ) { return x - y; }
double por  ( double x, double y ) { return x * y; }
double entre( double x, double y ) {
  if ( 0 == y ) {
    throw ERR;
  } else {
    return x/y;
  } // if-else
}
```

Programa

```
#include "aritmetica.h"
#include <iostream>
#include <cstdlib>
#include <iomanip>

const std::string PROMPT( "\nEntre dos n\243meros separados\npor un espacio en blanco: " );

// formato: precisión y ancho
const int PRES = 3;
const int WIDE = 6;

int main() {
  //  información:
  std::cout << "Programa 21: punteros a funciones.\n\n"
  "Calcula las cuatro operaciones\naritm\202ticas b\240sicas.\n";

  // variables
  double (*pFunc)(double x, double y); // puntero a función
  double n1, n2;                       // los dos números
  std::string sn;                      // ¿una vez más?

  std::cout << PROMPT << std::fixed << std::setprecision(PRES);

  while ( std::cin >> n1 >> n2 ) {
    pFunc = mas;
    std::cout << "\nx + y = " << std::setw(WIDE) << pFunc(n1, n2);
    pFunc = menos;
    std::cout << "\nx - y = " << std::setw(WIDE) << pFunc(n1, n2);
    pFunc = por;
    std::cout << "\nx * y = " << std::setw(WIDE) << pFunc(n1, n2);

    //  abre el bloque de excepción try-catch
    try {
      pFunc = entre;
      std::cout << "\nx / y = " << std::setw(WIDE) << pFunc(n1, n2);
    } catch (std::string) {
      std::cerr << '\n' << ERR;
    } // try-catch

    std::cout << "\n\250Una vez m\240s? (s/n)> ";
    std::cin >> sn;

    if ( 'N' == toupper(sn[0]) ) {
      return EXIT_SUCCESS;
    } // if-else

    std::cout << PROMPT;
  } // while cin

  system("pause");
  return EXIT_SUCCESS;
}
```

Salida

```
Programa 21: punteros a funciones.

Calcula las cuatro operaciones
aritméticas básicas.

Entre dos números separados
por un espacio en blanco: 12 7

x + y = 19.000
x - y =  5.000
x * y = 84.000
x / y =  1.714
¿Una vez más? (s/n)> s

Entre dos números separados
por un espacio en blanco: 12 0

x + y = 12.000
x - y = 12.000
x * y =  0.000
x / y =
No puedo dividir por cero.

¿Una vez más? (s/n)> n
```

Funciones como parámetro

Una función se pasa como parámetro apuntando su dirección (que simplemente es su nombre) a otra función, considerada de orden superior, para que aquella la invoque desde su propio código. Por ejemplo, sea la función **`double suma(double x, double y) { return x + y; }`** a la cual se le aplican estos cuatro pasos:

1. Se obtiene la dirección de la función —simplemente es su nombre: **`suma`**
2. Se declara una función de orden superior que tome como parámetro dos números (**`n1`** y **`n2`**) y un cálculo sobre dichos números (**`calc()`**):

   ```
   float calcular(float calc(float n1, float n2), float n1, float n2);
   ```

3. A la función de orden superior se le pasa la que va a realizar el cálculo y sus números, respetando el orden de declaración de los parámetros:

   ```
   calcular(suma, n1, n2);
   ```

4. La función de orden superior devuelve el resultado del cálculo en algún lugar dentro su código:

   ```
   float calcular(float calc(float n1, float n2), float n1, float n2) {
       ⋮             ⋮
       return calc(n1, n2); // devuelve el resultado de la función
   }
   ```

Ahora, la función de orden superior puede llevar los parámetros en cualquier orden, aunque al llamarla hay que respetarlo, y cualquier función con la misma firma puede ser invocada por la que llama, desde cualquier parte del programa. Este es el modelo básico del pase de una función como parámetro: la función pasada es invocada dentro de la que le llamó.

Programa 22: funciones como parámetro

Codificar un programa con una función de orden superior que, dados dos números y usando ella misma una función de cálculo como parámetro, les aplique las cuatro operaciones aritméticas básicas: suma, resta, multiplicación y división. Proponer un ciclo que pregunte por dos valores y ponga en consola la salida de sus cuatro resultados. La división por cero no está definida en aritmética, por lo tanto, para evitar un fallo catastrófico en ese caso, deberá aplicarse programación defensiva.

Declaración

```
#ifndef ARITMETICA_H_INCLUDED
#define ARITMETICA_H_INCLUDED

#include <string>
const std::string ERR("\nNo puedo dividir por cero.\n");

//  las cuatro operaciones aritméticas básicas; x, y son los parámetros
//  de las funciones; todas ellas llevan los mismos parámetros, en la
//  misma posición.

//  entrada: dos números reales
//  salida: el resultado según la función aplicada
double mas  ( double x, double y );
double menos( double x, double y );
double por  ( double x, double y );
double entre( double x, double y );

//  Este es el modelo de uso de una función por puntero: en la interfaz
//  van la función a apuntar y los parámetros que necesita, en cualquier
//  orden, pero se trata de poner algo que sea fácilmente legible para
//  un ser humano.

//  input: la función apuntada y los números que necesita
//  output: el cálculo que realiza la función
double calcula(double calc(double x, double y), double x, double y);

#endif // ARITMETICA_H_INCLUDED
```

Definición

```
#include "aritmetica.h"
#include <string>

//  las cuatro funciones aritméticas:
//  la resta se calcula usando la suma;
//  la división se calcula usando la multiplicación
double mas  ( double x, double y ) { return x + y; }
double menos( double x, double y ) { return mas(x, -y); }
double por  ( double x, double y ) { return x * y; }
double entre( double x, double y ) {
  if ( 0 == y ) {
    throw ERR;
  } else {
    return por(x, 1/y);
  } // if-else
}
```

```
//  modelo básico de uso de una función pasada como parámetro:
//  en este caso sólo se devuelve el resultado de la función
double calcula(double calc(double x, double y), double x, double y) {
  return calc(x, y);
}
```

Programa

```
#include "aritmetica.h"
#include <iostream>
#include <cstdlib>
#include <iomanip>

const std::string PROMPT( "\nEntre dos n\243meros separados\npor un espacio en blanco: " );

// formato: precisión y ancho
const int PRES = 3;
const int WIDE = 6;

int main() {
  //  información:
  std::cout << "Programa 22: funciones como parámetros.\n\n"
  "Calcula las cuatro operaciones\naritm\202ticas b\240sicas.\n";

  // variables
  double n1, n2;  // los dos números
  std::string sn; // ¿una vez más?
  std::cout << PROMPT << std::fixed << std::setprecision(PRES);

  // este es el modelo de uso en el programa
  while ( std::cin >> n1 >> n2 ) {
    std::cout <<
    "\nx + y = " << std::setw(WIDE) << calcula(mas, n1, n2)  <<
    "\nx - y = " << std::setw(WIDE) << calcula(menos, n1, n2) <<
    "\nx * y = " << std::setw(WIDE) << calcula(por, n1, n2);

    //  abre el bloque de excepción try-catch
    try {
      std::cout << "\nx / y = " << std::setw(WIDE) << calcula(entre, n1, n2) << '\n';
    } catch (std::string) {
      std::cerr << ERR;
    } // try-catch

    std::cout << "\250Una vez m\240s? (s/n)> ";
    std::cin >> sn;

    if ( 'N' == toupper(sn[0]) ) {
      return EXIT_SUCCESS;
    } else {
      std::cout << PROMPT;
    } // if-else
  } // while cin
}
```

Salida

Exceptuando el encabezamiento, con los mismos datos la salida es la misma que en el programa anterior, sin embargo, hay sutiles diferencias: en esta forma hay más flexibilidad; en la otra –aunque por ahora no se vea– hay más alcance, más profundidad.

QUINTO BLOQUE DE EJERCICIOS

En la solución de estos ejercicios puede usarse todo lo mostrado hasta aquí, pero el objetivo aplicado directamente sobre el trabajo con punteros es hacer ganar confianza, crear código eficaz y desarrollar eficiencia, *por lo que ambos deben ser manejados a través de ellos.* El atento lector al analizarlos se da cuenta de que la función principal en ambos (**`mostrar(notas);`** y **`mostrar(notas, ALUMNOS);`** respectivamente) no están bien formadas: no son ni cohesionadas ni pequeñas. Además, como son problemas de cierta complejidad, se puede pensar en estructuras, aunque todos pueden resolverse con lo dicho hasta ahora en el texto.

27. Tomar el programa dado en la página 104 y montar su arreglo en punteros. Hacer sus funciones bien formadas. Menos el encabezado, la salida deberá ser igual. Recordar que un puntero sólo contiene direcciones.
28. Tomar el programa dado en la página 107 y montar su arreglo en punteros. Hacer sus funciones bien formadas. Menos el encabezado, la salida deberá ser igual. Estudiar detenidamente el problema del arreglo bidimensional y su pase como parámetro.

13 - MANEJO DINÁMICO DE LA MEMORIA

> *La arquitectura Von Neumann o arquitectura Princeton, es un diseño para una computadora digital electrónica constituida por una unidad de procesamiento central (CPU), una unidad de control (UC), memoria para almacenar datos e instrucciones, almacenamiento masivo externo y mecanismos de entrada y salida. Editado de (Wikipedia en español)*

Este es el modelo más común entre las computadoras personales y se le conoce así por su autor, el húngaro John von Neumann. Como se ve en la tabla que sigue, en la pila el programa controla las llamadas de funciones. En la tienda gratis hay mucha memoria que el programador puede aprovechar "gratuitamente" usando punteros y técnicas dinámicas de acceso. Cuando el SO ejecuta un programa, le otorga una cantidad de su propia memoria como espacio de trabajo y según indica (Sutter), el compilador almacena sus datos e instrucciones en dicha memoria, segmentada en varias partes.

Tabla 27. Áreas de la memoria en C++: características y tiempos de vida de sus objetos

Área	Características y tiempos de vida de sus objetos
Constantes	El área de las constantes guarda las cadenas literales y otros datos cuyos valores se conocen en tiempo de compilación; todos son de lectura solamente, y toda esa área está disponible durante la vida del programa.
La pila	En la pila se guardan las variables automáticas. Variables de todo tipo son construidas en su punto de definición y destruidas inmediatamente que salen de ámbito. Su trabajo es mucho más rápido que el almacenamiento dinámico. Comparte espacio físico con la tienda gratis/el bulto y crece hacia esa área.
La tienda gratis	Esta área está a disposición del programa cuando las variables se construyen con el operador **`new`**. Comparte el mismo espacio físico del bulto y crece hacia la pila, pero su tratamiento no es de igual forma y no deben mezclarse en un mismo programa, o se desincronizará el trabajo de la pila, con resultados catastróficos.
El bulto	Esta área está a disposición del programa cuando las variables se construyen con las familias de operadores **`malloc()`**. Comparte el mismo espacio físico de la tienda gratis, pero su tratamiento no es de igual forma y no deben mezclarse en un mismo programa, o se desincronizará el trabajo de la pila, con resultados catastróficos.

Espacio global de nombres	Esta área guarda las variables y constantes que se declaran al comienzo de la ejecución del programa, aunque pudieran no ser inicializados en ese instante. Por ejemplo, una variable estática declarada dentro de una función será inicializada por primera y única vez cuando la ejecución del programa le traiga en ámbito.

La figura muestra que ambas secciones de memoria comparten el mismo espacio físico, pero en la medida en que se llaman bloques y funciones, la pila crece en dirección a la tienda gratis. En la medida en que se reserva memoria, la **`free store`** crece en dirección a la pila. Si en un momento de la ejecución los dos segmentos se encuentran, aparece un error fatal de falta de memoria y el programa colapsa. Actualmente hay mucha memoria disponible y este tipo de error, que antaño no fuera desconocido, hogaño es difícil que aparezca.

Ilustración 10. Manejo de la Memoria

Resumiendo:

- En ANSI C, el montón de memoria se llama bulto (*heap* en inglés) y se administra con las funciones de la familia **`malloc`**. ANSI C no acepta a **`new`** o **`delete`**, porque no "conoce" las clases.
- En ISO C++ el montón de memoria se llama tienda gratis y se administra con los operadores **`new`** y **`delete`**, quienes llaman correctamente a los constructores y destructores.

Si bien C++ acepta la herencia de **`malloc`** y compañía, no las gestiona bien, puesto que ANSI C no conoce de las clases y como sus funciones no llaman ni a constructores ni a destructores, pueden aparecer resultados imprevistos y efectos colaterales indeseados en un programa que las use. En C++ no se mezclan los operadores del bulto con los de la tienda gratis para reservar memoria, so pena de que se destruya la sincronía del trabajo en la pila, llevando al programa a un fallo irrecuperable.

El operador new

Para pedir memoria a la tienda gratis se usa el operador **`new`**, seguido por el tipo de variable u objeto para el cual se desea reservar, indicando al compilador el tamaño requerido de la memoria. Por ejemplo, sea complejo una clase; entonces:

```
Complejo *ptrZ  = new Complejo;
```

reserva suficiente memoria en la tienda gratis para albergar correctamente toda la clase. **`new`** devuelve una dirección de memoria que es asignada al puntero, por lo tanto, la reserva de memoria con new siempre es a un puntero.

Comprobación del pedido de memoria

Puede ser que el pedido falle, porque después de todo, la memoria es un recurso, aunque ya no es tan escaso como lo fue en los primeros tiempos. En los programas que se usan de ejemplo, o en los que se proponen como ejercicios, las exigencias de memoria son siempre cumplidas debido a lo necesariamente pequeño de sus tamaños y la actual abundancia de memoria en cualquier PC, por lo que, aunque más abajo se explica el mecanismo, en casi todos los ejemplos del texto se ignora la comprobación en aras del espacio.

En programas profesionales no es el caso. Simplemente, se puede agotar la memoria durante un pedido y ahí llega el desastre, por lo tanto, en ese ámbito nunca se debe dar por hecho que **`new`** trabajó correctamente: el cuidadoso lector debe comprobarlo.

Al pedir memoria con **`new`**, siempre debería controlarse si se otorgó

En las primeras versiones de C++ se indicaba el fallo de **new** devolviendo un puntero nulo, pero ahora se muestra lanzando una excepción **bad_alloc**, y meramente aborta la ejecución. Una forma QAD de evadir este comportamiento es pasarle a **new** el parámetro **std::nothrow**, para que elimine la excepción y retorne un puntero nulo.

```
Item *pItem = new(std::nothrow) Item;
if ( nullptr == pItem ) {
   std::cerr << "\nError: memoria agotada. . .\n";
        ⋮      // más código defensivo aquí, etc.
} // if
```

Desde luego, el programador decide cuál será la acción a tomar después del fallo. En este ejemplo, la acción correctora consistió en un mensaje explicativo y más código. Aún hay otra forma que es más académica y más profesional, consistente en proteger todo el código que tiene que ver con la petición de memoria:

```
try {
   Item * pItem = new Item;  // si falla se lanza automáticamente una excepción bad_alloc
catch (const exception& e) {
   cout << "Excepción: " << e.what() << '\n';
} // try-catch
```

El operador delete

Al terminar la tarea que requirió el uso de la tienda gratis, debe desactivarse el puntero, aplicándole el operador delete. La memoria se libera, pero el puntero, si se declaró local a una función o método, se pierde al salir fuera de ámbito. Como C++ no posee recolector de basura[30], la memoria reservada ya no puede ser accedida y aparece el fenómeno de fuga de memoria (*memory leak* en inglés), consistente en "robarle" memoria al programa: ahora el programa tiene a su disposición menos que cuando empezó y esta no se recobrará hasta que el programa termine su ejecución.

El operador **delete** devuelve la memoria tomada de la tienda libre, pero no borra al puntero, el cual, si tiene tiempo de vida puede ser reasignado para cumplir otra tarea.

Liberación de memoria

Al pedir memoria de la tienda gratis, se devuelve un puntero al cual es obligatorio seguirla la pista, pues si se pierde aparece una fuga de memoria. Al pasar al puntero entre funciones, "alguien" se adueñará de él. Lo más usual es que sea pasado por referencias y la función que lo obtuvo sea la que lo elimina. Pero es lo que se acostumbra; no es un elemento sintáctico ni mucho menos una camisa de fuerza.

Sin embargo, usar una función que reclame memoria y otra que la libere es al menos ambiguo, fuente segura de ofuscación y puede dar origen a uno de dos errores que son altamente peligrosos: u olvidar borrar el puntero o intentar bórralo dos veces.

Se debe considere el pedir memoria en una función que luego pase al puntero por referencias y al final, sea ella misma quien lo libere. Esto pone la gestión de memoria en un lugar único y evita ese par de errores.

Si se libera memoria con **delete**, su dirección regresa a la tienda gratis para ser empleada de nuevo si hace falta, pero si el puntero es global o local al **main()**, puede reservar memoria una y otra vez, cuidando de no crear fugas. Lo mejor es

[30] Un recolector de basura (*garbage collector* en inglés) es un mecanismo implícito de gestión de memoria implementado en muchos lenguajes de programación. Cuando el algoritmo lo decide, compacta la memoria disponible para datos del programa en cuestión, destruyendo automáticamente punteros ociosos. Por ser muy poco eficiente, ISO C++ no lo usa.

eliminarle la vieja dirección que aun contiene, para evitar que quede colgando, anulando al puntero inmediatamente después de liberar la memoria. El ciclo es reserva-liberación-anulación, o **new-delete-nullptr**.

```
// punteros para arreglos
<T> *pAry = new Ary[n];
   . . .// más instrucciones aquí
delete [] pAry;
pAry = nullptr;
```

```
// punteros para lo demás
<T> *pColl = new <T>;
   . . .// más instrucciones aquí
delete pColl;
pColl = nullptr;
```

Ilustración 11. Administrando la tienda gratis

Vea cómo se administra la memoria en la tienda gratis. A la izquierda **T** es el especificador del tipo del arreglo para el que se asigna memoria (**int**, **float**, etc.), **ptrAry** es su puntero, **new** es el operador que asigna al puntero suficiente memoria de la tienda gratis como para contener al arreglo y **delete []** es el operador que devuelve la memoria cuando ya no se necesita.

A la derecha se muestra cuando no es un puntero. **T** es el especificador del tipo del objeto para el que se asigna memoria (**int**, **float**, una clase, etc.), **ptrColl** es su puntero, **new** es el operador que asigna al puntero suficiente memoria de la tienda gratis como para contener a dicho objeto y **delete** es el operador que devuelve la memoria cuando ya no se necesita.

Resumiendo:

- Si se reserva memoria sin corchetes, se libera sin corchetes.
- Si se reserva memoria con corchetes, se libera con corchetes, o queda fuga de memoria.
- No se libera la misma memoria dos veces seguidas: es un error de ejecución.
- Una vez liberada la memoria, inmediatamente el puntero debería ponerse a nulo, pero es responsabilidad exclusiva del programador seguir el ciclo **new-delete-nullptr**.

¿Se puede ignorar la memoria dinámica?

Si claro. Pero se ignora aparecen dos problemas de peso en un programa grande, serio: por un lado, las variables locales son volátiles y cuando la función termina, son descartadas; por el otro, las variables globales son persistentes, pero permiten acceso sin restricciones a todo lo largo de la ejecución, lo cual degrada tanto la ocultación de información, como el principio del privilegio mínimo, y favorece la creación de código ofuscado.

Poniendo los datos en la tienda gratis se solucionan ambos problemas. Se puede pensar que la tienda gratis es como la sección de apartados de una oficina de correos: miles de apartados numerados, a la espera de cargar su correspondencia. Pero hasta ahí la analogía. La memoria no puede ser accedida directamente: hay que pedir una de sus direcciones y luego pasarla a un puntero; es todo. El puntero permite el acceso a los datos, escondiendo la mecánica implementada.

Es de saber que en C++:

- Sólo las funciones que manejan a un puntero tienen acceso a los datos a que él apunta. Esto provee una interfaz controlada y elimina el problema de otra variable cualquiera cambiando esa información indiscriminadamente.
- La pila es vaciada automáticamente cuando una función acaba. Todas las variables locales y los valores de sus parámetros salen de ámbito y son removidos de la pila.
- Si un puntero apunta a otra dirección sin haber liberado la primera, esa memoria ya no se puede acceder, no se puede usar y se produce una fuga de memoria. Si hay muchas fugas, el recurso se agota y entonces el programa colapsa,

aunque actualmente, con la potencia y capacidad que entregan los equipos más humildes, este fenómeno es bastante raro, pero si ocurre es indicio de que el programador no está adecuadamente preparado para esa tarea.
- La tienda gratis no es vaciada hasta que el programa acaba o le reclaman la memoria reservada.
- El puntero declarado fuera de la tienda gratis permanece disponible hasta que termina el programa, presentando el peligro de convertirse en un puntero colgante. Se le puede apuntar a memoria nuevamente reclamada una y otra vez, pero nunca de seguido: el ciclo es *reserva-liberación-anulación*. Si se viola, lo más probable es que ocurra un error de ejecución, pero ¡atención! si se libera la misma dirección de memoria dos veces seguidas, sí que ocurre dicho error.

Los dos últimos puntos son de la total responsabilidad del programador

---

Cualquier programador debe saber cómo crear un puntero para un área de la tienda gratis y pasarlo entre funciones y C++ proporciona un método conveniente y seguro para administrar la memoria de la **free store**: utilizando **new** para reclamarla, y liberándola posteriormente con **delete**.

El puntero creado estáticamente dentro de un ámbito global, estará disponible hasta el fin del programa. Si también es creado estáticamente, pero dentro de una función, sólo será válido hasta el fin de ésta. Pero si es creado en la tienda gratis, la memoria a que apuntaba podrá ser devuelta con **delete**, a conveniencia del programador.

Al administrar memoria dinámica en código nuevo, hacerlo siempre con **new** y **delete**

¿Puede apuntarse a una variable simple?

Sin dudas. Por ejemplo, si se declara una variable entera, se le reclama memoria, se inicializa y luego es invocada:

```
int x = 4;
int *pInt = new int;
pInt = &x;
cout << "Entero: x = " << *pInt;
```

la salida será: **Entero: x = 4**. Finalmente se libera la memoria con **delete pInt;**

¿Puede apuntarse a una estructura?

Por supuesto. Por ejemplo, si se declara la estructura

```
struct FltStruct {
   float x;
};
```

y se reclama memoria para esa variable, se inicializa y luego se invoca:

```
FltStruct *pStruct = new FltStruct;
pStruct->x = 3.14; // como es un puntero se usa el operador ->
cout << "Estructura: x = " << pStruct->x << "\n";
```

la salida será: **Estructura: x = 3.14**. Finalmente se libera la memoria con **delete pStruct;**

¿Puede apuntarse a un arreglo?

Claro. Sin temores. Por ejemplo, si se declara un arreglo de 3 cadenas, se le reclama memoria, se le inicializa un elemento (digamos, el 2do.) y luego se invoca:

```
string *pStr = new string[3];
*(pStr + 1) = "Cadena #2";
```

```
cout << "Arreglo: valor = " << *(pStr + 1) << "\n";
```

la salida será: **`Arreglo: valor = Cadena #2`**. Finalmente se libera la memoria con **`delete [] pStr;`**

Debido al fuerte acoplamiento entre punteros y vectores, C++ "ve" el nombre del vector como si fuera un apuntador *r/o* a su elemento cero ya que **`miArray`** y **`miArray[0]`** comparten la misma dirección en memoria. La porción de código anterior pudiera haberse escrito también así, con una sintaxis más clara:

```
string *pStr = new string[3];
pStr[1] = "Cadena #2";
cout << "Arreglo: valor = " << pStr[1] << "\n";
```

resultando la misma salida. E igual sucede con un arreglo numérico: sus valores también pueden ser accedidos por el operador de compensación **`[]`**. Pero el lector recordará que un arreglo estático —declarado fuera de la tienda gratis— no puede ser borrado con **`delete`**; su tiempo de vida dependerá de su ubicación dentro del programa.

¿Puede apuntarse a otro puntero?

Sí, desde luego y siempre que sean compatibles (ambos del mismo tipo, o uno de ellos **`void`**), un puntero puede apuntar al otro sin más. Si son **`void`**, las operaciones de E/S con ellos deben ser debidamente encasilladas. Pero en este texto introductorio, ¿para qué agregar más de un nivel de indirección?

Los niveles extras de indirección son muy empleados en ANSI C, donde el uso de punteros es primordial en ciertos tipos de aplicaciones, y hay que saber manejarlos muy bien, pero C++ hace un buen trabajo escondiendo esos detalles y son pocas las ocasiones en que hace falta lidiar con ellos manualmente.

¿Por qué punteros?

Si se puede trabajar directamente con variables, entonces ¿por qué usar punteros (declarados en el programa o pedidos a la tienda gratis) con esas dificultades de comprensión y empleo que conllevan?

La respuesta, aunque múltiple es sencilla: en primer lugar, porque hay muchísimo código ANSI C lleno de punteros ahí afuera, que el curioso lector pudiera encontrarse algún día, siendo obligado a interpretarlo; en segundo lugar, porque hay funciones de la biblioteca estándar de C++ que requieren de punteros como parámetros; en tercer lugar, porque unos pocos elementos de la biblioteca estándar de plantillas de C++ (STL: *Standard Template Library*, en inglés y en lo adelante) también requieren de punteros como parámetros; y en un muy sumamente importante cuarto lugar, porque sin punteros no hay acceso a la tienda gratis y su montón de memoria.

La buena nueva es que la STL oculta y maneja casi todo el trabajo con punteros, como se verá en el segundo tomo de esta serie.

Correspondencia entre punteros y arreglos

Hay veces en que resulta conveniente reservar memoria en tiempo de ejecución usando una función de reservación de memoria. Usar este método permite posponer la decisión del tamaño del bloque de memoria necesario, por ejemplo, para guardar un arreglo hasta el tiempo de ejecución. O permitirnos usar una sección de la memoria para guardar un arreglo de enteros en un tiempo determinado y posteriormente, cuando esa memoria no sea necesaria, liberarla para otros usos... (Jensen)

He aquí la declaración de tres arreglos:

```
1.  int  elem1[500];              // arreglo "normal" de 500 enteros
2.  int *elem2[500];              // arreglo de 500 punteros a enteros
3.  int *elem3 = new elem3[500]; // puntero a un arreglo de 500 enteros en la tienda gratis
```

El lector tratará de comprender correctamente ese código antes de proseguir, porque sus diferencias afectan dramáticamente cómo se opera con ellos.

Arreglos unidimensionales

Seguidamente se verá cómo crear un arreglo en la tienda gratis, y un par de formas de armar allí una tabla. En un arreglo unidimensional no debe haber problemas. Las conversiones son simples y de uso inmediato. Por ejemplo, sea un vector de 5 enteros. Podría escribirse código de la siguiente manera:

Programa auxiliar a4: un vector en la tienda gratis

```
#include <iostream>
#include <cstdlib>

using namespace std;
const int N = 5;

void mostrar(int *vect);

int main() {
  cout << "Programa auxiliar 4: muestra un\n"
      "vector de la tienda libre.\n\n";
  int * pVect = new int[N];  // se reserva memoria
  mostrar(pVect);
  delete [] pVect;           //se libera memoria
  system("pause");
  return EXIT_SUCCESS;
}

void mostrar(int *vect) {
  cout << "Vector de " << N << " columnas:\n";

  for ( int i = 0; i < N; ++i ) {
    vect[i] = (i + 1);
    cout << vect[i] << ", ";
  } // for

  cout << "\b\b \n\n";
}
```

Y la salida es:

```
Programa auxiliar 4: muestra un
vector de la tienda libre.

Vector de 5 columnas:
1, 2, 3, 4, 5
```

Arreglos bidimensionales

Los arreglos bidimensionales (multidimensionales por extensión) son una abstracción que esconde la verdadera mecánica de acceso a los datos, como muestra la ilustración en la página 106. Realmente, todos pudieran ser declarados y tratados como arreglos simples, con la diferencia que la falta de abstracción obliga al programador a calcular la verdadera posición de cada dato, delegándole más responsabilidad.

Así pues, hay varias maneras de manejar las tablas que están en la tienda gratis y EODA, ninguna es bonita (*bello es mejor que feo*). Se muestran dos de ellas y se deja su aplicación a gusto del consumidor. El primer caso, más abstracto[31], cambia flexibilidad por facilidad de codificación y menos responsabilidad; el segundo, hace el cambio al revés: codifica a nivel más bajo, con más responsabilidad, pero ganando en flexibilidad. El programador decidirá en un momento dado cuál de los dos le gusta más, o le conviene más.

Estas soluciones de compromiso se ven muy a menudo en el diseño de aplicaciones reales

Manejo por índices

Primero se declara un puntero normal a un vector lineal de tamaño fijo y luego se declara dicho puntero en la tienda gratis, a un vector con filas variables, pero con columnas necesariamente fijas, del tamaño antes declarado, perdiendo flexibilidad. En cambio, se delega al compilador la responsabilidad del cálculo de las posiciones, ganando en abstracción. La declaración debe ser:

```
int (*pTabla)[COLS];
```
[32]

y dicho puntero reserva una secuencia bidimensional en la tienda gratis:

```
pTabla = new int[filas][COLS]; // filas es dado, COLS es de tamaño fijo
```

Programa auxiliar a5: manejo de una tabla por índices.

```
#include <iostream>
#include <cstdlib>

using namespace std;
const int COLS = 5; // son constantes

void mostrar(int pTabla[][COLS], int filas);

int main() {
  cout << "Programa auxiliar 5: manejo\n"
       "de una tabla por \241ndices.\n\n";
  int filas = 2;                  // pueden ser pedidas al usuario
  int (*pTabla)[COLS];            // un puntero normal a las columnas
  pTabla = new int[filas][COLS]; // se apunta a una tabla en la tienda
  mostrar(pTabla, filas);
  delete [] pTabla; // libera memoria
  system("pause");
  return EXIT_SUCCESS;
}
```

[31] Aquí el término *abstracto* es una medida cualitativa del trabajo que pasa el codificador al escribir código: a más abstracción, menos código a escribir y menos probabilidad de errar. ¡La abstracción es buena!

[32] Si se pone **`int *pTabla[COLS]`** se declara un vector de **`COLS`** punteros a enteros. En el programa auxiliar, el paréntesis preserva la precedencia ya que el operador asterisco tiene menor primacía que el operador corchetes.

```
void mostrar(int pTabla[][COLS], int filas) {
  cout << "Tabla de " << filas << " x " << COLS << ":\n";

  for ( int i = 0; i < filas; ++i ) {
    for ( int j = 0; j < COLS; ++j ) {
      pTabla[i][j] = (i + 1) * (j + 1);
      cout << pTabla[i][j] << "  ";
    } // for j
    cout << "\n";
  } // for i

  cout << "\n";
}
```

Y la salida es:

```
Programa auxiliar 5: manejo
de una tabla por índices.

Tabla de 2 x 5:
1  2  3  4  5
2  4  6  8  10
```

Manejo por cálculo de posiciones

Se declara la tabla como un puntero a un vector simple, ganando flexibilidad, ya que filas y columnas se pueden determinar en tiempo de ejecución, pero a cambio el codificador tiene que calcular los índices, dispensando al compilador de esa responsabilidad, perdiendo en abstracción[33]. Tener presente la forma de construcción del índice del arreglo.

Programa auxiliar a6: manejo de una tabla por posiciones

```
#include <cstdlib>
#include <iostream>

using namespace std;
void mostrar(int *pTab, int f, int c);

int main() {
  cout << "Programa auxiliar 6: manejo de una\ntabla por c\240lculo de posiciones.\n\n";
  int filas = 2;                          // pueden ser pedidas al usuario
  int columnas = 5;                       // pueden ser pedidas al usuario
  int *pTabla = new int[filas * columnas]; // apunta a una secuencia en la tienda
  mostrar(pTabla, filas, columnas);
  delete [] pTabla;
  system("pause");
  return EXIT_SUCCESS;
}

void mostrar(int *pTab, int f, int c) {
  cout << "Tabla de " << f << " x " << c << ":\n";

  for ( int i = 0; i < f; ++i ) {
```

[33] También aparecen ocasiones en que este tipo de codificación es de carácter obligatorio, por ello es bueno aprenderla.

```
    for ( int j = 0; j < c; ++j ) {
      pTab[i*c + j] =  (i + 1) * (j + 1);
      cout << pTab[i*c + j] << "  ";
    } // for j
    cout << "\n";
  } // for i

  cout << "\n";
}
```

Y la salida desde luego es:

```
Programa auxiliar 6: manejo de una
tabla por cálculo de posiciones.

Tabla de 2 x 5:
1  2  3  4  5
2  4  6  8  10
```

Nota: este concepto del manejo de arreglos tabulados por índices no es exclusivo de la memoria dinámica; puede aplicarse a los arreglos estáticos. De hecho, los compiladores modernos, al crear código de máquina, llevan cualquier forma de codificación del índice de un arreglo a esta última.

Punteros y la tienda gratis

El programa #18 decía: En la cafetería El Jardín se ofertan a la población tres productos cuyos precios son los siguientes: refresco a $1.00, dulce rollito a $1.20, y bola de helado a $0.55. Al informático ubicado en ese establecimiento se le ha dado la tarea de elaborar un programa que, mediante un menú, permita mostrar las ofertas (nombre y precio de cada producto), pueda entrar la venta del día y determinar la recaudación total.

La solución del problema usando memoria dinámica se da en el programa 23a-punteros y la tienda libre. El análisis es el mismo dado en en la página 117: ¡exteriormente nada cambia, excepto la información del programa principal y el empleo de la tienda gratis! Y es lo que se refleja aquí:

- En **main()**, la declaración de la variable (el objeto) invoca el operador **new**:

```
// estructura C++ en la free store
Cafeteria *pJardin = new Cafeteria (
  "Refresco", 1.00,
  "Helado",   0.55,
  "Rollito",  1.20,
  "El Jard\241n"
);
```

- En **main()**, para llamar un método cualquiera se emplea el operador flecha, porque trabaja con un puntero:

```
pJardin->mostrarMenu();
```

La ganancia estriba en que los datos están en una sección de memoria muy extensa. Ahora el programa puede crecer para tornarse profesional, sin problemas de espacio y sin esfuerzo por parte del programador. Y con los mismos datos, las posibles salidas (con encabezamiento diferente) son iguales a las del programa 18.

Justificación del número complejo

En una muy breve explicación, el concepto de número complejo es una extensión de los números reales y puede representarse como la suma simbólica de un número real y un número imaginario, múltiplo real de la unidad imaginaria, indicada con la letra i. El primer elemento se define como parte real y se denota como $a = reZ$; el segundo elemento se define como parte imaginaria y se expresa como $b = imZ$. Un número complejo z en forma algebraica se escribe como un par ordenado de números reales: $z = (a, b)$ y en forma binomial como una suma simbólica: $z = a + bi$ Ambas representan al mismo número complejo z en formatos diferentes.

Sean dos números complejos $z_1 = (a, b)$ y $z_2 = (c, d)$ Entonces se definen las operaciones siguientes como:

Conjugado: $\sim z_1 = (a, -b)$
Igualdad: $z_1 == z_2 \rightarrow (a == c \text{ y } b == d)$
Suma: $z_1 + z_2 = (a + c, b + d)$
Resta: $z_1 - z_2 = (a - c, b - d)$

Una clase ADT para el número complejo

La solidez de un ADT se basa en que la implementación está escondida al usuario y sólo la interfaz es pública. Esto significa que la implementación del ADT puede cambiar con el tiempo o los requerimientos del usuario, pero mientras sea consistente con la interfaz, no se afectan los programas que lo usan. Editado a partir de (Wikipedia en español)

Ahora se verá por qué el ADT se considera como cimiento de la OOP. Al aumentar las demandas de los usuarios frente a una aplicación comercial dada, se comprobó que el escalamiento funciona bien hasta cierto nivel cuya estimación exacta es muy puntual y depende de múltiples factores, pero que ciertamente es real, está presente y obedece mayormente al grado de dificultad que posee el programa original.

En aquel momento (allá por los inicios del ‘75) se buscó una línea de trabajo mucho más sistemática, fácil y barata, consistente en definir un tipo de dato mediante la unión lógica de sus propiedades básicas y el conjunto de operaciones que soportaba, modelando el comportamiento de los tipos nativos. Por ejemplo, el tipo **`int`** de C++ acepta números enteros en cierto rango y las reglas sintácticas estipulan lo que se puede y no se puede hacer con él. Esto aportó un grado mayor de abstracción a la solución del problema, permitiendo reducir la complejidad inherente.

Según (Prata, C Primer Plus, p. 787), y este autor concuerda, el método más probado para definir un ADT consta de 3 pasos: **(1)** diseñar una descripción general del tipo de dato junto con sus operaciones: este paso no va atado a ningún lenguaje en específico y delinea el principio de ocultación de información; **(2)** desarrollar la interfaz, compuesta por el tipo de dato que sea y las operaciones que se le realizarán: este paso ya puede ser definido en seudocódigo lo cual sirve para el diseño en múltiples lenguajes, lo que se conoce como encapsulación de datos y métodos; y **(3)** codificar la interfaz y esto ya es específico.

El asunto es que, aunque en teoría el usuario sólo debería trabajar el dato a través de sus interfaces, los seres humanos no somos muy disciplinados que digamos. En la práctica se violaron muchas reglas, y en aquél entonces, no había forma física de encapsular el ADT: su agrupamiento era lógico, y la encapsulación aplicada de esta forma es débil. Y en eso llegó la OOP y su concepto de clase, donde la agrupación entre el dato y sus operaciones es física, las interfaces son muy breves y tersas, el encapsulado es fuerte porque forma parte del tipo de dato, y el escalamiento es muchísimo más administrable.

Por otro lado, la STL suministra numerosos recursos para viabilizar el desempeño del humilde codificador, cuyos conocimientos requeridos para un trabajo dado ya no les son forzados a obtenerlos como un profesional graduado de los cursos de Ciencias de la Computación, y se acercan cada vez más a los de un técnico, eso sí, altamente calificado. Por ejemplo, para manejar una lista, un vector, o un número complejo no se requiere conocer de las clases y fabricarlas una a una; sólo se necesita saber usar correctamente la STL, incluso a un nivel elemental.

Reconociendo esa verdad, EODA hay saberes que se deben dominar como cimiento firme desde donde se está en pie. Es por eso que en el texto se presentan ejemplos elementales y cortos de vectores, tablas, listas, números complejos y bases de datos.

Seguidamente se muestra una sencilla estructura C++ sustituyendo un ADT para números complejos, de los cuales existe una gran cantidad de textos para su estudio y aplicación. Las matemáticas hace tiempo los definieron; se parte de ahí, y se simplifica y acorta bastante, pues es un ejemplo que será usado en el texto con fines puramente didácticos y no laborales. Para el ejemplo sólo se definen las operaciones de conjugado, igualdad, suma y resta, y presumir que el tipo de dato es **`double`**.

Aunque C++ posee capacidades para hacer aritmética compleja, este es un ejemplo de "juguete" para mostrar un par de cosas: se presenta una de las muchas formas de hacer entrada formateada en que el usuario entrará sus datos en formato de par ordenado y el programa dará la respuesta en formato binomial; y se trabajará con punteros a la tienda gratis. Una comprobación QAD consiste en probar que el cuadrado de la unidad imaginaria es calculado correctamente como –1

Se advierte al lector no ceder a la tentación durante su trabajo real, de hacer uno de calidad profesional por sí mismo. La biblioteca estándar tiene uno facturado por los mejores programadores en la industria y avalado por años de aplicación sin fallos ¡Es el que se debe usar! De nuevo el autor aconseja firmemente: *No trate de reinventar la rueda, primero vea si alguien ya lo hizo y úselo o adáptelo.*

Al encarar un problema real, se debe reusar código: primero, tomado tanto de la biblioteca estándar como de la STL y luego, de bibliotecas comerciales suministradas por fabricantes del IDE, o por terceras personas de su confianza, de libros de texto o de consulta, de sitios serios en la Internet, etc. El resultado será posiblemente más rápido de obtener, más fácil de escribir y mucho más seguro. Y no olvidarse de agradecer si es el caso…

Programa 23: una ADT

Crear un ADT para los números complejos, diseñando una estructura adecuada al problema, ponerla en la free store y codificar un programa que pida dos números complejos z1 y z2 (en notación algebraica) y los muestre en notación binomial y polar, así como a sus conjugados y las operaciones de igualdad, suma y resta.

Declaración

```
#ifndef COMPLEJO_H_INCLUDED
#define COMPLEJO_H_INCLUDED

#include <string>
typedef std::string Str;

const double RAD2GRD = 57.295779;

struct Complejo {
    Complejo ( double a = 0, double b = 0 ); //  constructor
    //  métodos
    bool esIgualQue   ( Complejo z ) const; // igualdad
    Complejo sumadoCon( Complejo z ) const; // adición
    Complejo restadoDe( Complejo z ) const; // sustracción
    Complejo conjugado() const;             // su conjugado
    void fmtCoord( Str id ) const;          // muestra (a,b)
    void fmtPolar( Str id );                // muestra (mod,sita)
    void fmtSuma( Str id );                 // muestra z = a+bi
    void entraNumero( Str id );             // entra un número (a,b)
    //  miembros
    double real; // parte real
```

```
    double imag; // parte imaginaria
};

#endif // COMPLEJO_H_INCLUDED
```

Definición

```
#include "complejo.h"
#include <iomanip>
#include <iostream>
#include <cmath>
#include <ios>

using std::cout;
using std::cin;

//  constructor
Complejo::Complejo(double a, double b) {
    real = a;
    imag = b;
}

//  comprueba si dos complejos son iguales
bool Complejo::esIgualQue (Complejo z) const {
    return real == z.real and imag == z.imag; // a1 == a2 y b1 == b2
}

//  suma dos complejos
Complejo Complejo::sumadoCon(Complejo z) const {
    double a = real + z.real; // a = r + rz
    double b = imag + z.imag; // b = i + iz
    return Complejo(a, b);
}

//  resta un complejo de otro
Complejo Complejo::restadoDe(Complejo z) const {
    double a = real - z.real; // a = r - rz
    double b = imag - z.imag; // b = i - iz
    return Complejo(a, b);
}

//  conjugado
Complejo Complejo::conjugado() const {
    return Complejo(real, -imag);
}

//  entra un número complejo
void Complejo::entraNumero(Str id) {
    cout << id;
    cin.ignore(1, '(') >> real; // ignora '('
    cin.ignore(1, ',') >> imag; // ignora ','

    // en el búfer de entrada quedan restos: eliminar hasta
    // el último carácter contenido en el búfer
    while ( cin.get() not_eq '\n' ) {
        continue;
    } // while
}

// muestra un número complejo en formato coordenado
void Complejo::fmtCoord( Str id ) const {
```

```
    std::ios::fmtflags oldFlags = cout.flags();
    cout << id << std::setprecision(2) << std::fixed
         << "(" << real << "," << imag << ")\n"
         << std::setprecision(6);
    cout.flags(oldFlags);
}

// muestra un número complejo en formato de suma
void Complejo::fmtSuma(Str id) {
    std::ios::fmtflags oldFlags = cout.flags();
    cout << id << std::setprecision(2) << std::fixed << real
         << std::showpos << imag << "i\n" << std::setprecision(6);
    cout.flags(oldFlags);
}

// muestra un número complejo en formato polar
void Complejo::fmtPolar( Str id ) {
    double sita = atan2( imag, real ) * RAD2GRD;
    double modulo = sqrt( pow( real, 2.0 ) + pow( imag, 2.0 ) );
    std::ios::fmtflags oldFlags = cout.flags();
    cout << id << std::setprecision(2) << std::fixed
         << "(" << modulo << "," << sita << ")\n"
         << std::setprecision(6);
    cout.flags(oldFlags);
}
```

Programa

```
// ****************************************************
// Estipulaciones del ADT Complejo
// Nombre: Complejo
// Por: S. Galliano
// Fecha: mayo 13/2020
// Propiedades: Puede hacer algo de aritmética compleja
// Operaciones:
//      determina si dos complejos son iguales,
//      calcula el conjugado de un complejo,
//      suma y resta dos complejos.
// ****************************************************
#include "complejo.h"
#include <iostream>
#include <cstdlib>

int main() {
    //  información
    std::cout << "Programa 23: aritm\202tica compleja.\n\n";

    //  pedido de memoria
    Complejo  *ptrZ  = new Complejo,
              *ptrZ1 = new Complejo,
              *ptrZ2 = new Complejo;

    //  entrada
    std::cout << "Entrar datos en formato (a,b)\n";
    ptrZ1->entraNumero("Complejo z1 = "); // 1er. complejo
    ptrZ2->entraNumero("Complejo z2 = "); // 2do. complejo

    //  salida:
    std::cout << "\nEn formato (Mod,Grad)\n";
    ptrZ1->fmtPolar("z1 = ");
    ptrZ2->fmtPolar("z2 = ");
    std::cout << "\nEn formato z = a+bi\n";
```

```
    ptrZ1->fmtSuma("z1 = ");
    ptrZ2->fmtSuma("z2 = ");
    std::cout << '\n';

    //  operaciones:
    std::cout << "Conjugados:\n";
    *ptrZ = ptrZ1->conjugado(); ptrZ->fmtCoord("~Z1 = ");
    *ptrZ = ptrZ2->conjugado(); ptrZ->fmtCoord("~Z2 = ");
    std::cout << "\nIgualdad:\n";
    ( ptrZ1->esIgualQue(*ptrZ2) )
        ? std::cout << "z1 y z2 son iguales.\n\n"
        : std::cout << "z1 y z2 no son iguales.\n\n";

    std::cout << "Algunas operaciones aritm\202ticas:\n";
    *ptrZ = ptrZ1->sumadoCon(*ptrZ2); ptrZ->fmtCoord("(z1+z2) = ");
    *ptrZ = ptrZ1->restadoDe(*ptrZ2); ptrZ->fmtCoord("(z1-z2) = ");
    std::cout << '\n';

    //  fin
    delete ptrZ;
    delete ptrZ1;
    delete ptrZ2;
    system("pause");
    return EXIT_SUCCESS;
}
```

Salida

```
Programa 23: aritmética compleja.

Entrar datos en formato (a,b)
Complejo z1 = (9.8,6.5)
Complejo z2 = (5.4,2.1)

En formato (Mod,Grad)
z1 = (11.76,33.55)
z2 = (5.79,21.25)

En formato z = a+bi
z1 = 9.80+6.50i
z2 = 5.40+2.10i

Conjugados:
~Z1 = (9.80,-6.50)
~Z2 = (5.40,-2.10)

Igualdad:
z1 y z2 no son iguales.

Algunas operaciones aritméticas:
(z1+z2) = (15.20,8.60)
(z1-z2) = (4.40,4.40)
```

Notas:

- La petición de memoria puede ir encadenada; la liberación de memoria se hace uno a uno.
- En el programa se pide **`*ptrZ = ptrZ1->sumadoCon(*ptrZ2);`**

- **`*ptrZ`** = porque **`ptrZ`** llama al método, el cual devuelve un puntero.
- **`ptrZ1->`** porque se accede al método de suma de **`Z1`** mediante un puntero.
- **`sumadoCon(*ptrZ2)`** porque se envía el valor del puntero **`Z2`** al método que invoca **`Z1`**, el cual los suma, y devuelve el valor a **`Z`**, que fue quien llamó.

- En el módulo se codifica con devolución directa del complejo:

```
//  suma dos complejos
Complejo Complejo::sumadoCon(Complejo z) const {
  double a = reZ + z.reZ; // a = r + rz
  double b = imZ + z.imZ; // b = i + iz
  return Complejo(a, b);
}
```

Donde:

- Los valores llamados directamente son propios de la estructura que llama.
- Los valores llamados con punto son propios de la estructura llamada.

SEXTO BLOQUE DE EJERCICIOS

En la solución de estos cinco ejercicios puede usarse todo lo mostrado hasta aquí, pero el objetivo principal es cimentar la confianza adquirida en crear código eficaz y eficiente manejando punteros en la tienda gratis, *por lo que deben ser resueltos con punteros dinámicos.*

29. Tomar el programa 28 sobre las notas de tres estudiantes, dado en la página 147, y montar su arreglo en punteros dinámicos. En el tema hay todo lo que pueda necesitar. Menos el encabezado, la salida será igual.
30. ¿Se puede graficar en modo consola? La producción de electricidad a nivel mundial se calcula mediante seis acápites que, según (Wikipedia en español), para el año 2012 fueron: Turba (40.4%), Gas natural (22.5%), Hidráulica (16.2%), Nuclear (10.9%), Petróleo (5%) y Renovable (5%). Diseñar una estructura adecuada al problema, montarla en la tienda gratis y codificar un programa que brinde la tabla y su gráfico de barras. La salida sería algo así:

```
Tabla. Producción 2012
----------------------
1  Turba          40.4
2  Gas natural    22.5
3  Hidráulica     16.2
4  Nuclear        10.9
5  Petróleo        5.0
6  Renovable       5.0

Gráfica. Producción 2012
            |    5   10   15   20   25   30   35   40 %
          --+----+----+----+----+----+----+----+----+--->
       Turba |****************************************
 Gas natural |************************
  Hidráulica |****************
     Nuclear |***********
    Petróleo |*****
   Renovable |*****
```

31. En el programa 23 sobre la ADT, resuelto en el texto, se trabajó una versión muy simplificada del número complejo. Adaptarlo añadiendo las operaciones de producto, cociente y aritmética con un escalar E que afecta al complejo, todas ellas definidas como:

- Producto: $z_1 \times z_2 = (ac - bd, ad + bc)$
- Cociente: $\frac{z_1}{z_2} = \frac{(a,b)}{(c,d)} = \left(\frac{ac+bd}{c^2+d^2}, \frac{bc-ad}{c^2+d^2}\right)$

- Operaciones aritméticas de $z(a,b)$ con un escalar E $\begin{cases} z(a,b) \pm E = (a \pm E, b) \\ z(a,b) \times E = (a \times E, b \times E) \\ z(a,b) \div E = (a \div E, b \div E) \end{cases}$
- Norma: $z = \sqrt{z^2}$

La salida debiera verse más o menos así:

```
Problema 31: aritmética compleja.

Entrar datos en formato (a,b)
Complejo z1 = (9.8,6.5)
Complejo z2 = (5.4,2.1)
Escalar = 1.33

En formato z = a+bi
z1 = 9.80+6.50i
z2 = 5.40+2.10i

En formato (Mod,Grad)
z1 = (11.76,33.55)
z2 = (5.79,21.25)

Conjugados:
~Z1 = (9.80,-6.50)
~Z2 = (5.40,-2.10)

Igualdad:
z1 y z2 no son iguales.

Operaciones aritméticas:
(z1+z2) = (15.20,8.60)
(z1-z2) = (4.40,4.40)
(z1*z2) = (39.27,55.68)
(z1/z2) = (1.98,1.66)

Operaciones con un escalar:
(e+z1) = (11.13,6.50)
(e-z1) = (8.00,6.50)
(e*z1) = (13.03,8.64)
(z1/e) = (7.37,4.89)

Los módulos son:
|z1| = 11.76
|z2| = 5.79
```

32. Rehacer el ejercicio 24, bloque 4, sobre el Pañol de la UDG Blas Roca, pero montado en un puntero dinámico. Observar que **(a)** sólo cambia el programa principal, que ahora hace uso de la tienda gratis; **(b)** con los mismos datos, la salida deberá ser igual a la que se da allí, excepto por el encabezado.
33. Para manejar información acerca de las barritas de dulce, la gerencia de las TRD en Manzanillo pidió a su personal de informática crear una base de datos para operar sus productos. Como piloto se pidió una demostrativa (demo) para gestionar las tente-en-pie o barritas de dulce, *snacks* en inglés. El responsable estipuló el manejo de los siguientes datos: la marca de la barrita, el precio de venta en CUC y la cantidad de unidades en inventario:

Marca	Stock	Precio
Mocha Munch	208	0.90

Krispy Krunch	411	1.05
Honey Beez	789	1.25

La demo debe estar montada en un puntero dinámico, ser capaz de permitir cambiar el precio de venta a una marca y su cantidad de calorías, y añadir una nueva, o borrar cualquiera de las que están en existencia. Además, debe ser posible pedir más unidades de cualquier marca en inventario y saber cuántas unidades hay en el almacén. El gerente podrá entonces manejar la comercialización mediante un menú apropiado.

Notas de diseño:

- Debido a lo limitado de la importación, informática adopta el montaje sobre un arreglo de estructuras con capacidad para 5 marcas. Es de notar que esta solución QAD es simple y efectiva, pero no escala bien, pues si aparecen más marcas que las esperadas, hay que recodificar el programa.
- Se decide manejar la demo por marca.
- Al comenzar, se inician manualmente las tres barritas y el resto de arreglo se anula.
- Al borrar un elemento se deja un "hueco" poniendo la marca de la barrita a nula.
- Al adicionar un elemento hay ponerlo en el primer "hueco" que aparezca.

Es casi seguro que cada acción a tomar desde el programa principal —**`main()`**— sea más o menos larga y/o compleja. Si el lector piensa que es el caso, puede usar un par **`fichero.h/fichero.cpp`** para manejar las complejidades, o si prefiere, puede integrar ese manejo a la estructura del almacén o quizás, llamar las acciones directamente. Como quiera.

La salida debiera ser algo así:

```
Problema 33. Una base de datos.

1) Imprimir inventario
2) Adicionar un producto
3) Eliminar un producto
4) Editar un precio
5) Editar un stock
6) Vaciar el inventario
7) Ayuda
0) Terminar
Seleccione una opción: 1

En almacén:
=======================================
Marca           Precio  Stock   Costo
---------------------------------------
Mocha Munch      0.90     208  187.20
Krispy Krunch    1.05     411  431.55
Honey Beez       1.25     789  986.25
Total.........................1605.00$
=======================================

En stock hay 3 productos.
```

14 – LISTAS SIMPLEMENTE ENLAZADAS

Una lista enlazada…la constituye una colección lineal de elementos llamados nodos, donde el orden de los mismos se establece mediante punteros. Cada nodo se divide en dos partes: una primera que contiene la información asociada al

> *elemento y una segunda parte, llamada campo de enlace…que contiene la dirección del siguiente nodo de la lista. (Lipschutz, pág. 131)*

Se recuerda al lector que una estructura no puede contener *un miembro de su mismo tipo*, aunque perfectamente si puede contener *un puntero a su mismo tipo*, lo cual permite la definición, declaración y empleo de una estructura avanzada de datos llamada lista, en la cual cada elemento contiene información para mantener la lista operativa: es lo que se denomina una estructura autoreferenciada.

En general, las listas representan un conjunto de estructuras autoreferenciadas avanzadas compuestas por nodos enlazados unos con otros. Entre ellas están las listas lineales o circulares, que pueden ser simple o doblemente enlazadas. Son tema de estudio para la asignatura Estructura de Datos, impartida en cursos avanzados de Ciencias de la Computación.

- ✓ Nodo: estructura dividida lógicamente en dos secciones: una de datos, que puede ser compleja y/o extensa; y otra de enlace, que puede tener uno o más punteros para referenciar a otro nodo similar.

Si la sección de datos es extensa y/o compleja, es costumbre codificar dos funciones auxiliares que se encarguen de transferirla del nodo para afuera y desde afuera para el nodo, que bien podrían ser:

```
Nodo *inserta(Item k);                      // desde afuera para el nodo
Item extrae(const Nodo *head, Item valor);  // desde el nodo para afuera
```

Las listas genéricas presentan las características siguientes:

- Permiten adicionar o eliminar nodos con facilidad y rapidez**.**
- No tienen tamaño fijo: se agrandan o encogen a demanda de las necesidades del programa.
- Trabajan en estrecha vinculación con la tienda gratis, poniendo ahí sus nodos.

Si el nodo tiene un puntero, la única estructura que se puede construir con él es lineal y secuencial; genéricamente se le llama lista simplemente enlazada o LSE en lo adelante, que es la que se va a estudiar en el texto.

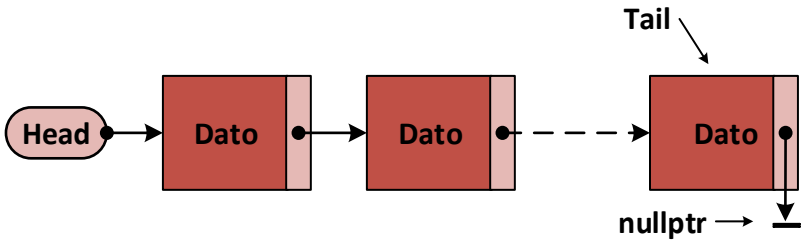


Ilustración 12. Lista genérica linear simplemente enlazada

Por convención universal, al primer nodo de la LSE se le llama **`head`** (cabeza en español), y al último se le llama **`tail`** (cola en español), cuyo enlace siempre es a cero (**`nullptr`**) Cuando el último nodo se enlaza al primero, la lista se dice que es circular simplemente enlazada: LCSE.

EODA las siguientes operaciones básico-auxiliares no deberían faltar en una aplicación que use una lista cualquiera: la que comprueba si está vacía, la que cuenta los nodos presentes, y si la lista es dinámica, la que devuelve la memoria a la tienda gratis.

Tipos de LSE

Las LSE pueden ser de cuatro tipos, pero todas necesitan de un puntero que pueda "viajar" desde su inicio (**`head`**) hasta su final (**`tail`**.) En la mayoría de las veces, este puntero viajero (sea **`p1`**) será acompañado por otro (sea **`p2`**) que le va a la zaga en su avance:

1. Listas simples. Son aquellas donde la inserción y la remoción de un nodo es en el lugar que el programador escoja. Necesitan comparar nodo por nodo contra la posición dada.
2. Listas ordenadas. Son aquellas listas simples donde la inserción del nodo está definida por uno de sus campos, de modo que un nodo siempre entra en el lugar que le corresponde según un orden definido por dicho campo en el problema, por lo que requieren comparar el valor del nodo entrante contra los valores existentes.
3. Pilas o LSE tipo LIFO (*"Last In, First Out*", en español, el último que entra es el primero que sale.) Son aquellas en que la inserción de un nodo siempre es por la cabeza, y esa operación es llamada **`push`** (empujar en español). La remoción de un nodo también siempre es por la cabeza, operación llamada **`pop`**, sonido onomatopéyico que representa el descorchado de una botella. A la cabeza, en vez de **`head`** le llaman **`top`**. Es el tipo de lista más simple, porque el puntero no se mueve: siempre apunta a la cabeza y es el mecanismo de llamada de funciones en C++.
4. Colas o LSE tipo FIFO (*"First In, First Out*", en español, el primero que entra es el primero que sale.) Son aquellas donde la remoción de un nodo siempre es por la cabeza, a la cual también le llaman **`top`**, y a esa operación le llaman **`dequeue`** (desencolar en español) y la adición de un nodo siempre es por el final, denominado **`tail`**, y a esa operación le llaman **`enqueue`** (encolar en español). A veces se diseña con un puntero extra que viaja de la cola a la cabeza.

Crear y mantener manualmente una lista de cualquier tipo requiere un esfuerzo considerable por parte del programador, por lo que modernamente las listas se manejan con la STL. Aquí se aborda un estudio elemental, pero necesario para fijar la ineludible destreza que deberá adquirir el programador de C++ al trabajar con punteros y la tienda gratis.

La más importante y principal virtud de las listas —su capacidad de no tener tamaño fijo— no puede hacerse justicia manualmente. Y aparece un nuevo caso en que no queda otra: hay que usar ciclos.

Recomendaciones del trabajo con listas

Las siguientes dos recomendaciones aparecen en (Marzal & Gracia, pág. 278 & sig.); el autor exhorta al lector a aplicarlas minuciosamente:

1. En el trabajo con listas genéricas hay dos cuestiones especiales que conviene estudiar directamente en todo caso: **(a)** la lista vacía y **(b)** la lista que sólo tiene un nodo.
2. Al trabajar con cualquier lista, se debe seguir sin falta una regla muy sencilla: si se requiere acceder a un nodo para indireccionarlo, siempre hay comprobar que su enlace no sea nulo, porque un puntero nulo apunta a…nada y ahí mismo el programa explota.

Las siguientes asignaciones usan dos punteros auxiliares y comprueban que el segundo, quien va inmediatamente detrás del primero, no es nulo. Si es nulo, está antes de la cabeza y no se le puede acceder.

```
Nodo *p1 = head, *p2 = nullptr;
    ⋮                   // más código aquí
if ( nullptr == p2 ) {
    head = nuevo;       // enlace al inicio de la lista
} else {                // ponerlos en sus lugares
    ⋮                   // más código aquí
} // if-else
```

Las siguientes asignaciones avanzan a los punteros auxiliares al próximo nodo:

```
p2 = p1;       // p2 se pone a la par de p1 y entonces...
p1 = p1->prox; // ...p1 avanza al próximo nodo, mediante su enlace
```

Las LSE en C++

Como las estructuras POD no lo permiten, montar una LSE desde una de ellas conlleva a codificar las acciones fuera de la estructura, tratándola como una ADT, lo cual implica que el usuario debe tener disciplina al usar las acciones y las interfaces tendrán varios parámetros de enlace. Todo ello hace que el código sea más complejo de lo que realmente debiera ser. Pero desde C++ se pueden integrar los métodos a la estructura, enlazando fuertemente métodos y miembros, simplificando mucho las interfaces y dando como resultado un código más sencillo.

Una fuente de ofuscación

En las listas simples se manejan el puntero estático **head** a la cabeza de la lista, el puntero estático **prox** al nodo que le sigue, y el puntero dinámico **nuevo** de un nodo nuevo. Todos son punteros en el sentido que sólo albergan direcciones de memoria, pero los dos primeros son estáticos, residen en el espacio global de nombres y se declaran de forma distinta que el último, que es dinámico y reside en la tienda gratis. Al ser manejados producen efectos diferentes.

¡Un mismo concepto con dos vías de especificación, sintaxis muy similar y efectos diferentes! Eso es ofuscamiento en buen español, y para un principiante el comprender cómo se maneja esta situación no es fácil. El lector debe tomar nota:

- En la estructura se declaran dos nodos estáticos que mantienen la lista:

```
struct Nodo {
            ⋮
    struct Nodo *head; // puntero a la cabeza de la lista
    struct Nodo *prox; // puntero al próximo nodo que entrará
};
```

- Dentro del código los nodos que posibilitan el enlace son estáticos y se declaran así:

```
Nodo *p1 = head;     // viajará a partir de la cabeza
Nodo *p2 = nullptr;[34] // le seguirá inmediatamente detrás
```

- Pero los nodos de la lista son dinámicos, residen en la tienda gratis y se declaran en el código así:

```
Nodo *nuevo = new Nodo; // un nuevo nodo a enlazar a la lista
```

Programa 24: LSE en C++

Se pide manejar por menú, un listado de enteros mediante una LSE. El programa debe ser capaz de crear una lista, insertar un nodo en una posición dada, enumerar los nodos, buscar un valor en la lista, imprimirla, eliminar un nodo con determinado valor y vaciarla y debe dar ayuda sobre sus opciones. Ahora se mostrará un análisis sobre la posible inserción de un valor a la cabeza de la lista, que se refuerza con sencillos diagramas hechos con lápiz y papel. Se muestra el módulo que adiciona un valor, con sus líneas numeradas, para facilitar el análisis:

```
01 void Nodo::adiciona(Item val, short pos) {
02   Nodo *p1 = head, *p2 = nullptr;
03   Nodo *nuevo = new Nodo;
04   nuevo->item = val; // toma el valor
05
06   // pone a p2 en la posición de inserción
07   // y p1 preserva la continuidad
```

[34] Un nodo de enlace puede ponerse a nulo, siempre que otro cualquiera retenga su dirección, para que el vínculo de la estructura no se pierda.

```
08   for (int i = 0; i < pos and p1 not_eq nullptr; i++) {
09     p2 = p1;
10     p1 = p1->prox;
11   } // for
12
13   nuevo->prox = p1;  // se pone en posición
14
15   if ( p2 == nullptr ) {
15     head = nuevo;      // se pone a la cabeza
17   } else {
18     p2->prox = nuevo; // se enlaza a la lista
19   } // if-else
20 }
```

Estudiando la salida se ve que una llamada desde **`main()`** tuvo que ser: **`adiciona(56, 0)`** y para el análisis, se supone que los otros dos valores ya estaban enlistados. La figura muestra la ejecución desde la línea **`02`** hasta la línea **`04`**: **`p1`** apunta a la cabeza de la lista, **`p2`** apunta al valor nulo, hay un puntero en la tienda gratis sin valor y apuntando hacia "basura" y la lista tiene dos nodos, donde el último, anclado a cero, marca el final de la lista.

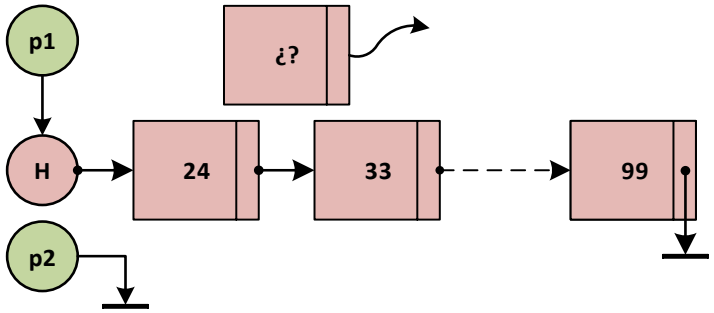


Ilustración 13. Comienza la inserción

Ahora la figura muestra la ejecución desde la línea **`08`** hasta la línea **`13`**: **`p2`** avanzó hasta donde estaba el enlace de **`p1`**, que era en este caso a la cabeza de la lista; entonces **`p1`** avanzó hasta donde apuntaba su propio enlace, que era al primer nodo. La operación se repitió hasta que se llegó a la posición deseada. El nuevo nodo tomó el valor dado.

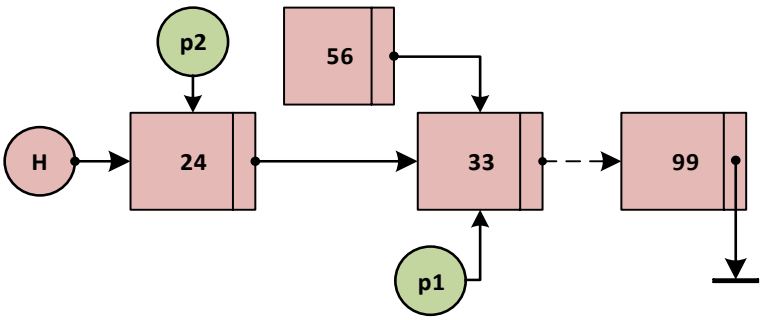


Ilustración 14, Los punteros avanzan

Por último, se ejecuta la condicional que va de la línea **`15`** a la **`19`**: **`p2->prox`** –la dirección del enlace del nodo anterior– se enlaza al nuevo nodo y se mantiene la continuidad. El nodo está enlazado a la lista.

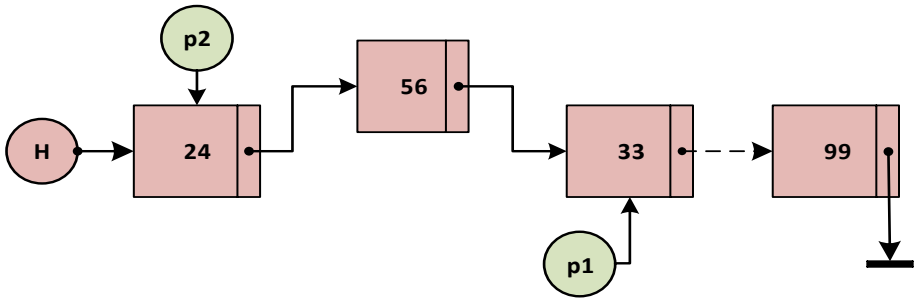


Ilustración 15. El nodo está enlazado

Replicar cuidadosamente el programa y ejecutarlo trabajando todas las variantes del menú. Seguir con lápiz y papel las asignaciones de los nodos, sus creaciones y sus borrados hasta comprender bien la mecánica de trabajo. ¡El autor garantiza que vale la pena!

Declaraciones

- El menú

```
#ifndef MENU_H_INCLUDED
#define MENU_H_INCLUDED

//  enumera lo que pide el menú
enum { terminar, insertar, enumerar, buscar, imprimir, eliminar, vaciar, ayudar };

#include <string>

//  un menú en consola
//  entrada: la información al usuario
//  salida: menú a consola
short menu(std::string info);

//  esclarece las opciones
//  input: nada
//  output: la ayuda en consola
void darAyuda(std::string info);

#endif // MENU_H_INCLUDED
```

- La lista

```
#ifndef LISTA_H_INCLUDED
#define LISTA_H_INCLUDED

#include <string>
const std::string WARN("La lista est\240 vac\241a.\n\n");
const std::string HECHO("\255Hecho!\n\n");
typedef int Item;

struct Nodo {
  // constructor
  Nodo();
  // métodos
  bool estaVacia () const;
  bool enLista   (Item val) const;
  int cuentame   () const;
  void muestrame () const;
```

```
  void adiciona  (Item val, short pos);
  void vaciaLista();
  std::string borraNodo(Item val);
  //  miembros
  Item item;          // valor de un nodo
  struct Nodo *head; // cabeza de la lista
  struct Nodo *prox; // puntero al próximo nodo
};

#endif // LISTA_H_INCLUDED
```

Definiciones

- El menú

```
#include "menu.h"
#include <iostream>
#include <cstdlib>
#include <string>
using std::string;
using std::cout;
using std::cerr;
using std::cin;

//  un menú en consola
short menu(string info) {
  string conjunto = "01234567";
  string opcion;
  string::size_type idx;

  while ( true ) {
    system("cls");
    cout << info <<
    "Lista simplemente enlazada\n"
    "1) Insertar un valor\n"
    "2) Contar los nodos\n"
    "3) Comprobar si un valor est\240\n"
    "4) Mostrar la lista\n"
    "5) Eliminar un valor\n"
    "6) Vaciar la lista\n"
    "7) Ayuda\n"
    "0) Terminar\n"
    "Entre una opci\242n: ";
    (cin >> opcion).get();
    idx = conjunto.find(opcion);

    if ( string::npos == idx ) {
      cerr << "\nError: opci\242n ilegal.\n\n";
      system("pause");
    } else {
      return idx;
    } // if-else
  } // while
}

void darAyuda(string info) {
  system("cls");
  cout << info <<
  "AYUDA DE LA LSE\n"
  "1) Inserta un valor V en la posici\242n P:\n"
```

```
  "     Si P es cero, pone V en la cabeza.\n"
  "     Si P est\240 fuera de l\241mites, pone V en la cola.\n"
  "     Si P est\240 entre l\241mites, pone V en la posici\242n P\n"
  "     Si V est\240, no lo inserta.\n"
  "2) Cuenta la cantidad de nodos en la lista.\n"
  "3) Comprueba si un valor est\240 en la lista.\n"
  "4) Muestra la lista.\n"
  "5) Si el valor est\240, lo elimina.\n"
  "6) Vac\241a la lista.\n"
  "     Si la lista existe, advierte al usuario.\n\n";
}
```

- La lista

```
#include "lse.h"
#include <iostream>
#include <cstdlib>
using std::cout;
typedef std::string Str;

Nodo::Nodo() {
  head = prox = nullptr;
}

bool Nodo::estaVacia() const {
  return nullptr == head;
}

int Nodo::cuentame() const {
  int n = 0;

  for ( Nodo *p1 = head; p1 not_eq nullptr; p1 = p1->prox ) {
    n++;
  } // for

  return n;
}

bool Nodo::enLista(Item val) const {
  for ( Nodo *p1 = head; p1 not_eq nullptr; p1 = p1->prox ) {
    if ( val == p1->item ) {
      return true;
    } // if
  } // for

  return false;
}

Str Nodo::borraNodo(Item val) {
  Str str;

  if ( not enLista( val ) ) {
    str = "El valor no est\240.\n\n";
  } else {
    Nodo *p1 = head;
    Nodo *p2 = nullptr;

    // p1 viaja por la lista y p2 le sigue
    while ( p1->item not_eq val ) {
      p2 = p1;       // pone a p2 sobre el nodo a borrar
```

```
      p1 = p1->prox; // preserva la continuidad
    } // while

    if ( nullptr == p2 ) {
      head = p1->prox; // lo desancla de la cabeza
    } else {
      p2->prox = p1->prox;     // lo desancla de la lista
    } // if-else

    delete p1; // el nodo se elimina de memoria
    str = HECHO;
  } // if-else

  return str;
}

void Nodo::vaciaLista() {
  Nodo *p1;

  while ( not estaVacia() ) {
    if ( head->prox == nullptr ) { //  llegó al final
      delete head;
      head = nullptr;
    } else { //  quedan nodos; sigue borrándolos
      p1 = head;
      head = p1->prox;
      delete p1;
    } // if-else
  } // while
}

void Nodo::adiciona(Item val, short pos) {
  Nodo *p1 = head, *p2 = nullptr;
  Nodo *nuevo = new Nodo;
  nuevo->item = val; // toma el valor

  // pone a p2 en la posición de inserción
  // y p1 preserva la continuidad
  for (int i = 0; i < pos and p1 not_eq nullptr; i++) {
    p2 = p1;
    p1 = p1->prox;
  } // for

  nuevo->prox = p1;  // se pone en posición

  if ( p2 == nullptr ) {
    head = nuevo;      // se pone a la cabeza
  } else {
    p2->prox = nuevo; // se enlaza a la lista
  } // if-else
}

void Nodo::muestrame() const {
  cout << "Lista: ";

  for ( Nodo *p1 = head; p1 not_eq nullptr; p1 = p1->prox ) {
    cout << p1->item << ", ";
  } // for
```

```
    cout << "\b\b \n\n"; // quita la coma final
  }
```

Programa

```
#include "menu.h"
#include "lse.h"
#include <iostream>
#include <cstdlib>
#include <string>

int main() {
  std::string info("Programa 24: manejo de una LSE.\n\n");
  Nodo *head = new Nodo; // la lista en la tienda libre
  short opcion;
  Item valor, pos, n;

  while (true) {
    opcion = menu(info);

    if ( terminar == opcion ) {
      if ( not head->estaVacia() ) {
        head->vaciaLista();
      } // if

      delete head;
      return EXIT_SUCCESS;
    } // if

    switch(opcion) {
    case ayudar:
      darAyuda(info);
      break;
    case insertar:
      std::cout << "\250Valor? ";
      (std::cin >> valor).get();

      if ( head->enLista(valor) ) {
        std::cout << "Ya est\240.\n\n";
      } else {
        std::cout << "(0 a la cabeza) \250Posici\242n? ";
        (std::cin >> pos).get();
        if ( 0 == pos ) {
          head->adiciona(valor, 0);
        } else {
          head->adiciona(valor, --pos);
        } // if-else
        std::cout << HECHO;
      } // if-else
      break;
    case enumerar:
      if ( head->estaVacia() ) {
        std::cout << WARN;
      } else {
        n = head->cuentame();
        ( 1 == n )
        ? std::cout << "Un nodo.\n\n" : std::cout << n << " nodos.\n\n";
      } // if-else
      break;
```

```
    case buscar:
      if ( head->estaVacia() ) {
        std::cout << WARN;
      } else {
        std::cout << "\250Valor? ";
        (std::cin >> valor).get();
        ( head->enLista(valor) )
          ? std::cout << "Ya est\240.\n\n"
          : std::cout << "No est\240.\n\n";
      } // if-else
      break;
    case imprimir:
      if ( head->estaVacia() ) {
        std::cout << WARN;
      } else {
        head->muestrame();
      } // if-else
      break;
    case eliminar:
      if ( head->estaVacia() ) {
        std::cout << WARN;
      } else {
        std::cout << "\250Valor? ";
        (std::cin >> valor).get();
        std::cout << head->borraNodo(valor);
      } // if-else
      break;
    case vaciar:
      if ( head->estaVacia() ) {
        std::cout << WARN;
      } else {
        char procedo;
        std::cout << "Esta operaci\242n destruir\240 la lista.\n"
              "\250Procedo? (s/n) ";
        std::cin >> procedo;

        if ( 'S' == toupper(procedo) ) {
          head->vaciaLista();
          std::cout << HECHO;
        } // if
      } // if-else
      break;
    } // switch

    system("pause");
  } // while
}
```

Una posible salida

```
Programa 24: manejo de una LSE.

Lista simplemente enlazada
1) Insertar un valor
2) Contar los nodos
3) Comprobar si un valor está
4) Mostrar la lista
5) Eliminar un valor
6) Vaciar la lista
7) Ayuda
0) Terminar
Entre una opción: 4
Lista: 24, 56, 33

Presione una tecla para continuar . . .
```

Bien, esa es la idea. *Es fundamental entender totalmente este ejemplo si se desea avanzar adecuadamente en los saberes del C++*. De lo contrario, la base de aprendizaje no estará bien cimentada y el alto nivel de programación que pudiera alcanzar un día el lector, estará seriamente comprometido. EODA, no queda de otra.

SÉPTIMO BLOQUE DE EJERCICIOS

En C++ las listas (y otras estructuras secuenciales) se usan sin reservas mediante la STL, pero EODA, cualquier programador de este lenguaje debería tener una buena idea del manejo manual de estos recursos, como parte de su formación profesional. Esta sección y los programas que en ella se proponen están diseñados para desarrollar en principio esa habilidad, que quizás nunca tengan que utilizar, pero de la que es bueno estar preparado. Ajustándose a las interfaces típicas que ofrece C++, el código se simplifica.

34. Se pide manejar por menú (adaptado del programa 25), un listado de enteros mediante una pila.

Tabla 28. Nombres sugeridos para los elementos típicos de una pila

C++	Programa	Acción
top	**tope**	puntero al inicio de la pila
push	**Apila**	apila un valor
pop	**desapila**	desapila la cabeza
peek	**mirar**	ver el valor de la cabeza
empty	**estaVacia**	**true** si está vacía
size	**cantidad**	cuenta los nodos

Una salida debiera ser como algo así:

```
Problema 34: manejo de una pila.

Pila
1) Apilar un valor.
2) Mostrar la cabeza.
3) Contar los nodos.
4) Despilar la cabeza.
5) Ayuda.
0) Terminar.
Entre una opción: 1
¿Valor? 21
21 fué apilado.

Presione una tecla para continuar . . .
```

35. Se pide manejar por menú (adaptado del programa 25), un listado de enteros mediante una cola.

    Tabla 29. Nombres sugeridos para los elementos típicos de una cola

C++	Programa	Acción
top	**tope**	puntero al inicio de la pila
next	**prox**	puntero al próximo nodo
enqueue	**encola**	encolar un valor al final
dequeue	**desencola**	desencolar el valor de la cabeza
front	**frente**	ver el valor de la cabeza
back	**cola**	ver el valor de la cola
empty	**estaVacia**	**true** si está vacía
size	**cantidad**	cuenta los nodos

    Una salida debiera ser como algo así:

```
Problema 35: manejo de una cola.

Cola
1) Encolar un valor.
2) Mostrar las puntas.
3) Contar los nodos.
4) Desencolar un valor.
5) Ayuda.
0) Terminar.
Entre una opción: 2
Cabeza = 92     Cola = 25

Presione una tecla para continuar . . .
```

36. En ocasiones se desea que los valores estén ordenados en las LSE. Transforme la lista del programa 25 en una lista numérica ordenada. Para ello modifique la función de inserción de un nodo comparando el valor entrante con los que ya están, y encajándolo en su sitio. Una salida debiera ser como algo así:

```
Problema 36: manejo de una LSE ordenada.

LSE ordenada
1) Insertar un valor
2) Contar los nodos
3) Comprobar si un valor está
4) Mostrar la lista
5) Eliminar un valor
6) Vaciar la lista
7) Ayuda
0) Terminar
Entre una opción: 4
Lista: 14, 18, 23, 32

Presione una tecla para continuar . . .
```

37. Para formar una bibliografía, crear una lista ordenada de cadenas C++ y manejarla por menú, apoyándose en la solución del problema anterior. El programa tomará nombre, apellido y título de la obra y automáticamente generará una identificación con las iniciales de los datos. No olvidar que la bibliografía debe listar cada autor en línea aparte.

    Tomar estos nombres y entrarlos en este orden: Richard Halterman. *C++ programming*; Rex Jaeschke. *Void Punteros*; Bruce Eckel. *Pensar en C++*; Ted Jensen. *Tutorial en C*; Fernando Bellas-Permuy. *El Lenguaje C++*; Byron Gottfried. *Programación Pascal*; Rex Jaeschke. *Header Design*; Deitel & Deitel. *Programar en C/C++*; Simón Galliano. *C++ Primer*

```
Problema 36: manejo de una biblioteca.

BIBLIOGRAFÍA
1) Insertar una obra.
2) Contar las obras.
3) Buscar una obra.
4) Mostrar las obras.
5) Eliminar una obra.
6) Eliminar todas las obras.
7) Ayuda.
0) Terminar
Entre una opción: 1
Entrando una obra (nombre, apellido y título)
Autor (nombre): Richard
Autor (apellido): Halterman
Obra (título): C++ progamming
Obra entrada.

Presione una tecla para continuar . . .
```

    Recordar aplicar los fundamentos del diseño antes de comenzar a codificar. Un posible punto de partida pudiera ser el siguiente:

    - Cada obra tiene el apellido y el nombre del autor, el título de su obra y un identificador único. Sabe entrar un dato y mostrarlo.
    - Cada biblioteca tiene un listado de sus obras, sabe entrar una obra, mostrarlas, buscar una, eliminar una o todas, contarlas y vaciarse; además puede decir si está vacía, o si una obra ya está listada.

15 - ORDENAMIENTO Y BÚSQUEDAS

> *La ordenación y búsqueda son operaciones fundamentales en informática. La ordenación se refiere a la operación de organizar datos…creciente o decrecientemente para números o alfabéticamente para caracteres. La búsqueda se refiere a la operación de encontrar la posición de un elemento dentro de un conjunto de ellos. (Lipschutz, pág. 361)*

Los problemas del ordenamiento y la búsqueda sobre un conjunto de datos pertenecen a un área importante de estudio en las ciencias de la computación. La clasificación elemental de los algoritmos de ordenamiento y búsqueda que solucionan estos problemas se basa en cuatro apartados básicos:

1. El almacenaje. Se clasifican como externos si hacen uso de medios externos (discos duros, memorias USB, cintas magnéticas, etc.); o internos cuando los datos residen totalmente en la memoria operativa del equipo.
2. El tipo de dato que aceptan. Se clasifican como simples si manejan datos simples (números enteros, reales, caracteres o cadenas); o como complejos si manejan datos estructurados, estructuras de datos, etc.
3. El tipo de orden. Un algoritmo de ordenación se dice de orden estricto si dos de sus elementos con los mismos campos, tienen sus datos homólogos iguales y se dice que son *idénticos*. Un algoritmo de ordenación es débil cuando dos de sus elementos con los mismos campos, difieren en valor, aunque sea en uno de ellos, que no en todos y entonces se dice que son *similares* o *equivalentes*.
4. La estabilidad. Un algoritmo de ordenación se considera estable si al ordenarlos, ante dos valores idénticos se preserva el orden original de ambos.

¿Qué técnica se les aplica?

Existen múltiples técnicas de ordenamiento y búsqueda, y cada una de ellas tiene puntos fuertes y débiles, pues a pesar del postulado de Tim Peters (*debería haber una —y preferiblemente sólo una— manera obvia de hacerlo)*, en este caso en concreto hay muchos factores en juego. De nuevo al programador no le queda otra: hay que hacer un estudio del tema, aunque sea somero. A este nivel sólo se estudiarán los algoritmos de almacenaje interno y no se considerará la estabilidad del ordenamiento.

La notación Big-O

En las Ciencias de la Información se usa cierta notación denominada Big-O para comparar el orden de complejidad relativa temporal de un algoritmo. Usando esta medida, se puede clasificar rápidamente en categorías el tiempo de ejecución relativo de un algoritmo, así como también se pueden realizar comparaciones cualitativas entre algoritmos similares. Big-O expresa aproximadamente el orden del tiempo de ejecución de un algoritmo como una función de la cantidad **n** de los elementos, comparada contra una función matemática estándar.

Si el tiempo de ejecución es independiente del número de elementos, la complejidad es constante y se simboliza **O(1)**. Si el tiempo de ejecución crece linealmente con el número de elementos —es decir, si al duplicarse **n**, se duplica el tiempo de ejecución— la complejidad se dice que es lineal u **O(n)**. Si crece según el cuadrado de **n**, se dice que es cuadrática u **O(n^2)**, si según el cubo, se dice que es cúbica u **O(n^3)**, etc. Las funciones más empleadas son: **n**, **n^2**, **n^3**, **log n^3**, **n log n**, y **log n**; esta última simbólicamente se refiere al valor realmente calculado como **≈log n^3 + 1**.

En la tabla que sigue se dan los órdenes de complejidad para una cantidad dada de comparaciones entre claves. Por ejemplo, para **n = 5**, la complejidad **log n** es **log 5^3 + 1 ≈ 3** y **n log n** es **5×3 = 15**; para **n** = 100, la complejidad **log n** es **log 10^6 + 1 = 7** y **n log n** es **100×7 = 700**.

Tabla 30. Órdenes de complejidad temporal

n	n^2	n^3	log n^3	log n	n log n
5	25	125	≈2	3	15
10	100	10^3	3	4	40
100	1000	10^6	6	7	700
1000	1000000	10^9	9	10	10000

Según plantea (Lipschutz, págs. 87, 89, 93), la complejidad en la búsqueda lineal es de orden lineal **O(n)** y en la búsqueda binaria es de orden logarítmica **O(log n)**. Esto significa que para encontrar un elemento entre 1000 de ellos, en el peor de los casos la búsqueda lineal tendrá que visitar los 1000, mientras que la búsqueda binaria tendrá que pasar a lo sumo por un máximo de 10. En el ordenamiento por burbujas la complejidad es de orden cuadrático **O(n^2)** y en el ordenamiento rápido es lineal logarítmica **O(n log n)**. Entonces en el peor de los casos, para ordenar 1000 datos, la burbuja hará cerca de un millón de comparaciones como promedio y Quicksort unas diez mil.

Es importante observar que la notación Big-O oculta los factores constantes, y para un análisis de complejidad, en cada caso en particular no tiene importancia cuánto tiempo toma un algoritmo en realizarse. Por ejemplo, cualesquiera de dos algoritmos lineales son considerados teóricamente con el mismo nivel de aceptabilidad, independientemente de su implementación en un lenguaje dado y de su desempeño real, que puede diferir ampliamente, como se muestra en las comparaciones entre la burbuja con intercambio incrustado (página 180) y la del ordenamiento de Shell en la página 189.

Esta es una crítica válida del Big-O

El resultado es sólo una regla general de orientación, que indica, por ejemplo, que el algoritmo de la burbuja es realmente malo para casos que involucren ordenamientos masivos. Ver las comparaciones entre la burbuja con intercambio incrustado, la del Quicksort (página 181) y la de la burbuja con comparación e intercambio externos (página 189.)

Ordenamiento

Los algoritmos de ordenamiento fueron estudiados a fondo durante los 10 a 15 primeros años del boom de las computadoras, mayormente para ayudar a implantar bases masivas de datos, y en este nivel se les puede cronometrar un desempeño con facilidad, aunque no muy precisamente. Existe toda una técnica de evaluación de sus complejidades y una línea editorial dedicada a este tema que se sale del ámbito de un texto introductorio de programación. Acá se dan algunos elementos básicos.

Ordenamiento de las burbujas

En el peor de los casos, forzado a pasar por todas las comparaciones, la ordenación por burbujas hace que su complejidad promedio sea cuadrática. Esto lo cataloga como el algoritmo más ineficiente que existe, aunque para muchos programadores sea el más sencillo de implementar y por ello es comúnmente utilizado para introducir el concepto de algoritmo de ordenamiento. Editado de (Wikipedia en español), artículo Ordenamiento de burbuja.

De aquella primera época del boom surgió el *kludge*, palabra inglesa que originalmente denotaba al corte irregular de madera y que fuera acuñada por los codificadores angloparlantes para indicar un procedimiento ineficaz, pero fácilmente entendible y que eventualmente funcionaba. Y así se etiquetó al algoritmo creado en 1956, del ordenamiento de burbujas, o ***b**ubble **sort*** en inglés: `bsort()`. La ordenación por burbujas es uno de los más aceptados debido a su sencillez. Se le llama así porque con cada ciclo el valor ordenado se desplaza hacia la parte superior del arreglo, como si fuera una burbuja. (Lenguaje C. Biblioteca de funciones, pág. 142) Una implementación sencilla es:

```
Para i = 0, i < n-1; incrementar i
    Para j = n-1; j > i; decrementar j
        Si ary[j] < ary[j-1] // compara
```

```
            tmp      ← ary[j]; // intercambia
            ary[j]   ← ary[j-1];
            ary[j-1] ← tmp;
         Fin-si
      Fin-para j
   Fin-para i
```

Marcando el tiempo de un proceso

¿Cómo marcar el tiempo de un proceso de interés? Se muestra una técnica bastante primitiva, pero que trabaja. La función **difftime()** está en la biblioteca **<ctime>** y toma la cantidad de tiempo que ha tardado la operación cronometrada. El valor devuelto es un valor **double** dado en milisegundos, que dividido entre una macro definida en **<ctime>** y llamada **CLOCKS_PER_SEC** (que se ajusta al número de pulsaciones por segundo de su equipo en particular), queda en segundos. La función toma la diferencia de los tiempos final y de inicio, que vienen dados en **time_t**, tipo de valor también definido en **<ctime>** y que es equivalente a un **unsigned long**. Para ganar eficiencia, el algoritmo se incrusta en el código. Se indica una manera sencilla:

```
time_t t1 = clock();  // marca el tiempo de inicio
     ⋮                // aquí va el código de la actividad a cronometrar...
time_t t2 = clock();  // marca el tiempo final
double crono = difftime(t2, t1) / CLOCKS_PER_SEC; // tiempo medido en segundos
```

Programa 25: ordenación por burbujas

Probar el método de ordenación por burbujas sobre un arreglo de 150,000 valores enteros generados al azar. Marcar el tiempo tardado en ordenarlos.

Programa

```
#include <iostream>
#include <cstdlib>
#include <ctime>

const int N = 150000;

int main() {
  std::cout << "Programa 25: bubble sort cl\240sico.\n\nTrabajando...";
  int *pAry = new int[N];

  //  generando los valores aleatorios
  srand( time( 0 ) ); // una semilla aleatoria

  for ( int i = 0; i < N; ++i ) {
    pAry[i] = rand();
  } // for

  time_t inicio = clock(); // tiempo de inicio
  int tmp;                 // variable de intercambio

  for ( int i = 0; i < N-1; ++i ) {
    for ( int j = N-1; j > i; --j ) {
      if ( pAry[j] < pAry[j-1] ) { // compara
        // intercambia
        tmp       =  pAry[j];
        pAry[j]   =  pAry[j-1];
```

```
        pAry[j-1] =  tmp;
      } // if
    } // for j
  } // for i

  time_t fin = clock(); // tiempo final
  double tSort = difftime( fin, inicio ) / CLOCKS_PER_SEC;
  std::cout << "tard\202 " << tSort << " s en ordenar " << N << " enteros.\n\n";

  //  fin
  delete [] pAry;
  system( "pause" );
  return EXIT_SUCCESS;
}
```

Salida

```
Programa 25: bubble sort clásico.

Trabajando...tardé 51.194 s en ordenar 150000 enteros.
```

Ordenamiento rápido o del Quicksort

Está matemáticamente demostrado que no se puede hacer un algoritmo de ordenación genérico más rápido como promedio que Quicksort (Wikipedia en español). El ordenamiento rápido o Quicksort es un algoritmo que tiene varias formas de ser codificado. Fue inventado por el británico Anthony Hoare en 1960, y está basado en la técnica recursiva de "divide-y-vencerás", pero además es el algoritmo interno más rápido commo promedio.

La norma actual no especifica cuál de los muchos algoritmos del Quicksort ha de emplear un compilador dado, o si se le aplica alguna característica especial de rendimiento, pero la función está allí, es muy eficiente y bastante fácil de usar en súper C una vez que se domina la escritura de la función auxiliar de comparación. Recordar el Zen de Python: *simple es mejor que complejo; y debería haber una manera obvia de hacerlo*.

qsort

C++ ofrece una implementación llamada **`qsort()`** que permite ordenar de menor a mayor un arreglo interno, de cualquier tipo compatible con ANSI C, y cuyo prototipo es:

Tabla 31. Prototipo de qsort()

Prototipo:	`void qsort(void *ary, size_t cant, size_t tam, int *comparar(const void *p,const void *q));`
Parámetro	**Significado**
`void *ary`	Puntero a un arreglo unidimensional de valores.
`size_t cant`	Cantidad de valores del arreglo.
`size_t tam`	Tamaño en bytes de cada elemento del arreglo.
`int *comparar(const void *p const void *q)`	Función de comparación codificada manualmente, que toma dos punteros genéricos a sendos elementos del arreglo.

El algoritmo ordena un arreglo apuntado por **`ary`**, de **`cant`** elementos residentes en memoria, de tamaño **`tam`**. Como ya se dijo —y realmente EODA es una pega— el programador es quien codifica la función de comparación generalizada. Si se desea codificar al revés, de mayor a menor, una forma QAD es revertir la función auxiliar de comparación, como se muestra en la Tabla 32.

Función auxiliar de comparación

Para aplicar el algoritmo habrá una función genérica de comparación hecha para cada tipo y que debe ser codificada por el programador. Por ejemplo, para tratar un arreglo de enteros, un prototipo pudiera ser:

```
//  función de comparación de enteros
int compInt(const void * p, const void * q) {
    int m = *static_cast<const int *>(p); // tipo de dato entero
    int n = *static_cast<const int *>(q); // tipo de dato entero
    if ( m > n ) return 1;  else
    if ( m < n ) return -1; else
    return 0;
}
```

Si fueran valores **`float`** o **`double`**, se cambiaría la función a **`compDbl`** o a **`compFlt`**. Una vez apropiadamente encasillados los punteros **`p`** y **`q`** (a **`double`** o **`float`**, respectivamente), el valor del retorno siempre es un valor entero 0, 1 o -1:

Tabla 32. Función auxiliar de comparación

Función estándar: menor a mayor (a→z)		
Comparación	**Tipo de orden**	**Valor**
`p > q`	ascendente	1
`p == q`	nulo	0
`p < q`	descendente	-1
Función revertida: mayor a menor (z→a)		
Comparación	**Tipo de orden**	**Valor**
`p < q`	descendente	1
`p == q`	nulo	0
`p > q`	ascendente	-1

Programa 26: ordenamiento rápido

Probar la función **`qsort()`** de la biblioteca estándar creando un arreglo de 150,000 valores enteros generados al azar. Marcar el tiempo tardado en ordenarlos. Se adaptan las técnicas delineadas en el programa anterior. Compare ambas salidas.

Declaración

```
#ifndef COMPARACION_H_INCLUDED
#define COMPARACION_H_INCLUDED

//  función de comparación de enteros
//  input: dos valores enteros
//  output: 1 si p > q; -1 si p < q; otrosí, 0
int compInt( const void *p, const void *q );

//  genera un arreglo de n enteros, tomados al azar
//  input: el arreglo y su tamaño
//  output: el arreglo está lleno
void generaValores( int *pAry, int n );

#endif // COMPARACION_H_INCLUDED
```

Definición

```
#include "comparacion.h"
```

```
#include <cstdlib>
#include <ctime>

//  función de comparación de enteros
int compInt( const void * p, const void * q ) {
  int m = *static_cast<const int *>( p );
  int n = *static_cast<const int *>( q );
  if ( m > n ) { return 1; }
  else if ( m < n ) { return -1; }
  else { return 0; }
}

//  genera un arreglo de n enteros, tomados al azar
void generaValores( int *pAry, int n ) {
  srand( time( 0 ) );

  for ( int i = 0; i < n; i++ ) {
    pAry[i] = rand();
  } // for
}
```

Programa

```
#include "comparacion.h"
#include <iostream>
#include <cstdlib>
#include <ctime>

const int N = 150000;

int main() {
  //  info
  std::cout << "Programa 26: ordenamiento Quicksort.\n\nTrabajando...";

  //  generando los valores aleatorios
  int *pAry = new int [N];
  generaValores( pAry, N );

  time_t inicio = clock();              // tiempo de inicio
  qsort( pAry, N, sizeof( int ), compInt ); // ordenando
  time_t fin = clock();                 // tiempo final
  double tSort = difftime( fin, inicio ) / CLOCKS_PER_SEC;
  std::cout << "tard\202 " << tSort << " s en ordenar " << N << " enteros.\n\n";

  //  fin
  delete [] pAry;
  system( "pause" );
  return EXIT_SUCCESS;
}
```

Salida

```
Programa 26: ordenamiento Quicksort.

Trabajando...tardé 0.062 s en ordenar 150000 enteros.
```

El tiempo real de salida depende de lo azaroso que sean los valores del arreglo; entre más ordenados estén, el algoritmo tiende a ser más rápido.

Búsqueda

Un algoritmo de búsqueda está diseñado para localizar un elemento dentro de un conjunto de valores representativos. El elemento posee una propiedad dada, que es única para este conjunto y la búsqueda se realiza a través de otra propiedad afín llamada clave (*key*, en inglés): si ambas propiedades se emparejan, el dato se encontró y en caso contrario, no está en el conjunto.

La variante más simple es la búsqueda de un entero, y es la que se estudiará

Para independizar el algoritmo de búsqueda de su función de comparación entre un par de elementos se sigue voluntariamente la regla antes dada en la página 182.

Una estrategia

Actualmente existen aplicaciones —mayormente bases de datos— que trabajan con cantidades cuantiosas, verdaderamente masivas de registros (decenas, centenas, millones y millares de millones de ellos) y se requiere de una búsqueda que tarde un tiempo razonable, siempre teniendo presente que *razonable* es un término subjetivo que representa algo distinto para cada individuo. Para ello se aplican múltiples técnicas, y una bien sencilla y bastante usada es localizar la clave con un algoritmo binario y si está, extraerla entonces con un algoritmo lineal.

Búsqueda lineal

El texto le ofrece al industrioso lector una implementación que complementa a la función de búsqueda binaria. Comúnmente se procede en el programa a usar puntero fijo sobre la clave:

```
int key;               // clave de búsqueda
int* const pK = &key;  // puntero fijo a la clave
```

lfind

Función cuyo trabajo complementa el de la búsqueda binaria y consiste en localizar la clave e informar dónde está y cuántas veces aparece. Este algoritmo no está implementado en la biblioteca estándar, por lo que hay que codificarlo. Su complejidad es lineal y para obtener resultados correctos necesita estar previamente ordenado.

Búsqueda binaria

Es un algoritmo de naturaleza recursiva que encuentra la posición de un valor en un arreglo ordenado, comparando la clave de búsqueda con el elemento medio del arreglo. Si no son iguales, la mitad en la cual el valor no puede estar es eliminada y la búsqueda aplica el mismo algoritmo en la mitad restante hasta que el valor se encuentre o la condición de salida pruebe que no está. Editado de (Wikipedia en español)

bsearch

C++ brinda una implementación de la búsqueda binaria llamada **`bsearch()`**, que aplica sobre un arreglo de elementos compatibles con ANSI C y previamente ordenados. Si ordenó de menor valor a mayor, el algoritmo buscará del inicio hacia

el final. Si ordenó de mayor a menor, buscará al revés. **bsearch** no refleja casamientos y devolverá un puntero **void** al valor que emparejó con la clave por vez primera, o un puntero nulo si la clave no está. Su prototipo es:

Tabla 33. Prototipo de bsearch()

Prototipo: `void *bsearch(void *key, const void *ary, size_t n, size_t t, int *comp(const void *m, const void *n));`	
Parámetro	**Significado**
`void *key`	Puntero al valor de la clave
`void *ary`	Puntero a un arreglo unidimensional de valores nativos de ANSI C.
`size_t n`	Cantidad de valores del arreglo.
`size_t t`	Tamaño en bytes de cada elemento del arreglo.
`int *comp(const void *m, const void n*)`	Función de comparación codificada manualmente, que toma dos punteros genéricos a sendos elementos del arreglo.

Notas:

- **bsearch()** devuelve un puntero **void** que hay que encasillar para desreferenciarlo. Por ejemplo, suponiendo un arreglo de enteros, al cual se le aplicó el algoritmo. El valor de la clave se puede obtener así:

```
//  búsqueda binaria: devuelve un puntero void al valor
void *idx = bsearch(K, pAry, N, sizeof(int), compInt);
std::cout << "El valor buscado es " << *reinterpret_cast<int*>(idx) << '\n';
```

- La función de comparación se codifica igual que para **qsort()** y sigue tomando dos punteros genéricos. Como ya se dijo, deberá ser hecha por el programador.

Programa 27: búsqueda

Para probar el algoritmo **bsearch()** con visualización, se crea un arreglo de 80 valores enteros entre uno y 100, generados al azar, sobre el cual puede comprobarse visualmente la operación de búsqueda. Una vez decidido que el valor clave está, mostrar sus posiciones.

Por lo común se procede a convertir la clave en un puntero de lectura solamente. Por otra parte, **bsearch()** devuelve un puntero genérico con el valor nulo si no está, o el valor de la clave si está, pero no da la posición de la clave en el arreglo y habitualmente lo que se quiere es saber ambas cosas. El programa estipula usarla para conocer si aparece la clave, por la rapidez de esta operación; una vez se sabe que está, usar la forma apropiada de búsqueda lineal para encontrar todos sus valores.

Declaración

```
#ifndef BINARY_SEARCH_H_INCLUDED
#define BINARY_SEARCH_H_INCLUDED

//  función de comparación de enteros
//  input: dos valores enteros
//  output: 1 si p > q; -1 si p < q; otrosí, 0
int compInt(const void *p, const void *q);

//  genera un arreglo de n enteros, tomados al azar entre 1 y 100
//  input: el arreglo y su tamaño
//  output: el arreglo está lleno
void generaValores(int *pAry, int n);

//  busca todas las posiciones de un valor en un arreglo de enteros
//  entrada: punteros a la clave y al arreglo, y el tamaño de éste
//  salida: un aviso con la respuesta
```

```
void lfind(const int *K, const int *pAry, int n);

//  muestra el arreglo en cols columnas
//  input: el arreglo, su tamaño y las columnas a listar
//  output: el arreglo está en consola
void mostrar(const int *pAry, int n, int cols);

#endif // BINARY_SEARCH_H_INCLUDED
```

Definición

```
#include "binary_search.h"
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iomanip>
using std::cout;

//  función de comparación de enteros
int compInt( const void *p, const void *q ) {
    int m = *static_cast<const int *>( p );
    int n = *static_cast<const int *>( q );
    if ( m > n ) { return 1; }
    else if ( m < n ) { return -1; }
    else { return 0; }
}

//  genera un arreglo de n enteros, tomados al azar entre 1 y 100
void generaValores( int *pAry, int n ) {
    srand( time( 0 ) );

    for ( int i = 0; i < n; i++ ) {
        pAry[i] = 1 + rand() % 100;
    } // for
}

//  busca todas las posiciones de un valor en un arreglo de enteros
void lfind( const int *K, const int *pAry, int n ) {
    int j = 0, k = 0;

    for ( int i = 0; i < n; ++i ) {
        if ( pAry[i] < *K ) {
            ++k;   // donde comienzan a repetirse
        } else if ( pAry[i] == *K ) {
            ++j;   // las repeticiones
        } else {
            break; // se acabó la visita
        } // if-en-cascada
    } // for

    ( 1 == j ) // C++ cuenta el índice, nosotros la posición (una más)
       ? cout << *K << " est\240 una vez en la posici\242n " << ++k
       : cout << *K << " est\240 " << j << " veces a partir de la posici\242n " << ++k;

    cout << "\n\n";
}

//  muestra el arreglo en cols columnas
void mostrar( const int *pAry, int n, int cols ) {
    for ( int i = 0, columna = 1; i < n; ++i, ++columna ) {
        cout << pAry[i];
```

```
        ( 0 == columna % cols ) ? cout << '\n' : cout << '\t';
    } // for

    cout << '\n';
}
```

Programa

El curioso lector inmediatamente se percata de que el programa no necesita la búsqueda binaria porque el algoritmo de la búsqueda lineal sobre un arreglo ordenado hace todo el trabajo. Y es verdad…en un programa de los llamados "de juguete", que obligatoriamente son los que vienen en la inmensa mayoría de los textos dedicados a la enseñanza de cualquier lenguaje de computación —y más aún, a nivel introductorio, lo cual es totalmente válido para este texto.

Pero en una aplicación que buscará valores, a veces complejos, sobre varios millones de registros, es imprescindible utilizar la estrategia mostrada. Por eso está puesta: como recurso pedagógico para exponer su uso.

```
#include "binary_search.h"
#include <iostream>
#include <cstdlib>
#include <ctime>

const int N = 80;

int main() {
    std::cout << "Programa 27: prueba de la b\243squeda binaria.\n\n";
    int *pAry = new int[N]; // un puntero al vector
    int key;                // clave de búsqueda
    int* const K = &key;    // puntero fijo a la clave
    void *idx = nullptr;    // valor a buscar

    generaValores( pAry, N );
    qsort( pAry, N, sizeof( int ), compInt );
    mostrar( pAry, N, 10 );

    while ( true ) {
        std::cout << "T para terminar. \250Clave? ";
        std::cin >> key;

        if ( not std::cin.good() ) {
            delete [] pAry;
            return EXIT_SUCCESS;
        } // if

        //  búsqueda binaria: devuelve un puntero void al valor
        idx = bsearch( K, pAry, N, sizeof( int ), compInt );

        if ( nullptr == idx ) {
            std::cout << *K << " no est\240.\n\n";
        } else {
            lfind( K, pAry, N );
        } // if-else
    } // while
}
```

Salida

```
Programa 27: prueba de la búsqueda binaria.

1       1       1       2       6       8       9       10      10      10
10      11      11      11      12      14      18      18      19      21
22      22      23      24      26      26      26      28      30      30
33      34      36      36      37      37      39      40      41      42
45      45      46      47      47      48      50      51      55      57
58      59      59      61      63      63      67      67      68      72
73      74      78      78      81      81      82      82      83      83
85      88      88      91      96      96      97      97      99      100

T para terminar. ¿Clave? 1
1 está 3 veces a partir de la posición 1

T para terminar. ¿Clave? 18
18 está 2 veces a partir de la posición 17

T para terminar. ¿Clave? 72
72 está una vez en la posición 60

T para terminar. ¿Clave? 98
98 no está.

T para terminar. ¿Clave? t
```

Estado del flujo

Véase la condición lógica dentro del ciclo:

```
if ( not std::cin.good() ) {
  delete [] pAry;
  return EXIT_SUCCESS;
} // if
```

Tiene que ver con el flujo de entrada **`cin`**. Por ejemplo, cuando se están leyendo enteros, si se entra un carácter o un número real, el flujo de entrada se corrompe, deja de ser "bueno" (*good* en inglés) y la condición falla, terminando el ciclo. Así que cualquier carácter o cualquier número real sirve para terminarlo. Y está bien cuando eso es lo que se desea, tal como es en este caso.

Pero, ¿qué pasa si el programa debe seguir? Entonces hay que hacer dos cosas en este orden: **(a)** reponer el estado del flujo y **(b)** "limpiar" el búfer, si no, la ejecución se "congela" y no sigue su curso. Un método QAD, aunque simple y seguro, es usar **`clear()`** para reponer el estado a **`cin.good()`** y seguidamente eliminar todo lo que se quedó en el búfer.

```
if ( not std::cin.good() ) {
   std::cin.clear();                      // (a) repone el estado del flujo cin

   while ( std::cin.get() not_eq '\n' ) {  // (b) elimina hasta el último carácter del búfer...
      continue;                            //     ...leyéndolo continuamente
   } // while

} // if
```

Siempre que la ejecución del programa deba continuar luego de aceptar una entrada corrupta, ante todo considerar el empleo de las instrucciones mencionadas. En el tomo que viene se da una técnica más fácil y mejor.

OCTAVO BLOQUE

38. Probar el método de ordenamiento por burbujas con comparación e intercambio externos, sobre un arreglo de 150,000 valores enteros generados al azar. Marcar el tiempo tardado en ordenarlos y compararlo contra el tiempo tardado en generarlos. Modificar la función **`bsort()`**para que compare y ordene externamente. El lector deberá codificar una función que intercambie dos valores enteros, y otra que devuelva verdadero si el primer valor es mayor que el segundo. Ambas serán usadas por **`bsort()`**. Contrastar con la salida del Programa 25: ordenación por burbujas. ¿Se nota alguna diferencia? ¿A qué se debe?

```
Problema 38: bubble sort con intercambio externo.

Trabajando...tardé 45.163 s en ordenar 150000 enteros.
```

En el programa 25, el tiempo empleado en ordenar fue de 51.194 s, mientras que aquí es de 45.163 s —en cada ejecución los tiempos pueden variar ligeramente. Prácticamente no hay diferencia entre ambas versiones. ¿Qué se ganó acá? Mucha abstracción. Con esta versión se pueden ordenar arreglos conteniendo los tipos estándar de datos POD de C++ sin tener que alterar su código; solamente añadiendo las funciones de comparación e intercambio a las cabeceras. El código se desarrolla más rápido y es más “limpio”.

Hoy día —y esto es muy importante— siempre se prefiere la abstracción

---

Hay formas mucho mejores de ordenar, pero eso es una historia por contar…

39. El ordenamiento **`shellsort()`** se llama así en honor de su inventor Donald Shell, quien lo publicó en 1959. *Fue desarrollado para subsanar la falta de eficiencia de la ordenación por burbujas con arreglos grandes*. (Lenguaje C. Biblioteca de funciones, pág. 144). Su implementación original, requiere **O(n2)** comparaciones e intercambios en el peor caso, pero aun así derrota fácilmente al de burbujas. Además, se requiere que en este ejercicio el **`shellsort`** sea codificado como una función de orden superior. Por ejemplo,

```
//  ordena un arreglo de enteros
//  input: el arreglo, su tamaño y la función de comparación
//  output: el arreglo está ordenado
void shellsort(int *ary, int n, bool mayorQue(int p, int q));
```

El seudocódigo fue tomado y transcrito de la fuente bibliográfica antes citada.

```
//  ordena un arreglo de n enteros
shellSort(ary, n, *fuera_de_orden, zancada):
    n ← número_de_elementos
    zancada ← n/2
    hubo_cambio ← falso

    mientras_que zancada > 0
        mientras_que hubo_cambio sea verdadero
           mientras_que i > (n – zancada)
              si ary[i] > ary[i+zancada]
                    intercambia ary[i] con ary[n + zancada]
                    hubo_cambio ← verdadero
                    incrementa i
              fin-si
           fin-mientras_que
        fin-mientras_que

        zancada ← zancada / 2
    fin-mientras_que_zancada > 0
fin-shellSort
```

Codificar un manejador parecido al del ejercicio anterior y comparar ambos resultados. La salida debiera verse así:

```
Problema 39: ordenamiento Shell.

Trabajando...tardé 2.77 s en ordenar 1500000 enteros.
```

En el ejercicio anterior el tiempo empleado en ordenar fue de 45.163 s, mientras que aquí es de 2.77 s. En cada ejecución los tiempos pueden variar ligeramente. Se nota la gran diferencia contra el de burbujas: con el mismo orden de complejidad, el ordenamiento Shell sobre cantidades iguales es alrededor del 85-95 % más eficiente.

16 - MANEJO BÁSICO DE ARCHIVOS

Un archivo o fichero informático es un conjunto de bits que son almacenados en un dispositivo. Un archivo es identificado por un nombre y la descripción de la carpeta o directorio que lo contiene. A los archivos informáticos se les llama así porque son los equivalentes digitales de los archivos escritos en expedientes, tarjetas, libretas, papel o microfichas del entorno de oficina tradicional. (Wikipedia en español)

Un archivo en C + + es una secuencia de bytes. Un búfer, que es un área de la memoria reservada para almacenar temporalmente datos, meramente lee o escribe esos bytes allí. El fichero **<fstream>** declara las clases y los otros tipos requeridos para hacer E/S con todo tipo de archivos (ficheros-fuente y/o externos.) La E/S está implementada enteramente por bibliotecas, porque debido a la tersura de C++, ninguna de las características del lenguaje soporta específicamente alguna de esas operaciones.

En C++ la salida se interpreta como un flujo de datos vertiendo desde un recipiente a una corriente de bytes; la entrada es interpretada como un flujo de datos manando desde una corriente de bytes a un recipiente; esos recipientes son los búferes: las áreas de memoria que antes fueran reservadas.

Ventaja de usar las clases con archivos

Comparada con el mecanismo clásico de la administración de E/S, la gerencia automática mediante clases que manejan los flujos de acceso, ofrece ventajas significativas. Los archivos son abiertos en el instante de su construcción, cerrados automáticamente en el momento de su destrucción y el trabajo es transparente: el programador puede permitirse “olvidar” la administración del flujo puesto que las clases lo hacen por él.

¿Por qué usar los archivos?

Para que una aplicación sea útil de verdad debe permitir guardar datos e información. Por ejemplo, las aplicaciones procesadoras de texto, las de bases de datos y de hojas electrónicas contables, las presentaciones, etc. pueden guardar y retomar datos a voluntad del usuario. Hasta ahora en el texto se han visto varias formas de manejar datos complejos, poniéndolos en registros, y usando arreglos y listas de estos registros. Todo ello sería más útil si los datos fueran persistentes.

- ✓ En informática, persistente se refiere a la propiedad de los datos para que estos sobrevivan de alguna manera En programación, la persistencia es la acción de guardar de forma permanente a un periférico la información generada por una aplicación, pero a su vez también se refiere a poder recuperarla, para ser nuevamente utilizada.

Usualmente un programa que maneje o genere datos persistentes permite crearlos y grabarlos a un fichero. En otro momento si así lo desea, el usuario puede recargar los datos y proseguir con su labor.

Archivo de textos

Para crear un archivo de texto para operaciones de entrada de datos, se usa un editor como el Notepad++ de la fundación libre GNU, o el Bloc de Notas de Windows™. El editor de código que aquí se aplica es suministrado por el IDE del Code::Blocks y también produce este tipo de archivo. Sus ficheros-fuente son ejemplos de archivos de texto y se puede emplear cualquier editor de la clase de los antes mencionados para ver o cambiar sus contenidos fuera del IDE. También se puede utilizar un procesador de textos como Word™ siempre que se grabe el archivo en formato de texto.

- ✓ Archivo de texto es aquél donde cada byte representa un código ASCII de caracteres.

No todos los archivos son de texto. Por ejemplo, las aplicaciones de bases de datos y de hojas electrónicas contables almacenan datos numéricos en formatos apropiados. Las aplicaciones procesadoras de texto, aun cuando contienen texto, también contienen datos no textuales para poder describir los tipos de caracteres que usan, sus tamaños y peso, los distintos tipos de impresoras, etc.

En este apartado se examinan las operaciones de E/S que imitan la entrada de datos al programa por teclado y la salida de datos del programa a consola, y siempre deben ser usadas con archivos de texto.

Operaciones de escritura a un archivo

Primeramente, se examinará la salida de datos desde el programa a un archivo. C++ hereda esta operación del ANSI C. La salida se maneja como datos vertiendo desde un búfer a un periférico adecuado (consola, ficheros, impresoras y otros tipos de accesos), sin embargo, C++ facilita la operación mediante clases especiales. Después de todo, ¡Stroustrup sólo quería facilitarle la programación a él y a un grupo de sus amigos!

Clase de flujo de salida

Tal como hay diferentes tipos de E/S, hay clases diferentes que los manejan, en dependencia del tipo. La clase más importante de salida para ficheros es **<fstream>**, quien define el flujo que puede ser usado para escribir a un periférico cualquiera, en particular a un archivo.

Objetos globales de flujo de salida

La biblioteca **<fstream>** define tres objetos globales de salida:

1. **cout** es el canal estándar de salida usado por el programa, y conectado por omisión a la consola.
2. **cerr** es el canal estándar de salida de los mensajes de error usado por el programa, y conectado a la consola por omisión. No tiene búfer: su salida es inmediata.
3. **clog** es el canal estándar de la bitácora. Por omisión va conectado al mismo destino que **cerr**, pero tiene búfer de salida, por lo que puede ser redirigido a un archivo para su posterior inspección. No será empleado aquí.

Al ejecutar un programa, la separación de mensajes de salida y de errores facilita tratarlos de forma diferente. Por ejemplo, la salida normal puede ser dirigida a un archivo, mientras los mensajes de error van a **cerr** apareciendo en consola. Algo así se ha hecho en el texto. Usualmente los mensajes de error por **clog** van a otro archivo que funge como cuaderno de bitácora.

Programa 28: escribiendo a un archivo

Se le pide codificar un programa que genere 200 números enteros aleatorios, entre 1 y 100 y los ponga en un archivo llamado C:\temp\datos. Si se da el nombre del archivo, el sistema lo graba en el directorio de salida que esté usando su IDE. Cuando posteriormente se desee leerle, habrá que replicarlo en el otro lugar donde se actuará con la lectura. Lo más fácil, aunque es una solución QAD, es poner el camino completo fuera del sistema y luego, llamarlo desde allí mismo.

Ya se vio en las *Principales secuencias de escape* que la barra contra-inclinada debe ponerse doble para una lectura correcta; en la página 61 se aprendió a generar números seudoaleatorios; y el programa dado en la página 184 se resolvió codificando una función que imprime los datos de un vector numérico en una tabla controlada por las columnas a poner en consola. Reuniendo todos estos conocimientos, se armará el ejemplo.

Declaración

```
#ifndef SALIDA_H_INCLUDED
#define SALIDA_H_INCLUDED

#include <string>
typedef std::string Str;

const Str ERR( "No pude abrir el archivo. Abortando.\n" );

//  escribe los valores del arreglo ary al archivo archy
//  entrada: nombre del archivo, y nombre y tamaño del arreglo
//  salida: los valores están en el archivo
void alArchivo( Str archy, const int *pAry, int n );

//  genera un arreglo de n dobles, tomados al azar entre 1 y 100
//  input: el arreglo y su tamaño
//  output: el arreglo está lleno
void generaValores( int *pAry, int n );

//  muestra el arreglo en cols columnas
//  input: el arreglo, su tamaño y las columnas a listar
//  output: el arreglo está en consola
void mostrar( const int *pAry, int n, int cols );

#endif // SALIDA_H_INCLUDED
```

Definición

```
#include "salida.h"
#include <iostream>
#include <cstdlib>
#include <fstream>
#include <ctime>
#include <iomanip>
using std::cout;
using std::setw;

//  esta función escribe los valores del arreglo pAry al archivo archy
void alArchivo( Str archy, const int *pAry, int n ) {
  std::ofstream Escribe( archy );
```

```
  if ( Escribe.fail() ) {
    throw ERR;
  } else {
    for ( int i = 0; i < n; ++i ) {
      Escribe << pAry[i] << '\n';
    } // for
  } // if-else
}

//  genera un arreglo de n enteros, tomados al azar entre 1 y 100
void generaValores( int *pAry, int n ) {
  srand( time( 0 ) );

  for ( int i = 0; i < n; i++ ) {
    pAry[i] = 1 + rand() % 100;
  } // for
}

//  muestra el arreglo en cols columnas
void mostrar( const int *pAry, int n, int cols ) {
  for ( int i = 0, columna = 1; i < n; ++i, ++columna ) {
    cout << setw( 3 ) << pAry[i];
    ( 0 == columna % cols ) ? cout << '\n' : cout << '\t';
  } // for

  cout << '\n';
}
```

Programa

```
#include "salida.h"
#include <iostream>
#include <cstdlib>

const int N      = 200; // cantidad de valores
const short COLS = 10;  // columnas

int main() {
  Str archy( "C:/temp/datos.txt" );

  // info
  std::cout <<
       "Programa 28: escribe " << N << " enteros aleatorios\n"
       "en el archivo '" << archy << "'\n\n";
  int *pAry = new int[N];   // los valores en la free store
  generaValores( pAry, N ); // los valores al arreglo

  try {
    alArchivo( archy, pAry, N ); // se escriben a disco
    mostrar( pAry, N, COLS );    // valores a consola
  } catch ( Str ) {
    std::cout << ERR;
    return EXIT_FAILURE;
  } // try-catch

  system( "pause" );
  return EXIT_SUCCESS;
}
```

Dos salidas

a. Sin error

```
Nombre del archivo a leer: c:\temp\datos.txt

    54      42      76      20       9       6      88      80      48      11
    86      53      16      23      76      81       8      33      16      86
    93      65      66       9      57      81      46      82       8      65
    72       6      41      57      26      75      81      85      82      27
    34      37      53      84      34      58      68      11      42      74
    93      20      92      55      83      60      30      22      53      44
    94      45      53      93      55       5       9      48      21      90
    46      52      16      39      51      98      33      30      59      21
    45      47      79      34      46      36      31      77      16      46
     8      13       4      90      61      12      59      18      12      99
    12      25      92      44      31      91      68      96      65      30
    94      84      87      70      81      34      72      83      29      50
    41      33      93      50       8      51      29      92      29      32
    28      75       3      10      55      91      61      11      24       6
    83      70      61      15      84      25      45      80      13      24
    88      40      44      63      69      78       1      96      77      33
    17      11      85      48      33      13      52      96       1       6
    10       7      61      78       8      71      90      62      81      24
    42       4      37      86       6      44      60       6      44      45
    41      46      56      51      83      15      38      18      59      95
```

b. Con error

```
Programa 28: escribe 200 enteros aleatorios
en el archivo 'J:/temp/datos.txt'

No pude abrir el archivo. Abortando.
```

C++ escribe en la localización dada. Si el archivo no existe, lo crea, pero si el camino no existe, no lo construye; simplemente no puede escribir y da error. En este sistema la torre **J** no existe. Si no se posee la ubicación C:/temp, la ejecución dará un error similar. Una vez que la localización está dispuesta, el programa no falla. Para hacer el subdirectorio se ejecuta la orden **`cmd`** desde Windows 10 que abre una ventana en el SO **DOS**. Otras versiones de Windows™ usan otro comando.

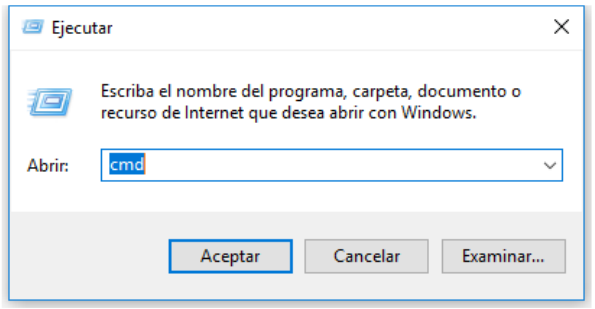


Ilustración 16. Accediendo al DOS

Ya en la ventana del **DOS**, teclear los comandos en el orden dado.

1. El comando **`chdir c:\`** cambia el directorio a torre C.
2. El comando **`mkdir c:\temp`** crea el subdirectorio.
3. **`dir`** permite ver lo que hay en el subdirectorio. La salida en otro equipo será diferente.
4. Comprobar visualmente que el subdirectorio **`temp`** ha sido creado.

```
C:\>dir
 El volumen de la unidad C es Simon
 El número de serie del volumen es: 9027-2A1B

 Directorio de C:\

17/02/2019  01.10 PM    <DIR>          Intel
30/08/2020  08.43 PM    <DIR>          MinGW
29/09/2017  09.46 AM    <DIR>          PerfLogs
05/08/2020  05.23 AM    <DIR>          Program Files
29/08/2020  12.56 PM    <DIR>          Program Files (x86)
01/09/2020  06.19 AM    <DIR>          temp
18/02/2019  01.17 PM    <DIR>          Users
31/08/2020  10.30 AM    <DIR>          Windows
               0 archivos              0 bytes
               8 dirs  62,210,441,216 bytes libres
```

Ilustración 17. El subdirectorio temp

Operaciones de lectura en un archivo

Ahora se examinará la entrada de datos desde un archivo al programa. C++ también hereda esta operación del ANSI C. La entrada es interpretada como un flujo de datos manando desde una corriente de bytes a un búfer adecuado (arreglos, estructuras, etc.), sin embargo, C++ facilita la operación mediante clases especiales. Después de todo, ya hemos dicho y repetido: Cuando todo empezó, ¡Stroustrup sólo quería facilitarle la programación a él y a un grupo de sus amigos!

Clase de flujo de entrada

Tal como hay diferentes tipos de E/S (por ejemplo, input, output, y accesos a los archivos), hay clases diferentes que los manejan, en dependencia del tipo. La clase más importante de entrada es **<fstream>**, quien define el flujo que puede ser usado para leer desde un archivo.

Objeto global de flujo de entrada

La biblioteca **<fstream>** define un solo objeto global de entrada: **cin** es el canal estándar de lectura, conectado por omisión al teclado.

Flujos a la medida

No obstante, crear flujos de entrada o de salida la medida de los intereses del usuario, incluyéndoles los parámetros que se requieran, hace que estas operaciones sean más “inteligentes” aclarando y simplificando mucho esas operaciones. Usualmente las interfaces son grandes, indicando que, en efecto, es una función a la medida de un programa, fuertemente enlazada a él y no son ajustadas, ni se hacen reflejando el entorno global, sino que son muy puntuales.

En este caso, por ejemplo, al tratar con enteros se sabe que los estadísticos **moda**, **máximo**, **mínimo**, **suma** y **cuenta** son resultados siempre enteros, mientras que el **promedio**, la **varianza** y la **desviación_típica** raramente lo son (en el caso de la **mediana**, depende de si la cantidad de elementos es par o no.) Entonces se puede crear una función inteligente que discrimine lo que ha de poner en la salida.

¿Y por qué no hacer una función “normal”? Pues porque habría que codificar una para la salida a consola y otra para la salida a fichero, lo cual no es muy eficiente. Con esta manipulación a la medida del flujo, la acción se puede redirigir a la consola o a un fichero con mucha facilidad. En el programa se usó la siguiente:

```
//  flujo de salida especializado
std::ostream &FMT( std::ostream &flujo,  // el flujo de datos
                   Str ID,               // la identificación de la salida
                   double stat,          // el valor del estadístico
                   bool real = true );   // el tipo de número: real o entero
```

Programa 29: leyendo desde un archivo

Se pide codificar un programa que lea los números contenidos en el archivo C:\temp\datos.txt mediante una función de orden superior, y les calcule los estadísticos típicos (ver su método de cálculo en un texto adecuado de estadística), que son seis: tres de tendencia central y tres de dispersión:

- Medidas de tendencia central: son el promedio $\bar{x}$ (average o media aritmética), la mediana y la moda. El promedio ya fue definido anteriormente, pero se repite por claridad:

  Sean $\bar{x}$ el promedio, x_i el número iésimo de un conjunto de valores y n su potencia (cantidad de valores del conjunto.) El promedio se define como la suma de todos los valores x_i del conjunto, desde $i-1$ hasta n dividida por su potencia:

  $$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n} = \frac{\sum x_i}{n}$$

  Sean x_{Med} la mediana, x_i el número iésimo de un conjunto ordenado de valores y n su potencia. La mediana se define como el elemento central de un conjunto de valores ordenados:

  $x_{Med} = x_{\frac{n}{2}}$ si n es par, y $x_{Med} = \frac{\left(x_{\frac{n}{2}-1} + x_{\frac{n}{2}}\right)}{2}$ si n es impar

  Sean x_{Mod} la moda, x un valor cualquiera de un conjunto de valores ordenados y n su potencia.

  $x_{Mod} = \forall x$ es el valor que más se repite

  Si el conjunto es grande, pueden aparecer varias modas. Si son dos, la distribución de valores se llama bimodal y sugiere que hay valores mezclados. Si hay tres, la moda se llama trimodal. Más valores son raros, pero ¡atención! si no hay un valor que aparezca más de una vez, no hay moda.

- Medidas de dispersión: las más importantes son el rango, la varianza y la desviación típica:

  Sean x_{Max}; x_{Min} los valores extremos de un conjunto numérico. El rango (A: alcance) se define como la diferencia positiva de los valores extremos:

  $$A = |x_{Max} - x_{Min}|$$

  Sean s^2 la varianza, $\bar{x}$ el promedio, x_i el número iésimo de un conjunto de valores y n su potencia. La varianza de una muestra se define como aproximadamente el promedio de las deviaciones elevadas al cuadrado (se divide por la potencia del conjunto menos uno.)

  $$s^2 = \frac{\sum(x_i - \bar{x})^2}{n-1}$$

  Sea s^2 la varianza de una muestra. La desviación típica s se define como la raíz cuadrada positiva de la varianza:

  $$s = \sqrt{s^2} = \sqrt{\frac{\sum(x_i - \bar{x})^2}{n-1}}$$

Una vez hecho esto, escribir la respuesta a consola y a un archivo llamado C:\temp\resp.txt

Hay dos problemas aquí. Se usa un arreglo de enteros y por lo tanto: (1) solamente lee enteros y, (2) el tamaño es fijo — se codifico el tamaño aparte y si los límites lo sobrepasan, hay que arreglar el código y recompilar. También hay que considerar que se mezclan enteros y reales; la salida refleja esto.

Declaración

```
#ifndef ESTADS_H_INCLUDED
#define ESTADS_H_INCLUDED

#include <ostream>
#include <string>
typedef std::string Str;
const Str ERROR( "\nNo pude abrir el archivo; abortando.\n" );
const int TAM = 200;

struct Estads {
    //  interfaz
    int suma          ();
    double mediana    ();
    int moda          ();
    double promedio   ();
    double varianza   ();
    double desviacion ();
    int mi_max        ();
    int mi_min        ();
    int cuenta        ();
    void leerArchivo  ( Str archy );
    // miembro
    int ary[TAM];
    int n;
};

//  flujos de salida especializados
void muestra( std::ostream & escribe, Str ID, int stat );
void muestra( std::ostream & escribe, Str ID, double stat );

//  función de comparación
int compara( const void *p, const void *q );

#endif // ENTRADA_H_INCLUDED
```

Definición

```
#include "estads.h"
#include <fstream>
#include <iomanip>
#include <cmath>

//  función de comparación
int compara( const void *p, const void *q ) {
    int m = *static_cast<const double *>( p );
    int n = *static_cast<const double *>( q );
    if ( m > n ) { return 1; }
    else if ( m < n ) { return -1; }
    else { return 0; }
}

//  valores leídos desde el archivo
void Estads::leerArchivo( const Str archy ) {
```

```
    // aquí hace falta código defensivo, porque
    // si el fichero no está, no puede abrirlo
    std::ifstream Leyendo;
    Leyendo.open( archy );

    if ( Leyendo.is_open() ) {
        double val;
        int i = 0;

        while ( Leyendo >> val ) {
            ary[i] = val;
            ++i;
        } // while

        n = i;
        qsort(ary, n, sizeof(int), compara);
    } else {
        throw ERROR;
    } // if-else
}

int Estads::suma () {
    int total = 0;

    for ( int i = 0; i < n; ++i) {
        total += ary[i];
    } // for

    return total;
}

double Estads::promedio() { return suma() / n; }
int Estads::mi_max() { return ary[n-1]; }
int Estads::mi_min() { return ary[0]; }

double Estads::mediana() {
    double med;
    int idx = 0.5 * n; // se buscó el índice del medio
    n % 2              // se calculó la mediana
        ? med = ary[idx]
        : med = 0.5 * ( ary[idx - 1] + ary[idx] );
    return med;
}

// moda
int Estads::moda() {
    int unaModa = ary[0];   // el 1er. valor
    int laModa  = unaModa;  // la moda actual
    int unaRacha    = 0;    // la 1ra. racha
    int laRacha     = unaRacha; // el tamaño de una racha
    int i           = 0;        // el índice de la secuencia

    while ( i not_eq n ) {
        if ( unaModa not_eq ary[i] ) { // ¿cambió la moda?
            unaModa = ary[i];          // SI: actualizarla

            if ( unaRacha > laRacha ) { // ¿cambió la racha?
                laRacha = unaRacha;     // SI: actualizarla y...
                laModa  = ary[i - 1];   // ...actualizar la moda
            } // if-interior

            unaRacha = 0; // contar otra racha
```

```
        } // if-exterior

        ++unaRacha; // incrementa la racha
        ++i;        // incrementa el índice
    } // while

    if ( 1 == laRacha ) {
        laModa = static_cast<int>(HUGE_VAL);
    } // if

    return laModa;
}

double Estads::varianza() {
    double total = 0;
    double ave = promedio();

    for (int i = 0; i < n; ++i) {
        total += pow( ary[i] - ave, 2 );  // se visitan los nodos
    } // for

    return total / ( n - 1 ); // varianza muestral
}

double Estads::desviacion() { return sqrt( varianza() ); }

void muestra( std::ostream & escribe, Str ID, int stat ) {
    escribe.setf( std::ios::fixed );

    if ( static_cast<int>(HUGE_VAL) == stat ) {
        escribe << "Moda = #N/A\n";
    } else {
        escribe << std::setw( 12 ) << ID << std::setw( 4 ) << stat << '\n';
    } // if-else

    escribe.unsetf( std::ios::fixed );
}

void muestra( std::ostream & escribe, Str ID, double stat ) {
    escribe.setf( std::ios::fixed );
    escribe << std::setprecision( 4 );
    escribe << std::setw( 12 ) << ID << std::setw( 9 ) << stat << '\n';
    escribe.unsetf( std::ios::fixed );
    escribe << std::setprecision( 6 );
}
```

Programa

```
#include "entrada.h"
#include <iostream>
#include <fstream>

int main() {
    std::cout << "Programa 29: lee los n\243meros de un archivo\n"
                 "y calcula sus principales estad\241sticos\n\n";
    Estads est; // los estadísticos
    Str archy;  // nombre de los archivos de lectura y escritura
    std::cout << "Nombre del archivo a leer: ";
    std::cin >> archy;

    // código defensivo aquí
```

```
    try {
        std::cout << "Leyendo del archivo " << archy;
        est.leerArchivo( archy );
        std::cout << "...hecho.\n\n";
    } catch( Str ) {
        std::cout << ERROR;
        return EXIT_FAILURE;
    } // try-catch

    //  respuestas a consola
    std::cout << "Estad\241sticos a consola:\n\n";
    muestra( std::cout, "Media = ",      est.promedio()    );
    muestra( std::cout, "Mediana = ",    est.mediana()     );
    muestra( std::cout, "Moda = ",       est.moda()        );
    muestra( std::cout, "Varianza = ",   est.varianza()    );
    muestra( std::cout, "Desviac. = ",   est.desviacion()  );
    muestra( std::cout, "M\240ximo = ",  est.mi_max()      );
    muestra( std::cout, "M\241nimo = ",  est.mi_min()      );
    muestra( std::cout, "Rango = ", est.mi_max() - est.mi_min() );
    muestra( std::cout, "Suma = ",   est.suma() );
    muestra( std::cout, "Cuenta = ", est.n );
    std::cout << "\n";

    //  respuestas al disco
    std::cout << "Nombre del archivo a escribir: ";
    std::cin >> archy;

    // un flujo de salida
    std::ofstream Escribiendo;
    Escribiendo.open( archy );

    if ( Escribiendo.is_open() ) {
        std::cout << "Escribiendo al archivo " << archy;
        muestra( Escribiendo, "Media = ",      est.promedio()    );
        muestra( Escribiendo, "Mediana = ",    est.mediana()     );
        muestra( Escribiendo, "Moda = ",       est.moda()        );
        muestra( Escribiendo, "Varianza = ",   est.varianza()    );
        muestra( Escribiendo, "Desviac. = ",   est.desviacion()  );
        muestra( Escribiendo, "M\240ximo = ",  est.mi_max()      );
        muestra( Escribiendo, "M\241nimo = ",  est.mi_min()      );
        muestra( Escribiendo, "Rango = ", est.mi_max() - est.mi_min() );
        muestra( Escribiendo, "Suma = ",   est.suma() );
        muestra( Escribiendo, "Cuenta = ", est.n );
        std::cout << "...hecho.\n\n";
    } else {
        std::cout << ERROR;
        return EXIT_FAILURE;
    } // if-else

    system( "pause" );
    return EXIT_SUCCESS;
}
```

Salida

C++ lee en la localización dada, pero si no existe, simplemente no puede leer. Una vez que la localización C:/temp está dispuesta, el programa no falla. C++ escribe en la localización dada, pero si no existe, simplemente no puede escribir. Ambos dan el mismo error de ejecución.

```
Programa 29: lee 200 números de un archivo
y calcula sus principales estadísticos

Nombre del archivo a leer: c:\temp\datos.txt
Leyendo del archivo c:\temp\datos.txt...hecho.

Estadísticos a consola:

    Media =   48.7650
  Mediana =   46.5000
     Moda =    6
 Varianza =  821.5174
 Desviac. =   28.6621
   Máximo =   99
   Mínimo =    1
    Rango =   98
     Suma = 9753
   Cuenta =  200

Nombre del archivo a escribir: c:\temp\resp.txt
Escribiendo al archivo c:\temp\resp.txt...hecho.
```

En cada ejecución las salidas son diferentes. Los valores en Excel™ son:

Media	Mediana	Moda	Varianza	Desviac.	Máximo	Mínimo	Rango	Suma	Cuenta
48.7650	**46.5000**	**6**	**821.5174**	**28.6621**	**99**	**1**	**98**	**9753**	**200**

NOVENO BLOQUE

40. La administración de nóminas es una importante tarea en el ámbito de la producción y los servicios. La CPA Santos García paga a su obrero agrícola un salario básico de 500 CUP por 40 h de trabajo semanal, más "tiempo y medio" por cada hora extra (no se consideran fracciones de hora.) Se desea codificar un programa que tome las horas totales trabajadas en el mes por sus obreros, y emita una nómina mensual de pago, sabiendo que la empresa toma cada mes como de 4 semanas. El lector debe emplear todo lo que necesite de lo estudiado hasta aquí.

    Este problema pretende que el lector desarrolle un gestor que actúe sobre una pequeña y rudimentaria base de datos, es decir, un conjunto de datos pertenecientes a un mismo contexto y almacenados sistemáticamente para su posterior uso. Esto requiere de persistencia en sus datos.

    - ✓ Persistencia: es la capacidad de “recordar” valores entre ejecuciones de un programa.

    Se presentan tres restricciones:

    - La nómina debe ir simultáneamente a consola y al archivo C:\temp\BD\nomina.txt
    - La estructura será montada en la tienda gratis, en una LSE débilmente ordenada.
    - El programa puede subir los datos desde una nominilla que está en el archivo C:\temp\BD\plantilla.txt, entrarlos manualmente o entrar varios y añadirlos a la plantilla en disco.

    Esta última restricción EODA es fuente de dificultad: lo último es más o menos trivial, pero lo primero presenta dos cuestiones de peso:

    1) ¿Con qué formato se escriben los datos en el archivo de texto? Hay múltiples formas de hacerlo. Aquí se usará el formato CSV (del inglés *Comma-Separated Values*.) Este tipo de formato se aplica sobre documentos de texto y es usado para representar datos en forma de tabla, donde las columnas (campos) se separan por comas y las filas (registros) por saltos de línea. Un archivo CSV se vería así:

```
c:\temp\BD>type plantilla.txt
Salvador,Mancuso,167
Maricusa,Avogadro,158
Daniel,Montero,170
Santos,Aldorso,163
Carlos,Avogadro,161
Fabia,Almudena,148
c:\temp\BD>
```

2) ¿Cómo se decodificarían los datos al programa? Hay varias maneras. De las tantas, una es aplicar la función **strtok()** para leer línea a línea todo el fichero, y a cada una de las líneas, desmembrarla en tokens, los cuales se pasarán a la variable correspondiente; es muy flexible. Otra de ellas —EODA más simple, adecuada y limitada— es usar la versión larga de **getline()** con las cadenas C++. Las dos formas son dadas.

Sea la estructura de la nómina algo así como:

```
struct Nomina {
   ⋮
   void subirRecords();
   ⋮
};
```

- **strtok()** es una función heredada del ANSI C y es flexible porque no tiene que saber la estructura real del registro que lee. Solamente lo lee completo, desmembrándolo en tokens. Para trabajar con C++ se requiere de pasarle la versión compatible de la cadena con el miembro **c_str()**.

```
void Nomina::subirRecords () {
    std::ifstream Lee( "C:/temp/BD/plantilla.txt" ); //  abrir la nominilla

    if ( Lee.fail() ) {
        throw ERR_A;
    } // if

    //  leer todos sus registros
    char * buf;
    char * val[3]; // tres valores de caracteres
    const char * delim = ",";
    Str linea;

    while ( Lee.good() ) {
        std::getline( Lee, linea );                 // lee una línea
        buf = const_cast<char *>( linea.c_str() ); // la pone en una cadena C
        char * tokenPtr = strtok( buf, delim );    // fija el primer token

        for ( short i = 0; tokenPtr not_eq nullptr; ++i ) {
            val[i] = tokenPtr;                      // guarda el token previo...
            tokenPtr = strtok( nullptr, delim ); // ...y busca el siguiente
        } // for

        if ( estaEnNomina( val[0], val[1] ) ) { // ¿ya está?
            continue;                           // obvialo
        } else {                                // otrosí...
            entrarRecord( val[0], val[1], atoi( val[2] ) ); // ...entra el registro
        } // if-else

    } // while
}
```

- El empleo total de **getline()** es más directo y sencillo:

```
void Nomina::subirRecords () {
    std::ifstream Lee( "C:/temp/BD/plantilla.txt" ); //  abrir la nominilla

    if ( Lee.fail() ) {
        throw ERR_A;
    } // if

    Str nombre, apellido, horas;

    while ( Lee.good() ) {
        std::getline( Lee, nombre, ',' );          // toma el primer valor
        std::getline( Lee, apellido, ',' );        // toma el segundo
        std::getline( Lee, horas );                // y toma el tercero

        if ( estaEnNomina( nombre, apellido ) ) {  // ¿ya está?
            continue;                              // obvialo
        } else {                                   // otrosí entra el registro
            entrarRecord( nombre, apellido, atoi( horas.c_str() ) );
        } // if-else
    } // while
}
```

Para los efectos del desarrollo y puesta a punto, los datos serán los abajo listados:

Trabajador	Horas
Salvador Mancuso	167
Maricusa Avogadro	158
Daniel Montero	170
Carlos Avogadro	161
Santos Aldorso	163
Fabia Almudena	148

Una salida se vería así:

```
Problema 40: Nómina de la CPA Santos García.

   MENÚ
-------------------------
1) Insertar registros
2) Cargar una nomimilla
3) Contar los registros
4) Verificar un registro
5) Editar un registro
6) Crear la nómina
7) Borrar un registro
8) Vaciar la nómina
9) Ayuda al usuario
0) Terminar
-------------------------
Entre una opción: 2
Los registros fueron insertados.

Presione una tecla para continuar . . .
```

La nómina se verá en pantalla y en disco más o menos así:

```
CPA S. Garcia - Nomina
Fecha: Sun Sep 06 08:59:59 2020

Obrero.............Horas........Salario
Aldorso, Santos.......163........2318.75  _________________
Almudena, Fabia.......148........2037.50  _________________
Avogadro, Maricusa....158........2225.00  _________________
Avogadro, Carlos......161........2281.25  _________________
Mancuso, Salvador.....167........2393.75  _________________
Montero, Daniel.......170........2450.00  _________________
Total............................13706.25$ Firma

Nómina en disco.

Presione una tecla para continuar . . .
```

17 - INTERACTUANDO CON EL DOS

> *DOS (Disk Operating System o Sistema Operativo en Disco, en español) es una familia de sistemas operativos creados originalmente para las computadoras personales de IBM. Cuenta con una interfaz de línea de comando en modo texto vía su propio intérprete de órdenes. Es suministrado con los SO Windows™. Editado de (Wikipedia en español)*

Los ejemplos del texto fueron montados a consola por la dificultad inherente de aprender los conceptos básicos de un lenguaje de computación cuando el entorno es gráfico. Para desarrollar en un entorno gráfico no sólo hay que saber los elementos que forman el núcleo del lenguaje y sus rutinas de E/S (que precisamente aquí es el súper objetivo), también hay que conocer qué son, para qué sirven y cuándo y cómo se usan los componentes gráficos, tales como las cajas de entrada (*input boxes* en inglés), de diálogo (*dialog boxes* en inglés), etiquetas de texto (*text labels* en inglés), etc.

El entorno gráfico presenta muchísimos distractores que no lo hacen ideal como para asimilar un conocimiento sólido de las bases sobre las que se asienta un lenguaje como es C++.

Según la (Wikipedia en español), el DOS (MS-DOS) fue el miembro más conocido de la familia de sistemas operativos de las compañías IBM y Microsoft, y el principal sistema para computadoras personales desde la década de 1980, hasta mediados de los ‘90, cuando fue sustituida gradualmente por varias generaciones de Windows™, pero aún está en uso para casos muy puntuales.

Los programas desarrollados hasta el momento han sido soportados sobre un emulador del DOS, que ahora viene integrado con Windows™ y existen varias formas de intercomunicación. Por omisión, los flujos típicos de datos de C++ se usan para comunicarse con la consola, sin embargo, en los SO que soportan la redirección (DOS, Unix, Linux, etc.), las operaciones de E/S pueden ser dirigidas a otros dispositivos, incluyendo archivos, impresoras, plotters, etc.

En este caso, el IDE se monta en Windows y los proyectos por consola se ejecutan en una ventana auxiliar donde actúa el emulador. Los programas en este acápite —aunque pueden ser ejecutados desde el IDE— son diseñados para serlos desde dentro del DOS, usando sus comandos de línea, y es ahí donde son visualmente atractivos.

En este apartado se verán dos temas:

1. Cómo interactuar con el SO en la línea de comandos.
2. Cómo direccionar la salida desde el programa.

En la línea de comandos

Según ISO/ANSI C++11, solamente dos definiciones de la función **`main()`** son totalmente portables:

```
1.  int main()
2.  int main(int argc, char *argv[])
```

y cualquier otra forma compromete seriamente la portabilidad. El primero (**argc**: ***arg****ument* ***c****ount* en inglés) es el número de argumentos en la línea de comandos, y el segundo (***argv[]**: ***arg****ument* ***v****alues* en inglés) es un puntero a un arreglo ANSI C de caracteres que alberga los nombres de los argumentos dados en dicha línea. A esta forma, se le inserta también **return EXIT_SUCCESS** para una salida exitosa y si se prevé una salida fallida, el programador puede poner **return EXIT_FAILURE**, constantes definidas en la biblioteca **<cstdlib>**.

El primer elemento de ese arreglo (**argv[0]**) es siempre el nombre completo del programa, incluyendo su camino, aunque cada IDE puede interpretarlo a su modo. Los elementos que le siguen son los parámetros expresados como nombres en la línea de comandos, en el orden dado: **argv[1]** es el primero, **argv[2]** es el segundo, etc.

Es costumbre disponer de una brevísima ayuda al usuario, que se muestra cuando el programa es llamado erróneamente desde el DOS.

Invocando argumentos en el IDE

¿Cómo enviar los argumentos necesarios al programa? Cada IDE tendrá su forma propia, aunque todos usarán los argumentos (**argv**) y su cuenta (**argc**.) De nuevo aparece el caso en que no queda otra: hay que consultar el manual.

Programa 30: DOS – parámetros en línea

Se pide modificar el programa 29 para que acepte parámetros en la línea de comandos. Debe tomar los datos del fichero de entrada, calcular sus estadísticos y poner la respuesta en el fichero de salida. En estos casos, y cuando haga falta un argumento de entrada por **main()**, se va a **Project->Set project's arguments…**

Clicar donde muestra la figura. Los puntos suspensivos indican que sigue una caja de diálogos.

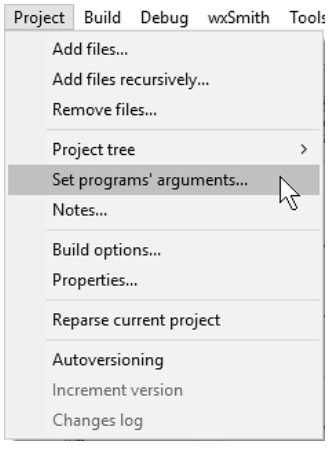


Ilustración 18. Argumentos del programa

Aunque puede probarse para su desarrollo y puesta a punto, finalmente lo correcto es llamarlo desde el DOS: **programa30 datos.txt resp.txt**, donde en **datos.txt** se pusieron los valores generados anteriormente por el programa 28. Los módulos del manejo del archivo, de estadística y la función de comparación se incluyen desde el programa 29. Desde el programa principal se escribe al archivo especificado en la línea de comandos.

Para este programa los dos argumentos necesarios son los nombres completos de los ficheros de lectura de datos y de escritura de información, como muestra la figura que sigue:

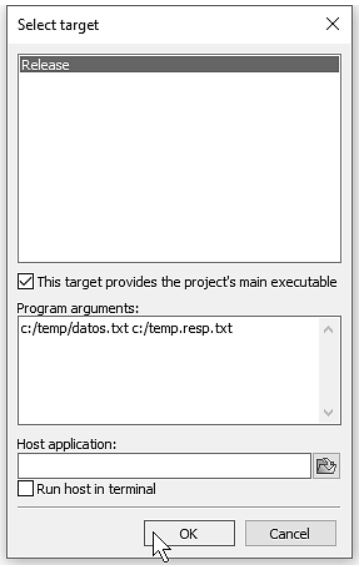


Ilustración 19. Poniendo los argumentos

Programa

```
// Programa p30: parámetros en la línea de comandos.
#include "../p29-leyendo/entrada.h"
#include "../p29-leyendo/estads.h"
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <iomanip>

int main( int argc, char *argv[] ) {
  if ( argc not_eq 3 ) {
    std::cerr <<
    "Uso: p30 entrada.txt salida.txt\n"
    "  entrada: nombre del fichero con los datos.\n"
    "  salida : nombre del fichero con la informaci\242n.\n"
    "  Si el fichero de salida existe, se sobrescribe; si no, se crea.\n";
    return EXIT_FAILURE;
  } // if

  // info
  std::cout << "Programa 30: interactuando con el DOS\n\n"
  "Lee los n\243meros del archivo \"" << argv[1] << "\", "
  "calcula sus principales\nestad\241sticos y los pone en "
  "el archivo \"" << argv[2] << "\"\n\n";

  // nombre de los archivos
  std::string archy1 = argv[1]; // de datos
  std::string archy2 = argv[2]; // de respuesta
  int n;

  //  la entrada
  try {
```

```
    n = contarDatos(archy1);
  } catch (Str) {
    std::cout << ERR;
    return EXIT_FAILURE;
  } // try-catch

  std::cout << "Leyendo de " << archy1 << "..."; // trayendo los datos al arreglo
  double *pAry = new double[n];                  // los valores en la tienda gratis
  leerArchivo(archy1, pAry, n);                  // llenando al arreglo
  qsort(pAry, n, sizeof(double), compara);       // ordenando el arreglo
  std::cout << "hecho.\nEscribiendo en "<< archy2 << "...";
  std::ofstream Escribiendo(archy2);

  try {
    if ( Escribiendo.fail() ) {
      throw ERR;
    } // if
  } catch (Str) {
    std::cout << ERR;
    return EXIT_FAILURE;
  } // try-catch

  std::cout << "hecho.\n";

  //  fin
  delete [] pAry;
  return EXIT_SUCCESS;
}
```

Salida desde el DOS

```
c:\temp>p30
Programa 30: interactuando con el DOS
Uso: p30 entrada.txt salida.txt
  entrada: nombre del fichero con los datos.
  salida : nombre del fichero con la información.
  Si el fichero de salida existe, se sobrescribe; si no, se crea.

c:\temp>p30 datos.txt resp.txt
Programa 30: interactuando con el DOS
Lee los números del archivo "datos.txt", calcula sus principales
estadísticos y los pone en el archivo "resp.txt"

Leyendo de datos.txt...hecho.
Escribiendo en resp.txt...hecho.

c:\temp>type resp.txt
Programa 30: interactuando con el DOS
Media    =    50.4250
Mediana  =    49.0000
Moda     =     2
Varianza =   878.9592
Desviac. =    29.6472
Máximo   =   100
Mínimo   =     1
Rango    =    99
Suma     = 10085
Cuenta   =   200
```

Direccionando las operaciones de E/S

Otra forma de manejar estas operaciones desde el mismo SO es direccionándolas en la línea de comandos, por lo que la cuenta de los argumentos aquí no sirve. Por ejemplo, en el programa que sigue, se aplica esta técnica que también debe ser hecha desde el mismo DOS, para que se vea cabalmente. Para probarlo, se tomaron los 20 primeros valores generados por el problema 28 y se copiaron al archivo muestra.txt. Ahora basta llamarlo desde el DOS: **`p31 <muestra.txt >resp.txt`**

- El signo **<** direcciona los datos desde un archivo al programa.
- El signo **>** direcciona la respuesta desde el programa a un archivo.

El programa toma la entrada desde un archivo, la procesa y dirige la salida a otro archivo. Si no se direcciona la entrada, la toma directamente del teclado, pero se advierte que la salida pierde el formato y no se ve correctamente. Si se omite la salida, va directamente a consola.

Este tipo de programa se conoce como utilitario *(utility*, en inglés) y generalmente forma parte de una familia de programas auxiliares para ayudar a trabajar mayormente en los SO tipo Unix o DOS. Casi siempre son monolíticos, pues se destinan a una única y muy simple tarea, y así se hará con este.

Programa 31: DOS – direccionando archivos

Tomar los 20 primeros números generados por el programa 28 y ponerlos en un archivo llamado muestra.txt. Codificar un programa que lea los números del archivo y muestre a consola, línea por línea los siguientes cálculos: el número de línea, el valor entrante, la suma acumulada, el promedio y los valores extremos hasta esa línea.

Programa

```
// Programa 31
#include <iostream>
#include <cstdlib>
#include <iomanip>

int main() {
  // encabezado
  std::cout
      << std::right
      << "Programa 31: algunos estad\241sticos.\n\n"
      << std::setw( 3 ) << "No."
      << std::setw( 8 ) << "Valor"
      << std::setw( 8 ) << "M\241n."
      << std::setw( 8 ) << "M\240x."
      << std::setw( 8 ) << "Rango"
      << std::setw( 8 ) << "Suma"
      << std::setw( 8 ) << "Prom." << '\n'
      << std::setw( 52 ) << std::setfill ( '-' ) << '\n';
  std::cout << std::setfill ( ' ' );
  // valores
  int valor, minimo, maximo, suma, rango;
  minimo = 65536; // número 'mágico' - ver Programa 5
  maximo = suma = valor = rango = 0;
  double promedio = 0;

  //  ciclo de lectura
  int i = 1; // índice
```

```
  while ( std::cin >> valor ) {
    //  manejo de valores
    if ( valor < minimo ) {
      minimo = valor;
    } // if

    if ( valor > maximo ) {
      maximo = valor;
    } // if

    suma += valor;
    rango = maximo - minimo;
    promedio = static_cast<double>( suma ) / i;

    //  saca la línea i-ésima
    std::cout
    << std::fixed << std::setprecision( 0 )
    << std::setw( 3 ) << i
    << std::setw( 8 ) << valor
    << std::setw( 8 ) << minimo
    << std::setw( 8 ) << maximo
    << std::setw( 8 ) << rango
    << std::setw( 8 ) << suma << std::setprecision( 2 )
    << std::setw( 8 ) << promedio << '\n';

    ++i; // avanza a la próxima línea
  } // while

  // fin
  return EXIT_SUCCESS;
}
```

Salida

```
c:\temp>p31 <muestra.txt
Programa 31: algunos estadísticos.

No.   Valor    Mín.    Máx.   Rango    Suma   Prom.
-------------------------------------------------
  1       8       8       8       0       8    8.00
  2       6       6       8       2      14    7.00
  3      33       6      33      27      47   15.67
  4      13       6      33      27      60   15.00
  5      59       6      59      53     119   23.80
  6      11       6      59      53     130   21.67
  7      99       6      99      93     229   32.71
  8      96       6      99      93     325   40.62
  9      91       6      99      93     416   46.22
 10      83       6      99      93     499   49.90
 11      72       6      99      93     571   51.91
 12      42       6      99      93     613   51.08
 13      24       6      99      93     637   49.00
 14      24       6      99      93     661   47.21
 15      66       6      99      93     727   48.47
 16      71       6      99      93     798   49.88
 17      93       6      99      93     891   52.41
 18       9       6      99      93     900   50.00
 19      73       6      99      93     973   51.21
 20      62       6      99      93    1035   51.75
```

BIBLIOGRAFÍA

Deitel, H. M., & Deitel, P. J. (1995). Como programar en C/C++. *Vol. 2*(Segunda ed.). Naucalpan de Juárez, Estado de México: Prentice Hall. Recuperado el 2018

Eckel, B. (2000). Pensar en C++. *I*. Madrid, España: McGraw-Hill Interamericana de España SAU. Recuperado el 2018

Gottfried, B. S. (s/f). *Programación en Pascal.* (Schaum, Ed.) USA: Serie de Compendios.

Halterman, R. L. (2018). Fundamentals of C++ programming. USA. Recuperado el 2018

Jaeschke, R. (feb de 1990). Header Design and Management. *C/C++ Users Journal, VIII*(2). Recuperado el 2018

Jaeschke, R. (1990, march). Void Pointers. *C++ User Journal, VIII*(3). Retrieved marzo 2018

Jaeschke, R. (CUJ, abril 1991). Data Structures, Part I. *C++ User Journal, IX*(4). Retrieved abril 2018

Jensen, T. (2003). Tutorial sobre Apuntadores y Arreglos en C. (M. Barquerizo, Ed.) Recuperado el 2018, de http://www.netcom.com/~tjensen/ptr/cpoint.htm

Kohl, N., & al. (2004, october). C/C++ Reference. USA: cppreference. Retrieved 2018

Linux Tricks and Tips. (2020). Gales, Gran Bretaña: Black Dog Media. Recuperado el mayo de 2020, de http://www.dbmpublications.com.

Lipschutz, S. (1991). *Estructura de Datos.* Habana, Habana, Cuba: Ediciones R.

Lischner, R. (2003, may). C++ in a Nutshell. USA: O'Reilly. Retrieved 2018

Marzal, A., & Gracia, I. (s/f). Introducción a la programación con C. (D. d. Informáticos, Ed.) Castellón de la Plana, Castellón, España: Universidad de Castellón. Recuperado el 2018

Microsoft Co. (s/f). *Microsoft Quick C compiler.* Seatle: Run-Time Library Reference.

Prata, S. (2012). *C++ Primer Plus* (Sexta ed.). Upper Saddle River, New Jersey, USA: Pearson Education, Inc. Retrieved 2019

Prata, S. (2014). *C Primer Plus* (Sexta ed.). Upper Saddle River, New Jersey, USA: Pearson Education, Inc. Retrieved 2018

Pugh, K. (abril de 1990). Questions & Answers. *Announcing The Great Name / Obscure Code Contest, VIII*. USA: Miller Freeman, Inc. Recuperado el 2019

s/a. (s/f). *Lenguaje C. Biblioteca de funciones.* La Habana, Cuba: Ediciones R.

Saks, D. (1990, january). Writing Standard Headers. *C++ User Journal, VIII*(1). Retrieved 2018

Schildt, H. (1995). C++. Guía de Autoenseñanza. (Primera ed.). Madrid, España: McGraw-Hill Interamericana de España. Recuperado el 2018

Schildt, H. (2003). *C/C++ Programmer's Reference.* Retrieved 2018

Sutter, H. (2001). Exceptional C++. 240. Indianápolis, IN, USA: Addison Wesley Longman, Inc. Retrieved 2018

Sutter, H. (2001). More Exceptional C++. (I. Addison Wesley Longman, Ed.) Indianápolis, IN, USA. Recuperado el 2019

Wikipedia en español. (s.f.).

Wikipedia en español. (2016). *https://es.wikipedia.org/*, Español. Recuperado el febrero de 2018

CONTRAPORTADA

El autor, Ing. Simón Adolfo Galliano Vidal, jubilado de la Universidad Médica de Granma "Celia Sánchez Manduley" con el cargo de Jefe de Informática del Nodo Provincial de Infomed, pone a disposición del amable lector este libro, primero de una serie de tres.

Su objetivo aquí es enseñar a codificar en C++ desde el paradigma de la programación imperativa, *sin necesidad de precondición alguna y como un súper C*, que el autor define cuando el programador usa un compilador C++ y sus características propias, pero crea su código como si estuviera programando en ANSI C, sin recurrir a la POO clásica, o directamente a la biblioteca estándar de plantillas.

Se presentan 17 temas distribuidos en nueve bloques. Los temas 1 (Sistema de desarrollo) y 2 (Un texto introductorio) son de carácter introductorio. El bloque uno comprende los temas 3 (El primer programa), 4 (Entrando números) y 5 (Entrando cadenas). El segundo bloque abarca los temas 6 (Valores límites), 7 (Decisiones y ciclos) y 8 (Ciclos y decisiones). El tercero engloba los temas 9 (Funciones) y 10 (compilación separada). El cuarto bloque incluye al tema 11 (Tipos estructurados de datos). El bloque cinco contiene al tema 12 (Punteros). El sexto bloque se dedica al tema 13 (Manejo dinámico de la memoria.)

El resto es sobre aplicaciones específicas. El séptimo bloque abarca el tema 14, dedicado a las listas simplemente enlazadas; el octavo comprende al tema 15, sobre el ordenamiento y búsqueda de datos, y el noveno al tema 16, acerca del manejo básico de archivos. Por último, el tema 17 no propone ejercicios, ya que se destina a introducir un conocimiento que, aunque no imprescindible, si es complementario: una forma fácil de interactuar con el DOS.

El texto se completa con 19 ilustraciones aclaratorias, 33 tablas para consultas, 6 programas auxiliares completos, 31 programas resueltos y 40 ejercicios propuestos, cuidadosamente diseñados para que el estudioso lector se apoye en conocimientos previos de forma que, por ejemplo, para resolver cualquier ejercicio propuesto en el tercer bloque, sólo requiera de lo previamente aprendido hasta ese punto.

Se da la sintaxis coloreada oficial del C++ y la salida es tal cual se ve.

Se adjuntan las soluciones del autor a los programas y ejercicios planteados. Para casi cualquiera de ellos hay más de una, si no mejor, al menos diferente. Si no está satisfecho, el curioso lector debe buscar una personal, que es una de las mejores formas de aprender con alegría y eficacia: ¡mediante la experimentación propia!

Se puede contactar al autor por e-mail en: mailto:simongv@infomed.sld.cu?subject=texto de Cpp para cualquier queja, consulta o sugerencia.

Muchas gracias.