Series HOMEPET

Introducción al C++

Tercera Parte: un C con mucha clase

Ing. Simón Galliano Vidal, ret.

11 Temas
12 Ilustraciones
10 Tablas
 3 Programas auxiliares
14 Programas resueltos
15 Ejercicios propuestos
 5 Apéndices

noviembre 2020

Publicado online por El Archivo Histórico de la Ciudad de Manzanillo

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

Fecha de publicación: 19 de noviembre de 2020

Versión: 2.2C.1020

Autor: S. GALLIANO - 2019

Publicista: Lic. Delio Orozco.

ISBN: 978-959-7179-67-2(3)

Páginas: 107

CONTENIDOS

ILUSTRACIONES 6

TABLAS 6

1 - SISTEMA DE DESARROLLO 8
- Objetivos del texto 9
- Una palabra sobre los ejercicios 9
- ¿Cómo escribir buen código? 10
  - La Filosofía del Zen de Python 10
- Caracteres en español 11
- Inclusión de ficheros 11

2-FILOSOFÍA DE LA OOP 11
- Características principales 12
  - Pensar/trabajar orientado a objetos 12
    - Abstracción 12
    - Ocultación de información 13
    - Encapsulación 13
    - Herencia 14
    - Polimorfismo 14
- Protección y seguridad de los datos 15

3-BREVÍSIMA HISTORIA DE LA OOP 15
- Paradigmas de programación 16
  - Programación modular 16
  - Programación orientada a objetos 17

4-DEFINICIONES BÁSICAS 17

5-ESPACIOS DE NOMBRES 19
- ¿Qué es el espacio de nombres? 20
- ¿Cómo se implementa? 20

6-UNA CLASE BÁSICA 21
- Programa 1-Un producto alimentario 21
  - Análisis y diseño 21
  - Eficiencia 22
  - Constructor 22
  - Destructor 23
  - Los métodos inline 23
  - Lista de inicialización 23
    - Nuevas formas de inicializar 24
  - Declaración 24
  - Definición 25
    - Accesores de miembros 26
    - Métodos 26
    - Un par de errores 26
  - Uso 27
  - Salida 28

7-UNA CLASE COMPUESTA 28
Categorías de los métodos 29
Métodos especiales automáticos 29
Programa 2-Una cafetería 30
Análisis y diseño 31
Declaraciones 31
Definiciones 32
Uso 35
Una posible salida 35

8-UNA CLASE UTILITARIA 36
Programa 3-Un cronógrafo 36
Diseño del Cronógrafo 37
La estructura de datos 37
Declaración y definiciones 37
Uso 38
Salida 40

9-SOBRECARGA DE OPERADORES 41
Sobrecarga de operadores 41
Sintaxis 41
Limitaciones 42
Funciones amigas 43
Referencias 43
Una clase para un vector geométrico 44
Programa 4-Una clase Vector 44
Declaración 45
Definiciones 46
Uso 48
Salida 49

10-TEMAS DE LA HERENCIA 49
Herencia simple 49
¿Cómo se deriva una clase? 50
Una clase hereda su constructor de copia 51
Una clase hereda su operador de direcciones 51
¿Qué no se hereda de una clase? 51
Programa 5-desarrollo por herencia pública 51
Declaración 51
Definición 52
Uso 53
Salida 54
Polimorfismo 54
Clases virtuales 54
Constructores y destructores 54
Sobrescritura y sobrecarga de métodos 55
Clase virtual pura 55
Programa 6-polimorfismo 56
Declaración 56
Definición 57

Uso 58
Salida 58
Aplicación del polimorfismo 58
Costos asociados 59
Una advertencia 59
Herencia múltiple 59
Programa 7-la herencia múltiple 59
Declaraciones 60
Definiciones 61
Uso 62
Salida 62
Ambigüedades: mismo método 62
Ambigüedades: clase en común 63
La herencia en diamante 63
Programa 8-la clase en común 64
Definiciones 64
Declaraciones 65
Uso 66
Salida 66
Programa 9-polimorfismo y herencia múltiple 67
Declaración 67
Definición 68
Uso 68
Salida 69
Programa 10-una BD polimórfica 69
Declaraciones 70
Definición 71
Uso 72
Salida 73

11-PROGRAMACIÓN GENÉRICA 73
Plantillas de clases 74
Una clase 74
Programa 11-una clase para plantilla 74
Salida 75
Una plantilla de clase 75
Programa 12-la plantilla de la clase 75
Definición y declaración 76
Uso 76
Salida 77
Functores 78
Functores predefinidos 78
¿Para qué usarlos? 78
Programa 13-una clase functor 79
Declaración 79
Definiciones 80
Uso 80
Salida 81
Programa 14-la plantilla 81

Declaración....81
Uso....82
Salida....84

PROBLEMAS PROPUESTOS....85

APÉNDICES....93
1-Organizaciones normativas para C++....93
2-Bibliotecas de terceras personas....94
Bibliotecas BOOST....94
Bibliotecas Blitz++....94
STLport....94
3-El fichero makefile....95
4- Métodos y Miembros Estáticos....98
Programa auxiliar 1-miembro estático público....98
Definición y declaración....98
Uso....99
Salida....99
Programa auxiliar 2-método público....100
Definición y declaración....100
Uso....100
Salida....101
5-Manejo de errores....102
Bloque try-catch....102
Programa auxiliar 3-división por cero....103
Declaración....103
Definición....104
Uso....104
Salida....105
¿Cuál es la ganancia?....105

BIBLIOGRAFÍA....106

CONTRAPORTADA....107

ILUSTRACIONES

Ilustración 1. Opciones básicas de inicio 8
Ilustración 2. Programación modular 16
Ilustración 3. Programación orientada a objetos 17
Ilustración 4. La eficiencia de un programa 22
Ilustración 5. La clase compuesta 31
Ilustración 6. Componentes de un Vector 44
Ilustración 7. Una jerarquía geométrica 50
Ilustración 8. Partes de un objeto Pegaso 60
Ilustración 9. Una clase común 63
Ilustración 10. Una clase común 64
Ilustración 11. La base de datos 70
Ilustración 12. Clases de error 102

TABLAS

Tabla 1. Códigos de escape en formato octal para el idioma español 11
Tabla 2. Cualificadores de clases 14
Tabla 3.Precios de alimentos 21
Tabla 4. Métodos automáticos 30
Tabla 5. (simplificada) Operadores que no pueden sobrecargarse 42
Tabla 6. (simplificada) Operadores que pueden sobrecargarse de ambas formas 42
Tabla 7. Functores predefinidos 78
Tabla 8. Normas norteamericanas que afectan a C++ 93
Tabla 9. Normas internacionales relacionadas con C++ 93
Tabla 10 (simplificada) Los símbolos del make 95

Tercera Parte: un C con mucha clase

La potencia de la OOP

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 IDE multiplataforma para el desarrollo de programas en el lenguaje ISO C++, creado en C++ y liberado bajo la Licencia pública general de GNU. (Wikipedia en español)*

Para esta tercera 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.
- El manejo de su entorno laboral es relativamente estable, cómodo, eficiente y flexible.
- Tiene opciones para crear programas de consola (*Console application*, en inglés.)

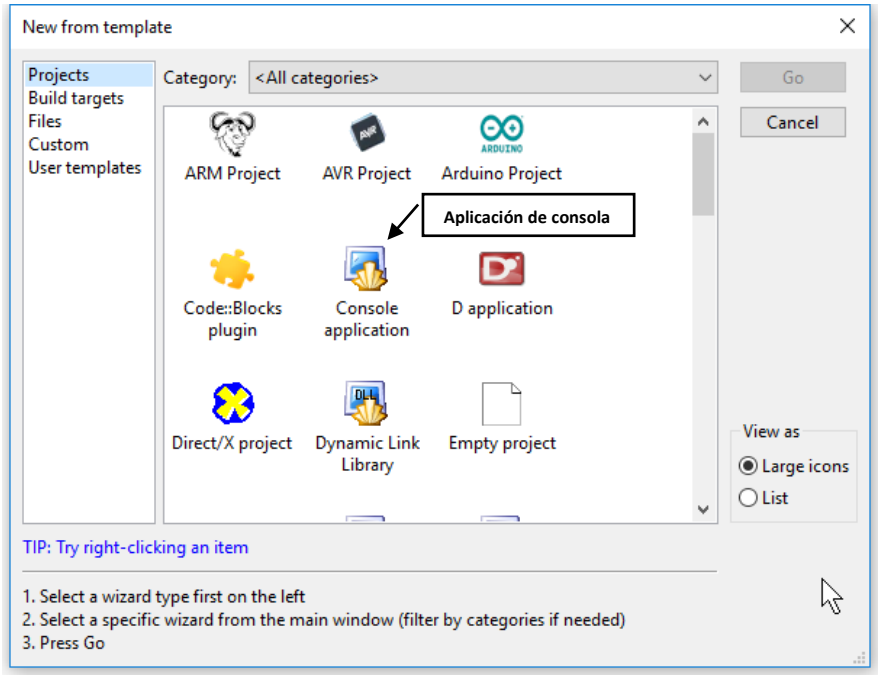


Ilustración 1. Opciones básicas de inicio

Todos los programas del texto toman como opción *Console application*

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!

En Microsoft Windows™ y bajo este IDE 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");`**

Objetivos del texto

En un nivel medio, asumiendo que el lector sabe programar en ISO C++[1] y manejar con alguna soltura la STL (*Standard Template Library*), precondición alcanzable entre otras cosas por haber vencido los dos tomos anteriores de esta serie, el objetivo general de este tercer y último tomo es enseñar las buenas prácticas básicas que deben usarse para lograr crear clases correctas. De paso aprenderá a utilizar el polimorfismo y conocerá lo elemental de la programación genérica de clases y de la creación de functores.

El objetivo particular de cada tema básico es que, al ir enseñando las buenas prácticas, se cubra un conjunto de conceptos que le sean asociados a ese paso en particular, demostrando la ganancia en poder de abstracción, facilidad y seguridad de codificación que un programador, ya sea novel o de experiencia estándar, puede obtener con algo de voluntad y un mínimo de esfuerzo al usar la OOP (*Object Oriented Programming*) y la STL combinadas, como base de su trabajo.

Sepa el amable lector que los programas dados, problemas propuestos y fragmentos de código aquí proporcionados fueron adaptados a los objetivos pedagógicos del texto y todos fueron comprobados por el autor. Dispense pues cualquier gazapo que aparezca.

Una palabra sobre los ejercicios

Partiendo de que el lector sabe programar en ISO C++, los ejercicios fueron cuidadosamente diseñados para que se apoyara en conocimientos previos obtenidos durante el estudio de los tomos anteriores de esta serie, pero no es un requisito indispensable. Si no los conoce, puede usar cualquier texto que trate la enseñanza del C++.

El texto despliega once temas: este primero se dedica a presentar el sistema de desarrollo, con un estilo de programación típica de ANSI C, pero conservando la claridad de lectura humana del código. El segundo tema revisa brevemente la filosofía de aplicación de la OOP y el tercero traza una muy breve historia de la misma. El cuarto tema revisa la jerga de los programadores de la OOP y como se ve en C++, y el quinto introduce el uso los espacios de nombres.

Hasta aquí los temas son mayoritariamente teóricos

El sexto tema va a la construcción correcta de una clase y el séptimo produce una clase compuesta a partir de la anterior. El octavo tema crea una clase utilitaria y el noveno enseña cómo hacer la sobrecarga de operadores. Luego viene el tema diez de la herencia dividido en cuatro partes: la herencia simple, el polimorfismo, la herencia múltiple y el polimorfismo junto a la herencia múltiple. Cierra el texto el tema once dividido en dos partes: la programación genérica de plantillas de clases y la programación genérica de functores.

Lleva cinco apéndices que son: (1) Las organizaciones normativas de C++; (2) Las bibliotecas BOOST, Blitz++ y STLport; (3) Un fichero **`makefile`** y cómo se compila desde la línea de comandos del DOS; (4) Métodos y Miembros estáticos, que tiene dos programas auxiliares; y (5) Manejo de errores, el cual a su vez tiene un programa auxiliar.

El texto se complementa con 12 ilustraciones aclaratorias, 10 tablas para consultas, 14 programas resueltos y concluye con 15 ejercicios propuestos. Se adjuntan los programas y las soluciones a los ejercicios planteados.

[1] ISO C++: Es la forma correcta de llamar al lenguaje inventado por Bjarne Stroustrup y normalizado por el comité conjunto ANSI/ISO que en su momento fuera denominado X3J16/WG21, pero el texto usa C++, intercambiable y con preferencia.

¿Cómo escribir buen código?

La elegancia en la escritura y la claridad en la lectura de un programa no es un asunto de gustos: ¡es una necesidad del oficio. Ver más adelante el Zen de Python, que entre otras cosas dice que la legibilidad cuenta. (Introducción al C++, tomo 1, pág. 15) Recordar que un buen código es aquél que posee **(a)** las características del Zen de Python; **(b)** está correctamente sangrado con estilo consistente a lo largo del trabajo; y **(c)** además posee las cualidades de ser:

1. Íntegro, pues da respuestas correctas a los cálculos hechos sobre los datos suministrados.
2. Preciso, ya que los valores calculados responden a las ecuaciones matemáticas usadas.
3. Fiable, porque hace lo planeado durante su ciclo entero de vida.
4. Claro, pues es legible por un programador afín.
5. Modular, al estar sus partes constituyentes auto contenidas en funciones/clases C++ lo mejor formadas posible.
6. Simple, ya que es modularmente corto, sencillo y entendible.
7. General, a causa de estar hecho en base a lineamientos que responden al entorno del problema.
8. Robusto, si es resistente a todo tipo de errores.
9. Tipificado, si estandariza la forma de su uso, la toma de los datos y la presentación de los resultados obtenidos.
10. Portable, ya que es ejecutable en diferentes plataformas con un mínimo de cambios, o ninguno.
11. Eficiente, pues costó menos recursos que lo planeado.
12. Eficaz, porque está descrito por sus usuarios al menos como bueno.

La Filosofía del Zen de Python

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

Un código mientras más posea de estas condiciones, mejor cumple con ser bueno

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. Cuando un escape presenta tres números, C++ los interpreta en formato octal. En el texto se emplean los siguientes códigos embebidos en las cadenas de los programas:

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

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

Inclusión de ficheros

La directiva **`#include`** dice al preprocesador que abra el fichero indicado e incluya su contenido en ese mismo lugar, por eso se tratarán de incluir los ficheros en el lugar adecuado siempre que se pueda, puesto que C++, para usar algo no contenido en su núcleo básico (por ejemplo, un componente auxiliar o una facilidad), debe "conocerlo" primero.

El orden de inclusión correcto es:

1. Los ficheros de cabecera hechos por el programador.
2. Si los hay, los que sean incluidos directamente por el IDE en uso (Code::Block no pone.)
3. Las bibliotecas de terceros, tales como STLPort, Boost o Blitz++ (el texto no las usa.)
4. Los de ANSI C (el texto tampoco los usa), pero siempre recordando que no deberían mezclarse con los del C++, ni usarse en código nuevo.

Los ficheros creados por el programador se incluyen entre comillas; los otros entre paréntesis angulares

2-FILOSOFÍA DE LA OOP

> *No todos están de acuerdo en lo que significa la orientación a objetos…aunque casi todos convendrán que entre sus conceptos básicos están la herencia y el polimorfismo. Otra mayoría pondrá en escena la encapsulación, algunos otros incluirán el manejo de excepciones y nadie considerará incluir la programación de plantillas. El punto aquí es que existen opiniones diversas al seleccionar una característica dada y cada punto de vista posee apasionados defensores. (Sutter, Exceptional C++, pág. Item 25)*

La OOP es una *metodología de programación* que viene a innovar la forma de obtener resultados en aplicaciones muy grandes y/o complejas. Es considerada un *paradigma* porque amplió sustancialmente el espectro de problemas a solucionar con ayuda de la computadora, debido a que entrega una buena facilidad de manejo de las complejidades, pero no es de obligatorio cumplimiento en C++.

- ✓ Metodología es un conjunto de métodos aplicables a un área determinada de las ciencias…Aunque los métodos están allí, en los lenguajes híbridos como el C++, la metodología dicta en cada caso cuáles son los escogidos y cómo se aplican, pero no obliga su imposición.
- ✓ Paradigma: el historiador Thomas Kuhn extendió la definición de la palabra para abarcar un conjunto de teorías, estándares y métodos, que juntos representan una forma de organizar el conocimiento, esto es, una forma de ver el mundo. (Budd, pág. 3)

Características principales

¿Cuáles son las principales características de la OOP? ¿Cuáles la describen mejor? EODA[2], antes de embarcarse en la aventura de describirlas, se debe precisar cómo se orienta el pensamiento hacia los objetos. En otras palabras, ¿qué significa pensar o trabajar orientado a objetos?

Pensar/trabajar orientado a objetos

Significa en primer lugar, aplicar la *abstracción* frente a un problema, para delimitar las fronteras donde se moverán los datos y sus acciones. Una vez hecho esto, se podrán diseñar correctamente las clases para que trabajen de conjunto y emitan mensajes que vayan solucionando el problema.

Estos mensajes van del cliente al servidor: el cliente pide y el servidor cumple, pero el cliente no le dice al servidor cómo hacer la tarea: solamente especifica qué tarea se requiere en ese instante, lo cual refuerza el principio de *ocultación de la información*. Por otra parte, un mismo mensaje dirigido a un servidor específico (una clase dada) tendrá un resultado, y dirigido a otro, resultará en algo diferente: eso es *polimorfismo*.

La abstracción, el polimorfismo y la ocultación de información *son conceptos generales*, sus implementaciones *son técnicas de codificación* que le pertenecen a un lenguaje dado. Por ejemplo, cada una de ellas se obtendrá de una manera en Smalltalk, de otra en Delphi, y aún de otra en C++.

Los lenguajes propios de la OOP y los híbridos, como son Delphi y C++, presentan ciertas características básicas, comunes, y las principales son la abstracción, la herencia, el polimorfismo, la ocultación de la información y el aislamiento de los datos.

Abstracción

La abstracción es la propiedad de formar la noción de una clase extrayendo de la entidad real los rasgos mínimamente esenciales al problema bajo estudio. Es fundamental en la creación de clases. El diseñador/programador reúne todos los elementos que pueden considerarse pertenecientes a una entidad bajo el punto de vista del problema dado. Esto permite aumentar el diseño estructurado de los componentes del sistema.

La abstracción encarada desde el punto de vista de la OOP expresa las características esenciales de un objeto, o sea, lo que lo distingue de lo demás, proveyendo además los límites conceptuales adecuados. Si a un objeto se le dan más características de las necesarias, resultará difícil de comprender, construir, usar y modificar. Si se le dan menos, es inservible.

La realidad es muy complicada para ser captada totalmente, así que cada jerarquía C++ es una representación arbitraria de la información disponible. El truco del buen diseño es representar las áreas de interés de forma que se mapee a la realidad de manera razonablemente fiel. (Wiegand, pág. Dia 12. Inheritance) La abstracción bien aplicada genera una ilusión de simplicidad dado que minimiza (y a la vez, optimiza) la cantidad de características que definen a un objeto. Editado de (Wikipedia en español)

Según Grady Booch (Object-Oriented Analysis and Design with Applications, Second Edition, Benjamin-Cummings, 1994, citado por Chuck Allison en el C/C++ User Journal de mayo/1995), la abstracción aísla las características básicas, esenciales, que distinguen un objeto de cualquier otro, permitiendo disponer de fronteras muy precisas y muy bien definidas, ya que

[2] EODA: en opinión del autor.

ha sido demostrado que el intelecto humano solamente puede manejar una cantidad limitada de complejidad, y para gestionar un sistema muy complejo es imperativo enfocarse en lo esencial según el contexto, e ignorar el resto.

Algunos profesionales confunden este concepto con el principio de ocultación de la información, principalmente porque suelen emplearse de conjunto.

La abstracción es clave y puntal en el proceso de análisis y diseño orientado a objetos, pues mediante ella y sólo ella, se puede armar un conjunto de clases que permitan modelar el problema que se quiere solucionar. Operativamente, la abstracción al aislar las características esenciales de un objeto, de paso las revela y además permite integrarle sus comportamientos.

El modelo abstracto de un ente real, objetivo, que responde a un problema, es una clase que instanciada en un objeto puede realizar acciones tales como informar, cambiar su estado y comunicarse con otros objetos en el sistema, sin revelar cómo lo hace. Evidentemente, procesos, funciones y métodos también pueden ser abstraídos.

C++ no obliga a la buena abstracción; eso lo debe de saber hacer el programador

Ocultación de información

La 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. Esto supone proporcionar una interfaz estable que proteja el resto del programa de la implementación, que es lo que cambia. En los lenguajes OOP este principio se manifiesta como la encapsulación aplicada en la clase. Editado de (Wikipedia en español)

La ocultación de información es una *metodología de trabajo* que proviene del paradigma de la programación modular y como tal, el programador la aplica si quiere o sabe hacerlo. Se concreta mediante el *principio del privilegio mínimo* de Parnas, que para las clases se define como:

- La clase tiene acceso a la información que necesita para hacer su tarea, *y nada más*.
- Sus métodos tienen acceso a la información que necesitan para hacer sus tareas, *y nada más*.
- Su desarrollador debe manejar en sus implementaciones toda la información que se necesita para que hagan su tarea, *y nada más*.

C++ no obliga la ocultación de información; eso lo debe de saber hacer el programador

Encapsulación

La encapsulación o aislamiento es una técnica que implementa la ocultación de información aplicada a la clase. Esto supone proporcionar una interfaz estable que proteja la mayor parte del código respecto a los cambios hechos en algunas de las implementaciones de sus componentes. Editado de (Wikipedia en español)

La encapsulación es una *metodología de trabajo* que protege las propiedades de un objeto contra su modificación no autorizada, aislándolas en una especie de cápsula; ocultándolas o prohibiéndolas a quien no tenga derecho a acceder a ellas; posibilitando solamente a los propios métodos internos del objeto el acceso a su estado. Esto asegura que otros objetos no puedan cambiarle su estado interno de manera inadvertida o ilegal, eliminando efectos colaterales nocivos.

Cada clase está aislada del exterior, constituyendo un módulo natural, y cada tipo de objeto perteneciente a una clase determinada, expone una interfaz única a otros objetos, la cual especifica cómo pueden actuar entre sí o con otros objetos.

C++ provee un mecanismo de encapsulación que restringe el ámbito de acceso y viene dado por el uso dentro de la clase de los cualificadores **`public`**, **`private`** y **`protected`**.

Tabla 2. Cualificadores de clases

Cualificador	Uso
`public`	Esta parte constituye la interfaz (definiciones y métodos dentro del bloque público) y está disponible para cualquiera que la desee usar.
`private`	Nadie puede acceder a lo definido dentro del bloque privado (miembros o métodos), a excepción de los métodos internos de la clase y las funciones y clases amigas.
`protected`	Es un cualificador intermedio entre los dos anteriores: actúa como privado con respecto a las funciones y clases externas, pero como público frente a las clases derivadas. Está relacionado con la herencia.

Nota: en la inmensa mayoría de las clases bien diseñadas, los métodos serán públicos y los miembros serán privados, o puede que protegidos si es que hay herencia involucrada y ello se requiere.

La abstracción, el encapsulamiento y la ocultación de información se emplean mucho en conjunto pues, aunque no son lo mismo, se complementan muy bien. Todas forman parte de una correcta *metodología de trabajo* con las clases, propia de, y aplicable a los lenguajes híbridos, como Delphi o C++

C++ no obliga la encapsulación; esa técnica la debe saber aplicar el programador

Herencia

Usualmente las clases no se encuentran aisladas, sino que se relacionan entre sí, formando una jerarquía, donde las que están por "debajo" en el árbol jerárquico heredan las propiedades y el comportamiento (miembros y métodos) de todas las que les están vinculadas por "encima." Esto permite entre otras cosas, el desarrollo por diferencias, reutilizando código al máximo.

¡Una clase no debe heredar de otra porque posea las mismas variables! Ante todo, es necesario que haya un sentido común entre ellas, una identidad común entre las instancias de dos clases. Si se tiene una identidad común, de ello se deriva que ciertas informaciones son idénticas ¡y no al revés. (Dubois, pág. 10)

Cuando un objeto sólo puede heredar de una clase se dice que la herencia es simple. Cuando hereda de más de una clase se dice que la herencia es múltiple. Siendo este último tipo de alta complejidad técnica, no todos los lenguajes la implementan, y donde se puede, su uso se recomienda con discreción. C++ implementa la herencia múltiple, y en este texto se reserva un tema para ella.

La herencia organiza y posibilita la aplicación del polimorfismo y la encapsulación, permitiendo a los objetos ser definidos y creados como punteros a tipos especializados de objetos preexistentes. Estos objetos son llamados tipos concretos de datos o CDT (*Concrete Data Type* en inglés) y su manejo debería de acercarse lo más posible a un tipo nativo del lenguaje a través de la sobrecarga de operadores. Además, la cooperación, la extensión o la modificación de su comportamiento serán obtenidas sin tener que volver a ser implementadas desde cero.

La herencia, los punteros y el polimorfismo van todos en yunta

Polimorfismo

El polimorfismo es la posibilidad de considerar una instancia de una clase también como una instancia de sus clases de base. (Dubois, pág. 11) En las ciencias de la computación, el polimorfismo está representado por «una interfaz, muchos métodos.» Cuando esto ocurre en tiempo de compilación, la acción es llamada asignación temprana o enlace temprano (*early binding* en inglés), como en el caso de las plantillas, las funciones `inline` y las funciones sobrecargadas. Cuando se hace en tiempo de ejecución, es llamada asignación tardía o enlace tardío (*late binding* en inglés), siendo considerado polimorfismo verdadero, una característica definitoria de la OOP y C++ proporciona los dos tipos de enlace

El polimorfismo es también clave y puntal en el proceso de implementación de la OOP, ya que admite crear un conjunto de clases interactuantes con interfaz única, lo que reduce el nivel de complejidad presente en el problema y permite la actuación conjunta de sus objetos con mayor simplicidad y transparencia.

El polimorfismo necesita de la herencia y de los punteros para poder estar presente

Protección y seguridad de los datos

La protección y la seguridad de los datos son dos cosas bien diferentes:

1. Protección: los objetos son variables que pertenecen a sus respectivas clases. Los métodos, al pertenecer a clases diferentes, son llamados por sus respectivos objetos. Usualmente las interfaces de esos métodos están vacías, o son muy breves, resultando un código más fácil de escribir y muy robusto frente a los errores de pase de parámetros. Los métodos están atados a los miembros de la clase, que no al programa.

   En la práctica una clase no domina a un método que no es suyo y es muy difícil que pueda haber algún tipo de error al llamarles con parámetros cambiados o erróneos, por la tersura de sus interfaces. 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.

   ¿Resultado? No hay pérdida ni equivocaciones en su empleo: los datos están protegidos

2. Seguridad: cuando el programa principal (el cliente) llama a uno de sus objetos (el servidor) a través de un método, está enviándole un mensaje. Cuando posteriormente el objeto (que ahora deviene en cliente), llama pidiendo ayuda a otro objeto (que pasa ahora a ser servidor), también está enviándole un mensaje, pero en ningún caso se puede tomar cualquier otro método por equivocación, o por descuido tomar el método con datos erróneos, o tomar un método perteneciente a otro objeto no involucrado en la transacción. Eso es una parte de la seguridad.

   Una clase bien desarrollada aplica la encapsulación de modo que sus miembros sólo sean accedidos por métodos selectos de la interfaz. Nadie puede alterar el estado de sus miembros o manejarlos fuera de lo que el diseñador de la clase le permite. Es la encapsulación quien limita y controla los accesos y les protege del mal uso. Eso completa la seguridad.

   ¿Resultado? Los datos ahora están encapsulados y por ende, más seguros

3-BREVÍSIMA HISTORIA DE LA OOP

> *La programación orientada a objetos es un paradigma de programación que viene a innovar la forma de obtener resultados…Basada en varias técnicas, incluyendo herencia, cohesión*[3]*, abstracción, polimorfismo, acoplamiento*[4] *y encapsulamiento, su uso se popularizó a principios de la década de 1990. En la actualidad, existe una gran variedad de lenguajes de programación que soportan la orientación a objetos. Editado de (Wikipedia en español)*

[3] Cohesión es la propiedad de una clase de realizar una y sólo una tarea bien definida. Dice (Sutter, Exceptional C++, pág. Item 10): *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++, pág. Item 5)

[4] Acoplamiento es la cualidad que une las distintas partes del programa. Cuando se habla de módulos se dice que el acoplamiento es débil, pues un módulo puede ser erróneamente llamado, o llamado con los parámetros incorrectos. Cuando se habla de clases se dice que el acoplamiento es fuerte, porque lo anterior se minimiza. La moderna programación busca el acoplamiento fuerte.

Allá por la década de los años 90’ del siglo pasado, la experiencia con la creación de sistemas de software de calidad industrial, indicaba que cantidades considerables de código trataban con casos especiales, pero muy relacionados, cuyos detalles obscurecían el panorama general.

Los conceptos de la POO tienen origen en Simula-67, en ese mismo año. Este es un lenguaje diseñado para hacer simulaciones, creado por Ole-Johan Dahl y Kristen Nygaard, del Centro de Cómputo Noruego en Oslo, pues sus simulaciones de naves, fueron confundidas por la explosión combinatoria de cómo las diversas cualidades de diferentes naves podían afectar unas a las otras. La idea surgió al agrupar los diversos tipos de naves en diversas clases de objetos, siendo responsable cada clase de objetos de definir sus "propios" datos y comportamientos.

Estos conceptos fueron refinados allá por los ’70 en Smalltalk, desarrollado en Xerox PARC, pero diseñado para ser un sistema completamente dinámico, orientado a la educación, en el cual los objetos se podrían crear y modificar "sobre la marcha" (en tiempo de ejecución) en lugar de tener un sistema basado en programas estáticos. Entonces se constató que, aplicando los conceptos de la OOP, los programadores se enfocaban mejor en las características comunes entre los objetos del sistema, pudiendo manejar con más facilidad las complejidades inherentes. Ya para mediados de los ’80 sus características fueron agregadas a muchos lenguajes existentes durante ese tiempo, incluyendo Ada, BASIC, Lisp, Pascal, y C entre otros y se fue convirtiendo en el estilo de programación dominante —debido en parte a la influencia de C++— y su dominación fue consolidada gracias al auge de las interfaces gráficas de usuario, para las cuales está muy bien adaptada.

Con los primeros éxitos el uso de la OOP se popularizó y en la actualidad existen tres variedades de lenguajes de programación que soportan la orientación a objetos:

1. *Lenguajes verdaderamente orientados a objetos*, donde el entorno suministra las clases a utilizar. Por ej., Simula, Smalltalk y Eiffel.
2. *Lenguajes basados en clases*, donde se admiten las clases, pero sin herencia. Por ej., Java, PHP y C#.
3. *Lenguajes híbridos*, donde se soporta la OOP completamente, pero su uso queda a discreción del programador. Por ej., BASIC, Delphi y C++, el lenguaje que más ha influido en la diseminación de esta metodología y que ha marcado las pautas de la aplicación de la OOP en la industria contemporánea del software comercial.

Paradigmas de programación

En este campo se mueven en la actualidad fundamentalmente dos paradigmas de programación: primero, el de la programación modular y le sigue el de la OOP, dialécticamente surgido de éste y llevando sus principios consigo.

Programación modular

Tradicionalmente los datos (variables) y los procedimientos (módulos) están separados y sin relación, ya que lo único que se busca es el procesamiento de unos datos dados a la entrada de un módulo, para obtener cierta información a su salida, por lo que su acoplamiento con el programa es débil. En este tipo de programación sólo se escriben módulos que procesan datos, por eso también se les conoce por moledores de bits, o *bit crunchers*, en inglés.

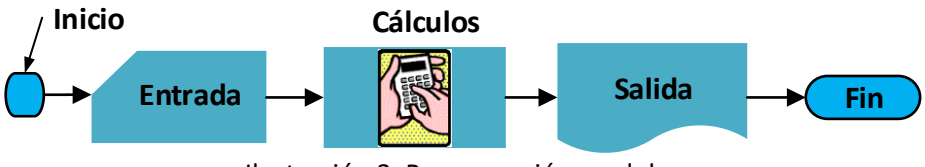


Ilustración 2. Programación modular

La programación modular alienta al programador a pensar ante todo en términos de procedimientos o funciones, y en el enlace con las estructuras de datos que manejan.

Programación orientada a objetos

La OOP se basa en el paradigma anterior usando su modularidad, pero quien la aplica cambia su forma enfocar un problema, y naturalmente, trabaja con otras herramientas. Este paradigma trata de encontrar solución a los problemas que presenta la complejidad de sistemas industriales muy grandes, introduciendo nuevos conceptos de codificación que superan y amplían los ya conocidos.

Teniendo presente que la OOP emergió como un desarrollo dialéctico debido a las limitaciones del anterior —y lleva en sí la esencia de sus raíces— si se programa en OOP correctamente, existe una muy alta probabilidad de que las creaciones sean también modulares.

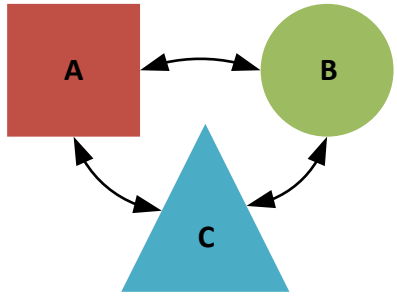


Ilustración 3. Programación orientada a objetos

Su empleo influye —a desarrolladores cuando planifican la aplicación y a programadores cuando escriben el código— a pensar en términos de objetos, en vez de procedimientos; a definir objetos para luego enviarles mensajes solicitándoles que ejecuten sus métodos por sí mismos y que interactúen con los otros objetos presentes, tratando de mapear[5] en su código las relaciones y acciones de las entidades que actúan en el espacio de soluciones del problema en cuestión. Por eso se dice que su acoplamiento es fuerte.

4-DEFINICIONES BÁSICAS

> *La POO es una forma de programar que…introduce nuevos conceptos, que superan y amplían conceptos antiguos ya conocidos. (Wikipedia en español)*

No se puede trabajar eficaz y eficientemente si no se conocen los elementos técnicos del lenguaje empleado, o como se dice, *la jerga en uso*: eso es obligatorio en un profesional. Para programar orientado a objetos, el codificador debe saber al menos el significado de una docena de sus términos más básicos, y sus relaciones generales con C++:

1. Clase: es un elemento del lenguaje que contiene los atributos de un ente afín al espacio de solución de un problema.

   Una clase permite al programador reunir en una unidad autónoma reconocida por el compilador, los aspectos figurativos y operacionales del nuevo tipo. También provee mecanismos para especificar niveles diversos de accesibilidad para los datos y funciones contenidos dentro de la clase. (Hazzah)

[5] Mapeo: anglicismo de *mapping*, que en contexto es crear en un lenguaje de programación, una representación abstracta de un ente real del espacio de soluciones del problema.

La instanciación de una clase se hace mediante su constructor, estando formada por la lectura de estas definiciones, el llenado de sus valores y la creación de un objeto como un nuevo tipo, a partir de ello. En C++ la clase se implementa como la unión de miembros y métodos en una estructura cerrada denominada **`class`**.

2. Objeto: es la instancia de una clase.

Cada vez que se instancia una clase, da origen a un objeto. Varias instanciaciones de una sola clase producen varios objetos idénticos en su accionar, pero independientes entre sí. Un objeto está formado por la lectura de sus atributos, el llenado de sus valores y en concreto se corresponde con una o más entidades afines al espacio de solución de un problema. En C++ un objeto es una variable definida por el usuario, cuyo tipo es una clase.

3. Atributo: engloba las características que definen una clase, determinadas básicamente por la unión del conjunto de sus miembros y sus métodos, su identidad, su estado interno y sus relaciones con otras clases. C++ soporta completamente el concepto de atributo.

4. Miembro: es la definición de una variable que necesita la clase para funcionar. En C++ un miembro es un tipo de datos nativo o definido por el usuario, que le pertenece a una clase.

5. Método: es un módulo asociado a una clase, cuya ejecución se desencadena tras los eventos ligados a la recepción de un mensaje. En C++ el método es una función-miembro de la clase. Desde el punto de vista del comportamiento, es lo que la clase permite hacer al objeto.

El conjunto métodos públicos proporciona una interfaz de comunicación para mensajes externos, proveyendo las operaciones necesarias y la forma de realizarlas.

6. Evento: es un suceso síncrono del sistema (es decir, al margen de los clics del ratón, el reloj del sistema, la administración de los recursos de SO subyacente, etc.), tal como una interacción del usuario con la aplicación, un mensaje enviado por un objeto, o la reacción que dicho mensaje desencadena. Se crea al invocar un método, suministrarle la información que requiere y recibir su respuesta. El sistema maneja un evento enviando el mensaje adecuado al objeto pertinente. En C++ el evento es generado por la llamada a un método.

> En la programación visual los eventos los controla el usuario. En la programación clásica —usada en el texto— los eventos están controlados por el programador y son de dos clases: *de salida* con la información obtenida, y *de entrada* solicitando acciones concretas. En conjunto se llaman operaciones de E/S.

7. Mensaje: es la comunicación dirigida al objeto, que le ordena la ejecución de uno de sus métodos. En C++, un mensaje es un evento desencadenado por la llamada al método de un objeto.

8. Interfaz: las clases pueden presentar partes públicas, protegidas y/o privadas: a la parte pública se le llama interfaz.

Es la única parte de la clase visible para las demás, por tanto, es lo único que los objetos disponen para comunicarse con ella, aparte de las clases y funciones amigas. En C++ todo lo que va en la sección pública forma parte de la interfaz de la clase.

9. Herencia: también conocida como derivación de clases, hace posible diseñar nuevas clases basándose en las ya existentes.

La clase derivada (clase-hija) posee un conjunto de los atributos de la clase base (clase-madre.) Cuando se deriva una clase de otra, normalmente se añadirán nuevos métodos y datos, pero es posible que algunos de estos métodos o datos de la clase-madre no sean requeridos; en ese caso pueden ser enmascarados en la clase-hija. También es posible que algunos de estos métodos o datos de la clase-madre necesiten un cambio de comportamiento; en ese caso pueden ser redefinidos.

Si una clase hereda de dos o más, la herencia es múltiple; si sólo puede heredar de una, es simple. C++ soporta ambos tipos de herencia y el texto cubre ese hecho.

10. Enlace: también conocido como *binding* en inglés, es la manera en que un compilador toma las partes del programa (código-fuente, bibliotecas, piezas de código de terceras personas, etc.) y las enlaza para formar un programa ejecutable.

Hay dos formas de hacerlo: *el enlace temprano*, donde todas las piezas se enlazan antes de la compilación del programa y *el enlace tardío*, que compila el programa, pero pospone el enlazado hasta que se determine durante la ejecución cuál es el módulo correcto. Esta característica es tan importante, que es considerada propia de la OOP. Los modernos compiladores de C++ manejan los dos tipos de enlace.

11. Escalamiento: es alterar un programa en vías de hacerse o que ya esté hecho [y posiblemente en producción] para integrarle nuevas prestaciones.

En la vida real, las aplicaciones comerciales por demanda se gestionan en el marco de un protocolo legal que rige la actuación, derechos y deberes tanto del cliente que contrata, como de la empresa que sirve. Pero lo más común durante el desarrollo es que el cliente se aparezca con nuevas demandas, y si se acepta esa petición, al elevarse la complejidad debido a las prestaciones a incluir, no es inusual que cambie todo el panorama, presentando otras perspectivas y otros diseños, porque muchas veces un cambio aparentemente menor implica alteraciones mayores, y entonces se dice que el diseño original no escala bien.

La OOP permite un escalamiento más fácil aplicando las reglas de la herencia para derivar los nuevos comportamientos exigidos por el cliente.

12. Reutilización: es el empleo de un código ya hecho.

Cuando en disímiles situaciones se emplea un código previamente hecho, se está en presencia de un fenómeno llamado reutilización. A veces se usa la herencia para implementar de forma relativamente fácil la reutilización del código con clases ya creadas y probadas, generando rápidamente un código minimalista y bueno, pero se advierte que la reutilización no debe ser el objetivo-guía de aplicación de la herencia.

La OOP permite un reúso más cómodo aplicando las reglas de la herencia para derivar los nuevos comportamientos exigidos por el cliente.

Desde luego, no todos los elementos aquí listados definen la OOP (hay más que no se conocerán en el texto, ni tampoco se requieren de dominar), pero es menester que el lector regrese a estas definiciones de vez en cuando, en la medida en que avanza en sus estudios. Las cosas se irán clarificando y será un índice de su progreso individual. Al menos, así le pasó a su humilde autor. Y se lo recomienda.

Por la extensión y complejidad del lenguaje, a C++ se le conoce popularmente por *La Bestia*. Como su creador Bjarne Stroustrup dijera en una conferencia para programadores profesionales: *…éntrele con calma; no piense que debe conocer y utilizar todas sus características y desde luego, no trate de utilizar todo el primer día.* Citado por (Prata, p. 564)

5-ESPACIOS DE NOMBRES

> *La elección de nombres es una actividad fundamental en la programación y cuando un proyecto se vuelve grande, el número de nombres puede ser abrumador. (Eckel, pág. xxx)*

Cuando los programas clásicos, modulares, codificados en C pasaban de cierto tamaño (el cual era variable y no vale la pena discutir aquí), se hacía difícil obtener nuevos nombres para sus variables. Al ser particionado el trabajo y repartido a

varios equipos profesionales de programación, cada uno de ellos era mantenido por diferentes programadores y todos los involucrados debían tener cuidado de no usar accidentalmente los mismos nombres en situaciones en las podían surgir conflictos.

El trabajo de evitar conflictos era un gran consumidor de tiempo, se hacía tedioso y ya de últimas, era un factor notablemente incidente en el encarecimiento del producto final que, debido a esto (entre otras causas), nunca cumplía con el tiempo calendario planificado para su entrega. Eso hizo quebrar a productos sólidos hasta ese momento y vienen a la mente del autor el sistema operativo CP/M 86 (igual o mejor que el DOS y más simple), el procesador de textos WordStar (mejor que Word) y la hoja electrónica de cálculo Lotus 1-2-3 (mejor que Excel), que no pudieron modernizarse a tiempo.

Para gestionar los nombres pertenecientes a un proyecto o evitar sobrescribir los elementos ya definidos al ponerles igual nombre, en C++ se crearon los espacios de nombres o **`namespace`**s del singular en inglés. Como dijera Tim Peters en su famoso Zen de Python: *los espacios de nombres son una gran idea ¡Hagamos más de esas cosas!*

¿Qué es el espacio de nombres?

Los identificadores […] deben ser significativos y comunicar muy claramente su intención. Las palabras en inglés[6] bien escogidas, o simples variantes de las comunes, son ideales. Pero esto significa que hay alta probabilidad de que una biblioteca desarrollada por terceras personas también contenga nombres usados por usted o por otras bibliotecas independientes. Cada una por sí misma no es problema, pero cuando se desarrolla una aplicación usando múltiples bibliotecas, pueden surgir (y verdaderamente, surgen) dificultades. (Halterman)

El espacio de nombres (**`namespace`** en inglés) es un ámbito o alcance de carácter local (*local scope* en inglés) que toma el ámbito global del programa y lo particiona, delimitando una región para albergar declaraciones, con el objetivo de evitar las colisiones de nombres. Ello posibilita usar igual nombre para dos elementos distintos, si están declarados en diferentes espacios, lo cual viabiliza el cumplimiento de las ODR (*One Definition Rules* o reglas de definición única en español) que establecen lo siguiente:

1. En un fichero-fuente de cabecera habrá al menos la definición de una entidad.
2. En un programa habrá exactamente una definición para cada elemento 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.

¿Cómo se implementa?

La creación de un espacio de nombres es muy fácil. Su sintaxis es:

```
namespace misNombres { ... }
```

[6] Claro, el autor de esas palabras es norteamericano ¿y qué va a decir? Pero realmente, muchos codificadores de otras nacionalidades trabajan sus nombres en inglés puesto que: (**a**) este es el idioma nativo de la programación; (**b**) usualmente son más cortos que sus contrapartidas en cualquier otra lengua; y (**c**) sus significados se dan más fácilmente, siempre que se conozca dicho idioma. El texto usa preferiblemente identificadores en español.

y se implementa mediante una declaración **`using`**, por ej.: **`using misNombres`**, que define el código aislado, o una directiva **`using`**, colocada al inicio del archivo, en la que se engloban todos los nombres que están protegidos dentro de su creación, como en **`using namespace misNombres`**.

Notas:

- Los espacios de nombres son aditivos, es decir, se pueden solicitar varias veces y en distintos ficheros: el efecto neto es la suma de todas sus creaciones.
- A menos que se sepa muy bien lo que se hace —y por lo antes dicho— no se deben incluir los espacios de nombres propios de C++ dentro de los definidos por el usuario.

6-UNA CLASE BÁSICA

En el principio creó Dios los cielos y la tierra. Y la tierra estaba sin orden y vacía. El Antiguo Testamento. Gn. 1.1-2

En 1991 dijo (Stroustrup): C++ no posee ni tipos de datos de alto nivel, ni operaciones primitivas de alto nivel. Por ejemplo, no existe un tipo de dato matriz y su operador de inversión, o un tipo de dato cadena de caracteres y su operador de concatenación. Si un usuario desea un tipo como cualquiera de esos, puede definirlo dentro del mismo lenguaje.

Pero con el correr de los años, la comunidad C++ produjo clases de uso general, de modo que no es menester reinventar la bendita rueda al pararse el curioso lector frente al lienzo en blanco. ¿Qué se necesita? ¿Una cadena de caracteres muy funcional? Pues tomarla de la biblioteca de plantillas STL. ¿Quizás una matriz altamente funcional? Pues tomarla de las bibliotecas Blitz++ o BLAS. ¿Tal vez un conjunto de funciones gráficas? Pues tomarlas de las bibliotecas Boost…y así sucesivamente.

Han pasado casi 30 años desde que C++ salió a viajar por el mundo, y su biblioteca ¡se pobló! Y muchas, muchas de sus clases han hallado el camino hacia el humilde programador, evitándole reinventar la rueda una y otra vez… ¡Aleluya!

Véase pues cómo se crea una clase.

Programa 1-Un producto alimentario

En la ciudad de Manzanillo la empresa de gastronomía utiliza ciertos productos en sus establecimientos. Tres de ellos son **(1)** los refrescos, suministrados por la Empresa Municipal de Bebidas y Licores; **(2)** los dulces, suministrados por la Empresa Municipal de Pan y Dulces; y **(3)** los helados, suministrados por la Empresa Provincial de Productos Lácteos. Crear una clase que los maneje.

Análisis y diseño

Al escribir una clase lo más correcta posible, siempre se debe preguntar ¿qué necesita saber hacer la clase? ¿qué necesita saber el usuario que adquiere este producto? ¿cuál será la interfaz? Es inevitable el hacer algún tipo de diseño, aunque sea el más elemental. Por ejemplo, el nombre, la cantidad adquirida, y los precios de costo y venta, que el autor imagina es por resolución nacional y que, para estos productos, es único.

Tabla 3.Precios de alimentos

Producto	Costo	Venta
Dulce rollito	0.73	1.00
Refresco botella	0.57	1.20

Helado	0.49	0.75

Se decide que cada producto debe saber cómo entrar y manejar sus datos, y cómo mostrarse en un medio. Para los propósitos ilustrativos, la abstracción que se brinda está completa por parte del usuario. Si fuera a crearse la clase para una de las empresas productoras, seguramente que se incluirían más características que ayudarían a particularizar completamente al producto. Así pues, la abstracción es un mecanismo poderoso que aísla y determina las características de la cosa real en dependencia del uso que se le va a dar.

Nota: el desarrollador debe conocer la mecánica de la obtención de información de los datos dados. Si las operaciones matemáticas están fuera de su competencia, solicitará la ayuda de un experto, pero esa generación de información tiene que estar correcta pues es la que aporta la integridad al programa, su primera y más importante cualidad.

Eficiencia

La eficiencia es $\eta = \left(1 - \frac{líneas\ de\ código}{total\ de\ líneas}\right) \times 100\ \%$ Las líneas vienen dadas por un **`plugin`** llamado *Code statistic*s.

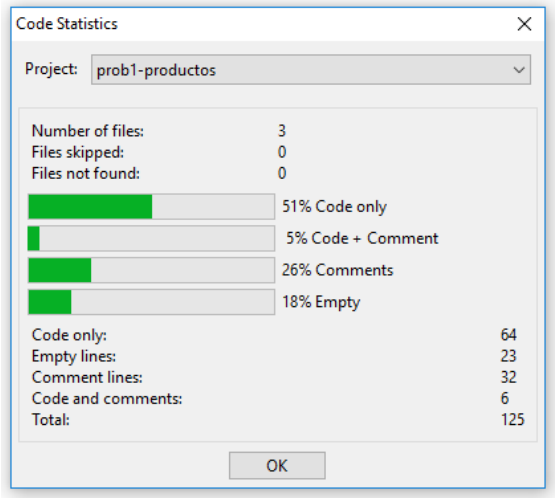


Ilustración 4. La eficiencia de un programa

Aplicado a este programa la eficiencia da $\eta = \left(1 - \frac{64+6}{125}\right) = 0.44$ o el 44%

Constructor

En C++, la inicialización en demasiado importante como para dejársela al programador...Si una clase tiene un constructor, el compilador hará que se llame automáticamente […] en el momento de la creación del objeto, antes de que el programador pueda llegar a tocar el objeto. La invocación del constructor no es una opción para el programador; es realizada por el compilador en el punto en el que se define el objeto. (Eckel, pág. 188)

Una clase 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*. Si no es definido, C++ emplea uno por omisión…que no hace nada, excepto permitir la declaración del objeto. Si el usuario suministra un constructor, C++ no pone el que va por omisión.

Si el constructor lleva parámetros usados para pasarle valores a los miembros de la estructura 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. Si el programador define un constructor y no le da parámetros por omisión, usualmente también deberá definir otro sin parámetros.

El constructor puede ser sobrecargado tal y como se hace con una función normal, y se pueden definir tantos como se requieran.

Destructor

En C++, la limpieza es tan importante como la inicialización y por eso está garantizada por el destructor, función especial que lleva el nombre de la clase con una tilde antepuesta (**~**) y no puede tomar parámetros o retornar valores, porque *su única misión es destruir un objeto*. Tal es así, que el constructor llama automáticamente a su destructor cuando la clase sale de ámbito.

Si no se usa la tienda gratis directamente en la definición de sus miembros, usualmente no hace falta explicitarlo, pues C++ emplea uno por omisión.

Finalmente, toda la construcción va envuelta dentro del espacio de nombres definido por el usuario, pero ¡muy importante!, excluyendo las llamadas a las funciones propias de ISO C++. Además, el autor recomienda poner un comentario de Doxygen en cada clase como: **`@brief <objetivo>`**, que debe ser dado en detalle, para facilitar la labor del usuario de la clase.

Los métodos inline

El uso del cualificador **`inline`** muchas veces se torna inapropiado, porque informa demasiados detalles, dificulta el mantenimiento y complica entender lo que los objetos de la clase se supone que deben hacer, porque el cómo, o sea, su reproducción allí donde se usa, está diseminado por todo su código.

Primero hay que considerar que las ineficiencias reales al día de hoy son insignificantes, y que —técnicamente hablando— el compilador no puede hacer un método **`inline`** si su implementación no es sencilla (que no significa necesariamente que sea codificada en una línea) o es recursivo, o muy complejo, o muy grande, o tiene ciclos, o **`switch`**es, o variables estáticas.

Por otra parte, toda definición de algún método que se haga en el mismo fichero de cabecera el compilador tratará de hacerla **`inline`** por omisión, lo cual eleva la rapidez de trabajo del programa, aumentándole en muy poco su tamaño, *pero violando en parte el principio de encapsulación*, pues al revelar una implementación en un fichero desprotegido, un usuario pudiera sustituirla por la suya propia, inmiscuyéndose en un área donde *nunca* debiera entrar.

Esta es la recomendación más evidente: si la seguridad es una preocupación de primer orden, entonces los métodos **`inline`** simplemente y por ahora, no se usan.

Lista de inicialización

La lista de inicializadores del constructor permite invocar explícitamente a los miembros. De hecho, no existe otra forma de llamar a esos miembros si son constantes. La idea es que los métodos sean llamados antes de la ejecución del cuerpo del constructor de la nueva clase. De esta forma, cualquier llamada que hagamos a las funciones miembros de los sub-objetos siempre serán objetos inicializados. (Eckel, pág. 407)

Dicen (García de Jalón, Rodríguez, Sarriegui, & Brazález, pág. 36) que C++ permite inicializar variables-miembro fuera del cuerpo del constructor y estas inicializaciones son más eficientes que las instrucciones de asignación típicas; además, permiten definir variables-miembro constantes, que pueden ser inicializadas al declararlas, pero no asignadas después.

Esta lista va entre el nombre de la clase y sus llaves y comienza con dos-puntos. Consiste en nombrar el miembro y poner entre paréntesis su valor, usualmente declarado en la interfaz de esa clase. Cuando son varios, los miembros van separados por comas.

Esta inicialización en concreto *solo vale para el constructor*. Dentro de cualquier otro módulo hay que asignar los valores tradicionalmente.

Nuevas formas de inicializar

Este concepto nuevo de inicialización admite proveer listas de valores a tipos de datos. Para hacer una sintaxis sólida, C++ permite crear variables de tipos predefinidos fuera de la clase. Por ej.:

```
int i(100);
int* pInt = new int(12);
```

Esta forma es más restrictiva que la tradicional heredada de ANSI C, porque no permite el fenómeno de conversión automática entre tipos llamado estrechamiento (*narrowing* en inglés) que asigna un valor a un tipo numérico normalmente incapaz de recibirlo. También es la principal razón por la que hay que definir una variable en el punto más cercano a su primer uso. Por ejemplo, sea la declaración **`int xyz = 123`**. La norma de ISO C++11 permite declarar lo mismo así:

```
int xyz = (123); // compatible con lo tradicional
int xyz = {123}; // compatible con lo tradicional
int xyz (123);   // norma C++11
int xyz {123};   // norma C++11
```

donde el operador de igualdad se conserva por motivos de compatibilidad retroactiva. Si hay que hacer algo más, se hace dentro del bloque del constructor. Entonces, este concepto se aplica como se muestra:

```
Producto::Producto(Str nb, float co, float ve, int ct): nombre(nb), costo(co), venta(ve), cantidad(ct){}
```

Por último, todo el constructo va envuelto en el espacio de nombres definido por el usuario y el anterior y éste se suman, abarcando todo el futuro objeto.

Notas:

- La lista sólo admite datos nativos C++. Para entrar contenedores hay que ir a lo tradicional.
- Si va a entrar un valor a un miembro que sólo existe en la clase-hija, entonces debe hacerse en el cuerpo de su respectivo constructor.

Declaración

```
#ifndef PRODUCTO_H_INCLUDED
#define PRODUCTO_H_INCLUDED

#include <string>
typedef std::string Str;
#include <fstream>

namespace misNombres { // <--
/**
@brief Maneja un producto alimentario.
@param <b>nombre</b> del producto.
@param <b>costo</b> del producto.
@param <b>venta</b> del producto.
@param <b>cantidad</b>del producto.
```

```
S. Galliano. 1/7/2020; 8:45 pm
*/

class Producto {
public: // interfaz pública
    // constructor
    Producto( Str nb = "", float co = 0, float ve = 0, int ct = 0 );
    // accesores de entrada
    void setNombre( Str valor );
    void setCantidad( int valor );
    // accesores de salida
    Str   getNombre()   const;
    float getCosto()    const;
    float getVenta()    const;
    int   getCantidad() const;
    // método
    void muestra( std::ostream & escribe ) const;
private: // miembros privados
    Str nombre;
    float costo;
    float venta;
    int cantidad;
};
} // misNombres <--

#endif // PRODUCTO_H_INCLUDED
```

Definición

```
#include <iomanip>
#include <ios>
#include "producto.h"

using std::ios;

namespace misNombres { // <--
Producto::Producto( Str nb, float co, float ve, int ct )
    : nombre( nb ), costo( co ), venta( ve ), cantidad( ct ) {}

// accesores de entrada:
void Producto::setNombre( Str valor )   { nombre   = valor; }
void Producto::setCantidad( int valor ) { cantidad = valor; }

// accesores de salida:
Str   Producto::getNombre()   const { return nombre;   }
float Producto::getCosto()    const { return costo;    }
float Producto::getVenta()    const { return venta;    }
int   Producto::getCantidad() const { return cantidad; }

// miembro
void Producto::muestra( std::ostream & escribe ) const {
    escribe << setiosflags( std::ios::fixed | std::ios::showpoint )
            << std::setprecision( 2 )
            << "\nNombre: "    << getNombre()
            << "\nCosto = $"   << getCosto()
            << "\nCantidad = " << getCantidad()
            << "\nA pagar: $"  << getCosto() * getCantidad() << "\n";
}
} // misNombres <--
```

Accesores de miembros

La seguridad de los datos implica ante todo la privatización de los miembros de la clase. No pueden ser accedidos libremente; la clase es quien lo permite. Entonces ¿cómo hacerlo? Existe una *metodología de trabajo* que define los accesores, funciones especializadas que se encargan manejar los valores de los miembros.

Hay accesores de entrada de datos y de salida de valores. A los primeros se les denomina seters y a los últimos, geters. El código muestra su prototipo más simple, aunque pueden ser usados con fines de validación o formateo. El método que fija un valor llama al correspondiente seter; el que necesita de uno llama a su geter.

Se conocen así porque sus respectivas identificaciones comienzan con **`set`** o con **`get`**.

1. ***accesor de entrada:*** `void Producto::setNombre( Str valor ) { nombre = valor; }`
2. ***accesor de salida:*** `Str  Producto::getNombre() const { return nombre; }`

Notas:

- El uso de accesores *es una metodología* y apegarse a ella es bueno, pero no obligatorio; no es una camisa de fuerza. El programador puede usar métodos o funciones que hagan lo mismo y se llamen de otra forma, aunque apartarse de la tradición es imponer por gusto un código que muy probablemente termine siendo ofuscado.
- Si no se necesita un accesor, no se codifica, en aras del *principio de la sencillez máxima*. Por ejemplo, en este caso el precio de venta y el de costo están definidos para cada línea alimentaria en el país y no necesitan ser alterados.
  - ✓ La sencillez máxima se aplica al código que cumple cabalmente con las cualidades de claridad y simplicidad

Métodos

Los métodos plasman el comportamiento de la clase y deben ser codificados teniendo siempre presente la ocultación de información, haciéndolos autocontenidos y autosuficientes. En este caso se programó para la generalidad, permitiendo escribir a cualquier periférico de salida:

```
void Producto::muestra( std::ostream &escribe ) const { ... }
```

Como efecto colateral benigno, los formatos de salida se retienen y el programa principal los usa.

Un par de errores

Estos errores son comunes en la programación del novato. El primero se da al tratar de acceder a un miembro privado directamente, lo cual viola la seguridad de los datos. El otro se da al aplicar la inicialización directa de una clase, a la cual no puede faltarle el operador de asignación.

1. Sea la instrucción **`total += prod1.costo * prod1.cantidad`**, que trata de acceder a los miembros **`costo`** y **`cantidad`** directamente. Al ejecutar el programa se obtiene:

   ```
   ||=== Build: Release in prob1-productos (compiler: GNU GCC Compiler) ===
   In function 'int main()':
   error: 'float Producto::costo' is private
   error: within this context
   error: 'int Producto::cantidad' is private
   error: within this context
   ||=== Build failed: 4 error(s), 0 warning(s) (0 minute(s), 1 second(s)) ===
   ```

   Que dice más o menos: en la función **`int main()`** hay error: **`float Producto::costo`** es privado. Y lo mismo para la invocación **`cantidad`**. Si se usa otro IDE, el mensaje cambia, pero esencialmente será el mismo. Esto es seguridad de los datos.

2. Si la lista se declara así: **`prod1 { "Dulce rollito", .73, 1, 1200 }`**, al obviar el operador de asignación, el compilador se queja pues la sintaxis demanda que la instrucción termine con la declaración de la clase prod1; el resto de la instrucción se ignora y el compilador lo advierte.

```
||=== Build: Release in prob1-productos (compiler: GNU GCC Compiler) ===
In function 'int main()':
error: expected ';' before '{' token
warning: statement has no effect [-Wunused-value]
||=== Build failed: 1 error(s), 1 warning(s) (0 minute(s), 4 second(s)) ===
```

   Que dice más o menos: en la función **`int main()`** en **`main.cpp`** hay un error: se espera **`';'`** antes del token **`'{'`**; advertencia: la instrucción no tiene efecto; se enlazó con la opción **`[-Wunused-value]`** del compilador. Esto significa que el programador aún no comprende bien el mecanismo completo de la declaración e iniciación de una clase.

Uso

Las clases, como CDT tienen multitud de usos, incluyendo en primer lugar los que dictaron su diseño, y muchos otros para los cuales ni siquiera se pensó. Se muestran tres declaraciones y sus usos en un manejador de muestra. Cada producto se sabe mostrar en un flujo de salida cualquiera: a disco, a consola o a otro periférico. Para el cálculo del total a pagar por el pedido se llaman directamente a los correspondientes geters y se realiza la operación financiera adecuada.

```
#include "producto.h"
#include <iostream>
#include <cstdlib>
#include <iomanip>

using namespace misNombres; // <--

int main() {
    using std::cout;
    cout << "Programa 1: una clase en acci\242n.\n";

    // una lista de valores
    Producto prod1;
    prod1 = { "Dulce rollito", .73, 1, 1200 };
    prod1.muestra( cout ); // lo muestra

    // usando el constructor
    Producto prod2( "Refresco botella", .57, 1.2, 1500 );
    prod2.muestra( cout ); // lo muestra

    // en la tienda gratis
    Producto *prod3 = new Producto( "Helado", .49, .75, 1500 );
    prod3->muestra( cout ); // lo muestra

    // calculando el pago de la factura
    float total = 0;
    total += prod1.getCosto()  * prod1.getCantidad();
    total += prod2.getCosto()  * prod2.getCantidad();
    total += prod3->getCosto() * prod3->getCantidad();
    cout << "\nTotal factura: $" << total << '\n';

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

Salida

```
Programa 1: una clase en acción.

Nombre: Dulce rollito
Costo = $0.73
Cantidad = 1200
A pagar: $876.00

Nombre: Refresco botella
Costo = $0.57
Cantidad = 1500
A pagar: $855.00

Nombre: Helado
Costo = $0.49
Cantidad = 1500
A pagar: $735.00

Total factura: $2466.00
```

La eficiencia de la respuesta es del 44%

7-UNA CLASE COMPUESTA

> *Cuando modele la relación tiene-un/tiene-una siempre prefiera la agregación y no la herencia. Use la herencia privada sólo cuando es absolutamente necesaria, es decir, cuando se necesita acceder a los miembros protegidos o privados, o se necesita sobrepasar una función virtual. Nunca use la herencia pública para el rehúso del código. (Sutter, Exceptional C++, pág. Item 22)*

La cooperación entre clases se basa en entender cada tipo de relación presente, y conocer cómo se implementa, ya que ello determina el tipo de mensaje que las puede interconectar. Las relaciones entre clases básicamente son de dos tipos:

1. La relación *tiene-un/tiene-una* indica que a una clase se le pueden agregar varias instancias de otra u otras. A esto se le llama composición por agregación, composición o delegación y se ha venido haciendo con estructuras en los dos primeros tomos de la serie.

La agregación siempre expresa la relación *tiene-un/tiene-una*, y por consiguiente, *se-implementa-en-términos-de*. Hace a la clase-cliente depender sólo de las partes públicas de la clase agregada. Al modelar *se-implementa-en-términos-de*, siempre prefiera expresarla usando la agregación y no la herencia.

Prefiera la agregación sobre la herencia

2. La relación *es-un/es-una*, implica la jerarquía básica de la clase y de sus descendientes, involucrando a la herencia.

A veces este tipo de relación puede ser vista como *trabaja-como-un/trabaja-como-una* o *es-usable-como-un/es-usable-como-una*. A esto se le llama desarrollo por diferencias y se aprenderá a hacer más adelante.

El desarrollo por diferencias tiene sus puntos fuertes en la programación visual, que no se estudia aquí

Categorías de los métodos

Los métodos tienden a ser agrupados bajo las siguientes categorías:

1. Constructores y destructores: casi de obligatorio cumplimiento; como ya se indicó, están para eso mismo, la construcción y destrucción seguras de los objetos.
2. Métodos estándar: ejecutan los comportamientos de la clase, manteniendo la información sobre los datos o el estado del sistema al día, dando así sentido al concepto de interfaz.
3. Fuente de datos: generan datos, usualmente por demanda y los transmiten a otras clases, o los guardan en algún dispositivo permanente.
4. Sumidero (pozo) de datos: procesan datos tomados ya sea de una fuente directa, ya de un dispositivo de almacenaje permanente y lo convierten en información.
5. Accesores: acceden a miembros privados y están formados por **`geters`** y **`seters`**, conjunto de funciones tipo **`get/set`**, permitiendo la encapsulación. Los métodos que recobran información casi siempre son muy simples y directos, aunque puede ser usados para entregar datos formateados; los que alteran los miembros también son simples y directos, pero pueden ser usados como filtros de validación de datos. Ambos tipos pueden ser sobrecargados.
6. De Vista: sirven para presentar información al usuario en un dispositivo de salida adecuado. Su código se encarga del formateo de los datos y muy a menudo es complejo, modificado con relativa frecuencia, y deben estar muy débilmente acoplados con los datos que manejan.

   Una buena práctica de diseño manda a separar las vistas del modelo, es decir, de los datos que manejan, lo cual simplifica el diseño —especialmente en la programación visual— de estos últimos, ya que un modelo puede tener varias vistas. Por ejemplo, unos datos financieros pueden ser vistos como una tabla o como una gráfica. Esta separación permite ampliar la reusabilidad del sistema.

   A veces una vista debe interactuar con su modelo, por ej., en un ambiente gráfico, donde los datos cambien dinámicamente y es deseable que la vista refleje dicho cambo al instante. Entonces, al cambiar el modelo, éste debe avisar a la vista para que se refleje dicho cambio.

7. Métodos utilitarios: ayudan a los otros métodos a ejecutar tareas largas, complejas, o difíciles. Son llamados desde donde se requieran, y pueden llegar a ser muchos y muy variados. Los principales son:
   a) Funciones especiales para el manejo de la memoria dinámica, entre todas ellas son ineludibles la sobrecarga del constructor de copias, la del operador de asignación y la del operador de direcciones. Siempre se deben definir en cualquier clase que contenga punteros a la tienda gratis.
   b) Sobrecarga de todo tipo de operadores que sirvan para acercar el manejo de una clase al de un tipo nativo de datos.
   c) Funciones y clases amigas de todo tipo, para flexibilizar al máximo la interfaz de la clase. Usualmente al sobrecargar un operador binario, se le declara una función amiga que unifique su expresión de llamada a ambos lados del operador.
   d) Funciones de conversión automática entre clases y tipos nativos, o entre clases.

Muchas veces no hay una clara distinción entre categorías de métodos

Métodos especiales automáticos

Son métodos definidos automáticamente por C++. En particular, siempre se proveen dos: un constructor y un destructor. Ambos son de oficio y no hacen nada. Están ahí para proveer funcionalidad adecuada al construir las clases. Es de notar

que si el usuario define un constructor, un destructor o ambos, C++ no suple su versión por defecto y el programador es quien debe hacerlo.

O sea, si se pone un constructor parametrizado, es muy posible que haya que definir uno de oficio. Y si se usa la tienda gratis con algún miembro que no sea STL, hay que codificar un destructor que se ocupe de devolver memoria.

Una forma de poner el constructor adecuado es darle valores por omisión a todos sus parámetros; entonces realiza una tarea dual: hace de constructor parametrizado y también de constructor de oficio.

Además, C++ suple un constructor de copia que hace copias fieles, un operador de asignación capaz de hacer copias fieles y un operador de direcciones. Estos tres últimos métodos usualmente trabajan bien si no está involucrada la tienda gratis, que no tolera las copias fieles debido a que se basa en punteros y requiere de copias y asignaciones "inteligentes" para no perder direcciones, necesitando además de un destructor explícito que opere sobre punteros.

- ✓ Copia fiel: copia hecha byte-a-byte que reproduce en otra parte de la memoria un doble idéntico al original
- ✓ Copia inteligente: copia que protege los punteros del doble borrado, replicándolos en otras direcciones.

Si una clase tiene uno o más miembros que son punteros de la tienda gratis, usualmente hay que definir un constructor de copia que se encargue de transferir las direcciones de los punteros que se copian a las direcciones de los copiados — es la copia inteligente. Y, por último, si el programador no define alguno de los métodos tabulados, C++ los pone implícitamente.

Tabla 4. Métodos automáticos

Método	Declaración
Constructor de oficio	`clase();`
Destructor de oficio	`~clase();`
Constructor de copia	`clase( const clase & rhs );`
Operador de asignación	`clase & clase::operator=( const clase & rhs )`
Operador de direcciones	`clase &rMiObjeto = objeto_de_la_clase;`

Nota: `rhs` significa *right hand side* o "a la derecha [del operador]", en español.

También se presenta muchas veces la necesidad de usar clases en expresiones que esperan un objeto diferente. Para eso se emplean operadores de conversión que circunvalan la regla de que *toda función regresa un valor*; al igual que constructores y destructores, ellos no regresan valor alguno. Son de dos tipos y complementan la interfaz:

1. Constructor de conversión `clase( tipo valor );` Por ej.:

```
Contador( unsigned val );
unsigned elValor = 5;
Contador laCant = elValor; // permite objeto = valor
```

2. Operador sobrecargado `operator unsigned();` Por ej.:

```
Contador ctd( 5 );
unsigned elValor = ctd; // permite valor = objeto
```

Por fortuna ahí está la STL para interactuar con la tienda gratis y simplificar la existencia del programador

Programa 2-Una cafetería

La cafetería "El Jardín" pertenece a la Empresa Municipal de Gastronomía de Manzanillo. En este momento, en ella se ofertan a la población tres productos cuyos precios están dictados por los dados en la página 21. 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. También se le dio el programa que controla los pedidos.

Análisis y diseño

Si se analiza el entorno más global del problema inmediatamente se ve que hay dos clases involucradas: la entidad "El Jardín" *es-una* cafetería (estructura básica) que *tiene-unos* productos (estructuras agregadas.) Los productos actuales son tres, pero nada implica que en un futuro la cafetería amplíe su oferta.

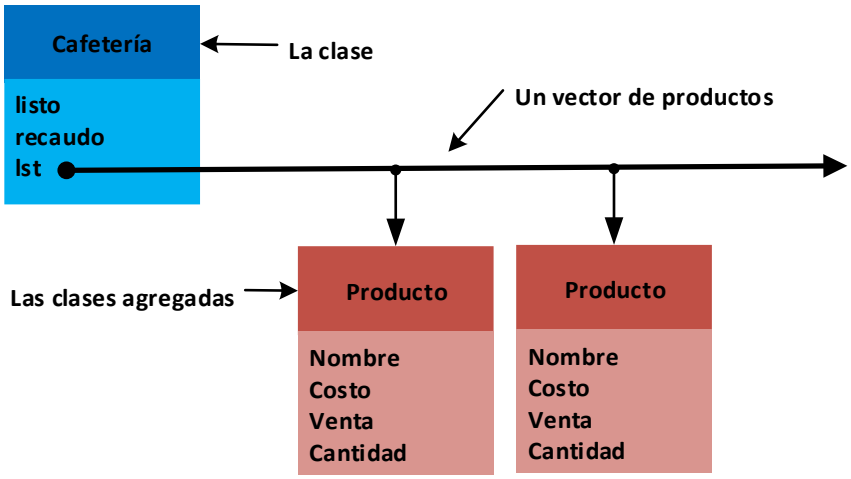


Ilustración 5. La clase compuesta

Entonces, una forma de verlo es: la cafetería es una clase llamada Cafetería que posee como miembro un vector STL de clases tipo Producto el cual se instancia varias veces. Tener presente que existe un nivel de indirección en el listado: hay que "bajar" el producto antes de accederlo.

¡Y qué bien! La STL se ocupa de copiar lo que sea, gestionar la memoria, construir, destruir y en general, simplificarle la existencia al humilde programador. Los manejos de las indirecciones son fáciles, el código es limpio, minimalista y bueno. EODA, siempre que se vaya a componer clases hay que pensar en sus miembros como entes de la STL, o al menos integrantes, como es este caso. Es decir, emplear cadenas C++, vectores, listas, etc. y dejarle los finos detalles del trabajo a la STL, que para eso la hicieron.

Declaraciones

¿Qué necesita saber el usuario sobre la entidad? 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 y el menú debe gestionarlos. Como la estructura agregada fue hecha en otra parte, hay que integrarle ambos ficheros.

- 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
```

- Entidad: no incluye al destructor pues no hace falta. Carga el listado inicial y el método de mostrar el valor toma como parámetro el total de las ventas. La clase agregada se toma de dónde se definió. Todo va dentro del espacio de nombres del usuario.

```
#ifndef ENTIDAD_H_INCLUDED
#define ENTIDAD_H_INCLUDED

#include <string>
#include <vector>
#include "../prob01-productos/producto.h" // <-- requerido

typedef std::vector<misNombres::Producto> Lst;
const std::string ERROR( "Tomar ventas primero.\n\n" );

namespace misNombres {
/**
@brief Maneja una entidad de venta.
@param <b>recaudo</b> del producto.
@param <b>listo</b> si ya puede reportar.
@param <b>lst</b> listado de los productos.
S. Galliano. 1/7/2020; 8:45 pm
*/
class Entidad {
public:
    Entidad( bool valor = false ); // constructor
    // accesores
    void  setBool( bool  valor );
    void  setReca( float valor );
    bool  getBool() const;
    float getReca() const;
    // métodos
    void  cargarLista();
    void  muestraMenu();
    float tomaVentas();
    void  muestraRecaudo( float valor );
private:
    double recaudo;
    bool listo;
    Lst listado;
};

} // misNombres

#endif // ENTIDAD_H_INCLUDED
```

Definiciones

- Menú

```
//  menu.cpp: definiciones para el programa 2
#include "menu.h"
```

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

//  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
}
```

- Entidad: Para cargar los productos se emplea el constructor, pero pudieran establecerse por el usuario...Para mostrar datos tales como las cantidades vendidas de cada alimento y el recaudo total, se baja un nivel de indirección con la estructura auxiliar **`Producto tmp`**, que posibilita sacar el dato iésimo con la instrucción **`tmp = listado[i]`**.

  Hay un pequeño problema: el texto utiliza la potencia del IDE, pero pierde la flexibilidad innata de cuando se hacen las cosas por sí mismo. Con una compilación a la medida, simplemente se incluye el fichero-objeto **`producto.o`** en la línea de comandos y ya está. Pero hay que saber hacerlo, y como este IDE crea todo desde cero y no contiene al fichero-objeto anterior, éste se pierde. Por eso hay que explicitar dónde está **`producto.h`** , para que lo enlace. También hay que integrar al programa el fichero **`producto.cpp`** y por último todo va dentro del espacio de nombres definido por el usuario.

```
#include "entidad.h"
#include <iostream>
#include <iomanip>
#include <cmath>

using std::cout;
using std::cin;
using std::setw;

namespace misNombres {
Entidad::Entidad( bool valor ) : listo( valor ), recaudo( 0 ) {
    listado.clear();
    cargarLista();
}

void  Entidad::setBool( bool  valor ) { listo = valor; }
void  Entidad::setReca( float valor ) { recaudo = valor; }
bool  Entidad::getBool() const { return listo; }
float Entidad::getReca() const { return recaudo; }

void Entidad::cargarLista() {
    Producto // primero cargar las estructuras
    p1( "Rollito",  .73, 1   ),
```

```
    p2( "Refresco", .57, 1.2 ),
    p3( "Helado",   .49,  .75 );
    // luego entrarlas en la lista
    listado.push_back( p1 );
    listado.push_back( p2 );
    listado.push_back( p3 );
}

void Entidad::muestraMenu() {
    Producto tmp;
    cout << "Hay " << listado.size() << " productos en el men\243.\n\n"
         << std::setprecision( 2 ) << std::fixed
         << std::left  << setw( 10 ) << "Oferta"
         << std::right << setw( 10 ) << "Precio$\n";

    for ( auto x: listado ) {
        cout << std::left  << setw( 10 ) << x.getNombre()
             << std::right << setw( 8 )  << x.getVenta() << '\n';
    } // for

    cout << '\n';
}

float Entidad::tomaVentas() {
    Producto tmp;
    float cant, total = 0;
    cout << "Entrando las cantidades vendidas.\n\n";

    for ( unsigned i = 0; i != listado.size(); ++i ) {
        cout << listado[i].getNombre() << ": ";
        cin >> cant;
        listado[i].setCantidad( cant );
        total += listado[i].getCantidad() * listado[i].getVenta();
    } // for

    cout << '\n';
    return total;
}

void Entidad::muestraRecaudo( float valor ) {
    cout << "\nTotal ventas del d\241a:\n\n"
         << std::setprecision( 2 ) << std::fixed << std::left
         << setw( 10 ) << "Oferta" << std::right
         << setw( 10 ) << "Cant."
         << setw( 10 ) << "Precio"
         << setw( 10 ) << "Venta"  << '\n';

    for ( auto x: listado ) {
        cout << std::left
             << setw( 10 ) << x.getNombre() << std::right
             << setw( 10 ) << x.getCantidad()
             << setw( 10 ) << x.getVenta()
             << setw( 10 ) << x.getCantidad() * x.getVenta() << '\n';
    } // for

    cout << std::right << "Total " << std::setfill( '.' ) << setw( 34 ) << valor << "$\n\n";
}

} // misNombres
```

Uso

Emplear el espacio de nombres definido por el usuario es seguro, porque está totalmente controlado. En el programa se usa con fines didácticos la restricción: se puede llamar a una entidad directamente porque se usó la directiva **`using misNombres::Entidad`**.

```
#include "menu.h"
#include "entidad.h"
#include <iostream>
#include <cstdlib>

using misNombres::Entidad;

int main() {
    Str info( "Programa 2: una clase compuesta\n\nCafeter\241a El Jard\241n - MEN\351\n" );
    Entidad * pEnt = new Entidad;
    short opcion;

    while ( true ) {
        opcion = menu( info );

        switch ( opcion ) {
        case Mostrar:
            pEnt->muestraMenu();
            break;
        case Vender:
            pEnt->setReca( pEnt->tomaVentas() );
            pEnt->setBool( true );
            break;
        case Recaudar:
            if ( pEnt->getBool() ) {
                pEnt->muestraRecaudo( pEnt->getReca() );
                delete pEnt;
                return EXIT_SUCCESS;
            } else {
                std::cerr << ERROR;
            } // if-else
        } // switch

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

Una posible salida

```
Programa 2: una clase compuesta

Cafetería El Jardín - MENÚ
Sistema de venta diaria

(O)fertas, (V)entas, (R)ecaudación
Entre una opción: o
Hay 3 productos en el menú.

Oferta      Precio$
Rollito        1.00
Refresco       1.20
Helado         0.75
```

La eficiencia es del 34%

8-UNA CLASE UTILITARIA

> *El tiempo es una magnitud física con la que se mide la duración de acontecimientos sujetos a cambios, en un sistema sujeto a observación, perceptible para un observador o aparato de medida. Su unidad básica en el Sistema Internacional es el segundo, cuyo símbolo es **s** — Debido a que es un símbolo y no una abreviatura, no se debe escribir con mayúscula, ni como seg, ni agregando a la letra "ese" un punto posterior. Editado de (Wikipedia en español)*

El fundamento de la OOP es que las clases ayuden a la solución del problema manejando sus complejidades. Una aplicación puede apoyarse en el trabajo específico de las clases. Esto permite crear clases utilitarias que se enfocan en una tarea que se necesitará en más de una aplicación. Luego pueden ser agrupadas en bibliotecas afines.

Uno de estos escenarios ya conocido es cuando se requiere medir el tiempo que tarda un proceso dado, para poder perfilar si es necesario mejorar esa parte del programa y obrar en consecuencia.

Programa 3-Un cronógrafo

En (Introducción al C++, tomo 2, pág. 99 y siguientes) se plantea una comparativa entre los tiempos de ordenamiento de varios contenedores STL. Para probar el funcionamiento del programa usando un cronógrafo se retoma el problema, pero en aras de una didáctica más clara, solamente con dos de ellos: el vector y la lista.

Poner todo en una función es una solución no muy satisfactoria, porque no se puede medir tiempos en eventos que se entrecruzan o están contenidos dentro de otros, además de tener que recurrir a variables globales, con potencial de choque de nombres, o definir `namespaces` innecesarios. Ese problema originalmente se solucionó empleando técnicas primitivas que, frente a la elegancia y tesitura de una clase auxiliar, presentan tres inconvenientes, fuente de errores molestos, a veces sutiles y siempre difíciles de encontrar:

1. La cantidad de código a añadir emplea tipos especializados, infrecuentemente utilizados, cuya forma de uso muchas veces ha de ser buscada en la documentación pertinente. Esto no se puede evitar, pero si paliar al usarlos una vez en la clase…
2. La aritmética involucrada no es directa, hay que calcular el tiempo transcurrido mediante una expresión que está sujeta a ser erróneamente escrita si hay que codificarla varias veces, como es el caso. Aun con técnicas de *copia-y-empasta* se pueden deslizar (y replicar) errores, gastando tiempo en depuraciones, cada vez que se aplica código nuevamente escrito. Esto se puede y se debe minimizar al depurar esas porciones una vez en la clase….
3. El resultado obtenido es entero, pero se necesita un valor real para dar un tiempo más preciso, con uno o varios decimales, por lo que hay que convertir. Con las formas modernas de encasillamiento, el aplicarlas una y otra vez puede traer a colación más errores. Esto también se puede y debe minimizar una vez en la clase…

¿Una solución mejor? Crear una clase que se encargue de hacer la tarea

Por otra parte, la medida del tiempo es usada en la aplicación, pero no es parte de la abstracción del problema, sólo es un proceso auxiliar. Debería poder usarse en otros problemas. Esto es valor añadido de la clase...

Diseño del Cronógrafo

Se suelen confundir los términos cronómetro y cronógrafo. El primero es un reloj que ha sido calificado como tal por algún organismo reconocido de observación de la precisión de mecanismos o calibres; el que será estudiado es un reloj que permite la medición independiente de tiempos.

El cronómetro es un reloj de precisión; el cronógrafo mide tiempos

Normalmente, los cronógrafos van provistos de un pulsador de puesta en marcha o **start**, que también lo detiene (**stop**), así como otro segundo pulsador de reinicio, o **reset**. El resultado se obtiene según la posición de una aguja en el dial si es analógico, o por la cifra dada en una pantalla, si es digital. La clase Crono deberá brindar esa funcionalidad.

La estructura de datos

Es muy simple: una variable que contenga el valor del tiempo de inicio, una que registre el tiempo medido y una que brinde el estado del crono: si está contando el tiempo, o si no lo está. El registro del estado es tipo booleano, el tiempo medido debe ser un número real (**double**) y el tiempo de inicio debe ser del tipo especializado en mediciones de tiempo: **clock_t** o **time_t**.

La función **clock()** devuelve el tiempo que va midiendo el microprocesador a un tipo de dato entero. Este tiempo se mide en *clics* que va registrando el micro desde el arranque de la PC. El valor viene dado en unidades definidas para cada implementación particular y se convierte a segundos dividiéndolo por la macro **CLOCKS_PER_SEC**, cuyo nombre realmente significa clics por segundo. La función fue concebida para ser usada comparando el valor registrado contra uno obtenido de una llamada anterior. Si el entorno falla al regresar el tiempo, devuelve el valor -1, compatible con **EXIT_FAILURE**

Para poder usar esta clase como utilitaria, crear un subdirectorio llamado deposito en el de los ejemplos, crear la clase en un fichero y ponerlo allí. Como se va a necesitar, copiar el fichero **rnd.h** para allí también.

Declaración y definiciones

```
#ifndef CRONO_H_INCLUDED
#define CRONO_H_INCLUDED

//  crono.cpp: definiciones para el problema 3

#include <ctime>

/// @brief Crono: un cronógrafo. Galliano, 2020.
class Crono {
public:
    Crono() : comienzo( 0 ), contando( false ), transcurrido( 0 ) {}
    void start() {
        if ( not contando ) {
            contando = true;
            comienzo = clock();
        } // if
    }
    void stop() {
        if ( contando ) {
            contando = false;
            clock_t fin = clock();
            transcurrido += static_cast<double>( fin - comienzo ) / CLOCKS_PER_SEC;
        } // if
```

```
    }
    void reset() {
        contando = false;
        transcurrido = 0;
    }
    bool estaContando() const { return contando; }
        double reportar() {
        double resp;

        if ( contando ) {
            clock_t actual = clock();
            resp = static_cast<double>( actual - comienzo ) / CLOCKS_PER_SEC;
            resp += transcurrido;
            // si estaba parado, el tiempo ya está marcado en stop()
        } else {
            resp = transcurrido;
        } // if-else

        return resp;
    }
private:
    time_t comienzo;
    bool contando;
    double transcurrido;
};

#endif // CRONO_H_INCLUDED
```

Uso

Ahora se trabajarán medio millón de valores enteros aleatorios por pase y se comparará el rendimiento promedio de **qsort()** por cinco pases contra los métodos de ordenamiento del vector y la lista. El tiempo es más preciso.

```
//  Programa 3: cronógrafo
#include "../deposito/crono.h"
#include "../deposito/rnd.h"
#include <iostream>
#include <cstdlib>
#include <list>
#include <vector>
#include <iomanip>

const int N = 500000;
const int PASE = 5;

typedef std::list<int>     lInt;
typedef std::vector<int>   vInt;

// función de comparación de enteros
// input: dos vectores void
// output: 1 si *m > *n; -1 si *m < *n; 0 si *m == *n
inline int compInt( const void * m, const void * n ) {
    int p = *static_cast<const int *>( m );
    int q  *static_cast<const int *>( n );
    return p - q;
}

int main() {
    int m = 1, M = N * 0.5; // límites de los números generados
```

```
    std::cout <<
    "Programa 3: Cronometrando.\n\n"
    "Comparaci\242n las formas de ordenamiento de qsort(),"
    "\nvector y lista, haciendo " << PASE << " pases sobre " << N <<
    "\nenteros, entre " << m << " y " << M <<  ", generados al azar.\n\n"
    "Tenga un poco de paciencia. Trabajando...\n\n";

    // variables
    vInt vct( N );           // un vector
    lInt lst;                // una lista
    int * pAry = new int[N]; // un arreglo en la tienda libre

    // las marcas del tiempo y los totales
    Crono tiempo, tiempo_t;
    double t_vct = 0, t_lst = 0, t_qck = 0, t_relleno = 0, t_total = 0;
    srand( time( nullptr ) );

    tiempo_t.start();

    for ( int i = 0; i < PASE; ++i ) {
        std::cout << "Pase #" << i + 1 << '\r';

        // el tiempo de llenar nuevos valores
        tiempo.start();
        RELLENA_I( vct, m, M );
        lst.assign( vct.begin(), vct.end() );
        copy( vct.begin(), vct.end(), pAry );
        tiempo.stop();
        t_relleno += tiempo.reportar();
        tiempo.reset();

        // el tiempo de sort() sobre el vector
        tiempo.start();
        std::sort( vct.begin(), vct.end() );
        tiempo.stop();
        t_vct += tiempo.reportar();
        tiempo.reset();

        // el tiempo de la lista
        tiempo.start();
        lst.sort();
        tiempo.stop();
        t_lst += tiempo.reportar();
        tiempo.reset();

        // el tiempo del qsort()
        tiempo.start();
        qsort( pAry, N, sizeof( int ), compInt );
        tiempo.stop();
        t_qck += tiempo.reportar();
        tiempo.reset();
    } // for

    tiempo_t.stop();
    t_total = tiempo_t.reportar();

    std::cout << std::fixed << std::setprecision( 4 ) <<
    "\n\nTiempo total consumido: " << t_total << " s" <<
    "\nTiempo promedio consumido por:\n"
    "    Llenado: " << t_relleno / PASE << " s/pase.\n"
    "  Quicksort: " << t_qck / PASE     << " s/pase.\n"
    "  El vector: " << t_vct / PASE     << " s/pase.\n"
```

```
    "  La lista:  " << t_lst / PASE     << " s/pase.\n\n";

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

Salida

```
Programa 3: Cronometrando.

Comparación las formas de ordenamiento de qsort(),
vector y lista, haciendo 5 pases sobre 500000
enteros, entre 1 y 250000, generados al azar.

Tenga un poco de paciencia. Trabajando...

Pase #5

Tiempo total consumido: 4.0880 s
Tiempo promedio consumido por:
    Llenado: 0.0830 s/pase. ⬅
  Quicksort: 0.2052 s/pase. ⬅
  El vector: 0.0500 s/pase. ⬅
  La lista:  0.4782 s/pase. ⬅
```

La eficiencia es del 35%.

Debido al factor azaroso, cada ejecución daba valores diferentes, pero muy cercanos a los mostrados. Los tiempos promedio consumidos por pase fueron más precisos, aunque consistentes con los hallados en la salida del el problema 13 de (Introducción al C++, tomo 2, pág. 102), que acá se replica.

```
Programa 13: comparativa de ordenamiento

Comparación entre qsort() y las formas de ordenamiento
total de la STL, haciendo 5 pases sobre 500000 enteros,
entre 1 y 250000, generados al azar.

Tenga un poco de paciencia. Trabajando...

Pase #5

Tiempo total consumido: 9.4600 s
Tiempo promedio consumido por:
  genera valores = 0.0790 s/pase. ⬅
  qsort()        = 0.2022 s/pase. ⬅
  array: sort()  = 0.0444 s/pase. ⬅
  lista: sort()  = 0.0456 s/pase. ⬅
  stable_sort()  = 0.0622 s/pase.
  sort_heap()    = 0.1182 s/pase.
  partial_sort() = 0.1176 s/pase.
  list.sort()    = 0.4642 s/pase.
  multiset       = 0.7546 s/pase.
```

9-SOBRECARGA DE OPERADORES

> *También existe la sobrecarga de operadores que…le da más de una implementación a un operador. Editado de (Wikipedia en español)*

El lector ya conoce de estudios anteriores la sobrecarga de funciones. Ahora, en este acápite, se tratan dos nuevos temas conceptualmente interrelacionados: la sobrecarga de operadores y las funciones amigas, elementos indispensables para acercar la sintaxis de la clase a la de un operador nativo.

Sobrecarga de operadores

Es tentador convertirse en un súper entusiasta de la sobrecarga de operadores, pero es sólo un endulzamiento sintáctico, es otra manera de llamar a una función. Mirándolo desde esa perspectiva, no hay razón para sobrecargar un operador excepto si eso hace al código más sencillo e intuitivo de escribir y especialmente de leer. Recuerde: el código se lee mucho más que se escribe. Si éste no es el caso, ni se moleste. (Eckel, pág. 331)

Dicho esto, en C++ hay tres tipos de operadores según su aridad: operadores unarios, que actúan sobre un elemento (aridad 1); operadores binarios que actúan sobre dos elementos (aridad 2) y sólo un operador ternario que actúa sobre sobre tres elementos (aridad 3) y que no es posible sobrecargar.

- ✓ La aridad de un operador está dada por el número de argumentos necesarios para que se pueda aplicar.

Por ejemplo, el operador suma es binario porque necesita dos argumentos para poder realizar una suma y se simboliza como **`respuesta = lhs + rhs`**. **`lhs`** está situado a la izquierda del operador de suma, mientras que **`rhs`** (*right hand side* o a la derecha en español) identifica al otro término de la operación. El operador suma posee aridad 2.

Sintaxis

A la sobrecarga de operadores se le suele llamar azúcar sintáctico, porque lo único que aporta es la dulzura de la claridad de lectura e interpretación del código al lector humano, pero eso es precisamente uno de los objetivos declarados de B. Stroustrup, ceador del C++, quien dijo: *si un usuario desea un tipo que se comporte como cualquiera otro, puede definirlo dentro del mismo lenguaje.*

Sobrecargar un operador es definir dentro de una clase una función **`operator<op>`** en la que **`<op>`** representa el operador a sobrecargar. La aridad de sus argumentos depende de dos factores, si está definido como un método o como una función amiga:

1. Si el operador está definido como un método, va sin argumentos para los unarios y uno para los binarios: el argumento del lado izquierdo (**`lhs`**) del operador. El retorno siempre es **`*this`**.

```
T clase::operador<op>();                 // para un operador unario
T clase::operador<op>( argumento_lhs );  // para un operador binario
```

2. Si el operador está definido como una función amiga lleva un argumento para los unarios y dos para los binarios.

```
T operador<op>( argumento );                     // para un operador unario
T operador<op>( argumento_lhs, argumento_rhs );  // para un operador binario
```

Limitaciones

Primero, hay una media docena de limitaciones a respetar:

1. No se puede crear operadores nuevos. Por ej., no se puede definir un operador de elevación al cuadrado, porque C++ no lo tiene.
2. Un operador siempre se sobrecarga con relación a una clase.
3. El operador sobrecargado no puede violar las reglas sintácticas de C++
4. Ni la aridad, ni el orden de precedencia del operador pueden ser cambiados.
5. Los operadores sobrecargados siempre deben regresar un objeto de su clase.
6. No se pueden sobrecargar los operadores dados en la tabla que se muestra a continuación**¡Error! No se encuentra el origen de la referencia.**.

Tabla 5. (simplificada) Operadores que no pueden sobrecargarse

Operador	Descripción
`.`	de membresía
`::`	de resolución de ámbito
`()?:`	condicional ternario
`sizeof()`	de tamaños de tipo
`const_cast, static_cast, reinterpret_cast, dynamic_cast`	de encasillamientos
typeid()	de identificación de tipos durante la ejecución, o RTTI

Todo lo anterior es obvio, de lo contrario ¡los programas serían un caos!

Y luego hay que valerse del sentido común —llamado popularmente *el menos común de los sentidos*— para saber emplear los lineamientos adecuados en su aplicación. Aunque es tentador crear nuevos operadores que actúen en casos difíciles, es casi seguro que se producirá código ofuscado, y eso no es bueno. Por ejemplo, aplicar el operador de división para dividir una cadena. Si en un caso aparecen buenas razones para hacerlo, lo indicado es actuar muy cautelosamente, siempre teniendo presente que los grandes objetivos de la sobrecarga son el incremento de la comprensión del código y la facilidad de uso.

Los tres operadores que sólo pueden sobrecargarse usando métodos son (**1**) el de asignación `=`; (**2**) el de acceso por compensación `[]`; y (**3**) el de agrupación `()`. En la tabla que sigue se muestran los operadores que pueden sobrecargarse usando métodos y funciones amigas.

Tabla 6. (simplificada) Operadores que pueden sobrecargarse de ambas formas

`+`	`-`	`*`	`/`	`%`	`^`
`&`	`|`	`~`	`!`	`>`	`<`
`>=`	`<=`	`==`	`!=`	`<<`	`>>`
`->`	`+=`	`-=`	`*=`	`/=`	`%=`
`^=`	`&=`	`|=`	`,`	`<<=`	`>>=`
`||`	`&&`	`++`	`--`		

Si no son matemáticos, usualmente la sobrecarga de operadores puede ofuscar más que esclarecer el código, anulando el objetivo fundamental de “endulzar” su lectura.

Considerar sobrecargar solamente los operadores matemáticos

Funciones amigas

Las amigas son funciones que sirven de extensoras de la interfaz de una clase. Son especiales en tanto que pueden acceder directamente a los miembros privados de la clase a que sirven y son muy útiles en determinadas ocasiones.

- ✓ La interfaz de la clase es un concepto muy importante. El objetivo en C++ es que la clase, cuando se concrete en un objeto, se comporte lo más posible como un tipo nativo de dato, extendiendo lo más posible su interfaz.

Aunque las clases e incluso, sus miembros también pueden tener la cualidad de amistad, a este nivel sólo se considera el estudio de las funciones amigas.

Dado que la amistad es dada solamente por la clase y no es tomada ciegamente por cualquiera, una función amiga es eso, una función que recibe un listado de parámetros y devuelve algún valor, pero lo que la caracteriza es que su trabajo estará definido por —y relacionado con— la clase que le otorga la amistad, y a la cual sirve. No existe *per se*; existe para servir a una clase dada.

Una función amiga se declara dentro de la clase, en cualquier lugar de la misma (pero es costumbre ponerla dentro de la sección pública, al inicio o al final) como: **`friend <clase>|<T> operator<op>(parámetros)`** y se define fuera de ella como: **`<clase>|<T> operator<op>(parámetros) { ... }`**

Todo lo que se dijo respecto a las limitaciones que se enfrentan al sobrecargar métodos es válido para estas funciones, y además, una función amiga *no es un miembro de la clase que sirve*, por lo que no posee el puntero **`*this`** y la clase que le emplea, para poder obtener el acceso total, debe de ser pasada como uno de sus parámetros, preferiblemente por referencia, y siempre que se pueda, constante.

Referencias

Ya vistas en los tomos anteriores de esta serie, se le recuerda al lector que las referencias son como unos punteros especializados que se comportan como cualquier nombre de variable, y que en este contexto existen dos modos de utilizarlas: pueden pasarse a una función y pueden ser devueltas por una función.

A las referencias, no se les aplica el operador de desreferencia * pues produce un error de compilación

El pase de parámetros por referencia ofrece varias ventajas sobre el pase con punteros, pero quizás el más importante en este momento es que al referenciar a un objeto, se obvia el constructor de copia: ni se construye ni se destruye nada. Esto es mucho más eficiente y desde luego, mucho más seguro. Y así se hace con la función amiga, que necesita conocer la clase a la que sirve. Es muy importante entender que una referencia no es un puntero común y cuando se pasa un objeto por referencia, el operador de acceso a un atributo es el punto (**`.`**) y no la flecha (**`->`**)

Nota: aunque se debe conocer que existen las referencias independientes, usualmente solo aportan código ofuscado, lo cual trae confusión al lector del programa. Si la norma no las quitó es porque no encontró razones para hacerlo, pero el programador novel tampoco tiene razones de usarlas.

Al crear una referencia debe inicializarse excepto cuando lo hace a atributos de una clase, valores devueltos por métodos o funciones, o parámetros pasados a métodos o funciones. Las restricciones que se aplican a toda referencia son:

1. Una referencia no puede referenciarse.
2. No se puede crear arreglos de referencias.
3. No se puede referenciar un campo de bits, tema que no se trata en la serie.

Una clase para un vector geométrico

El vector tal y como se verá aquí NO es el de la STL. Este vector es físico, representa a una magnitud física y se caracteriza por tener una longitud y un sentido. Algunos ejemplos de magnitudes vectoriales son la velocidad y la aceleración con que se desplaza un móvil, la fuerza mecánica o eléctrica que actúa sobre un objeto, el desplazamiento de un objeto, etc.

Un vector plano se puede expresar como un par de números reales que representan sus coordenadas cartesianas, es decir, perpendiculares entre sí, estableciendo un plano de la forma $\vec{\boldsymbol{A}} = (a_x, a_y)$ donde a_x representa la longitud de su componente medida por la dirección horizontal, y a_y la de su componente medida por la dirección vertical. Pero también puede representarse por un par de números reales que representan sus coordenadas polares, es decir, medidas desde un polo (usualmente el origen de coordenadas cartesianas) de la forma $\vec{\boldsymbol{A}} = (|\mathrm{A}|, \theta)$, donde el ángulo está dado en grados sexagesimales.

La norma, módulo o longitud de un vector está dada por $|\boldsymbol{A}| = \sqrt{a_x^2 + a_y^2}$

El ángulo entre dos vectores es calculado como $\cos\theta = \frac{\boldsymbol{A}\cdot\boldsymbol{B}}{|A||B|}$

El producto-punto entre dos vectores es el número $n = \boldsymbol{A}\cdot\boldsymbol{B} = [a_x \quad a_y]\begin{bmatrix} b_x \\ b_y \end{bmatrix} = (a_x \times b_x) + (a_y \times b_y)$

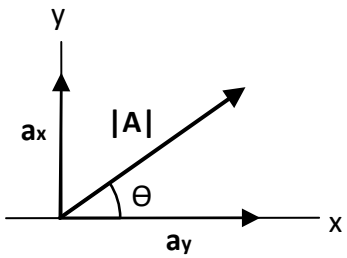


Ilustración 6. Componentes de un Vector

Hay dos operaciones que son especiales para un vector cualquiera: la suma entre dos vectores y el producto de un vector por un escalar, lo cual hace al ejemplo el modelo típico de representación de una clase como un dato nativo, aplicando la sobrecarga de operadores.

Programa 4-Una clase Vector

Son dos las formas de cómo representar un mismo vector plano: la notación polar, dando su módulo y dirección; o la notación cartesiana, dando el módulo de sus dos componentes. La representación dual de un vector hace que el objeto esté en uno de dos estados: polar o cartesiano. Un miembro de la clase debe controlar el estado de la misma en un momento dado; a dicho elemento comúnmente se le llama *miembro de estado*.

Esto es habitual en los casos en que un valor se representa por dos o más dualidades, como ocurrió con el problema de la emulación de una calculadora de conversión, y ya fue tratado en los dos primeros tomos de la serie, fuera del contexto de las clases.

Declaración

Un vector físico plano tiene dos componentes x, y; sabe cómo mostrarse en modo cartesiano o polar, calcular su producto punto con otro vector, y el coseno del ángulo entre ambos; además sabe clasificar su posición respecto al otro vector. Puede ser descrito de dos formas equivalentes: en coordenadas cartesianas **(x, y)** o en coordenadas polares **(M, A)**, que se refiere a (Módulo, Ángulo), este último en grados sexagesimales

Por diseño, el autor tomó la decisión de guardar las componentes cartesianas en sendas variables y hacer el cálculo cuando se necesite presentar el valor del vector físico en coordenadas polares. Este diseño es eficiente para aplicaciones que casi nunca ocupan la vista del vector en coordenadas polares. Podrían haberse tomado otras decisiones de diseño y eso es parte de la construcción eficaz de una clase bien formada.

Se usó un **namespace** para evitar conflictos de nombres. Se define un constructor parametrizado que también suple al que C++ pone por omisión. El miembro **modo** registra el estado de vector para su muestra a un periférico de salida, y para su manejo se definen accesores. Hay solamente un método y es para entrar las coordenadas del vector. El valor del factor de conversión se calcula.

Se sobrecargan los operadores necesarios para hacer álgebra vectorial plana: la suma, diferencia y cambio de signo, además del producto interno y del de un escalar por un vector. El operador de asignación permite el cambio de modo en una forma natural.

Para completitud de la interfaz se definen funciones amigas que calculan el producto de un vector por un escalar, pues si matemáticamente ambas operaciones son lo mismo, no es así al codificar. El cálculo del producto interno se acerca a la notación matemática; para ello se usan dos funciones amigas que se ven más de acuerdo con la notación vectorial: una para tomar el módulo y la otra para adquirir el ángulo de un vector contra el eje horizontal. Sólo hay tres miembros: el que registra el estado de la salida y las dos coordenadas.

```
#ifndef VECTORES_H_INCLUDED
#define VECTORES_H_INCLUDED

#include <string>
#include <cmath>

namespace miVector { // <--
const double GRAD2RAD = 45.0 / atan( 1.0 ); // 57.29578...
enum Modo { Polar, Cartesiano };

class Vector {
public:
    // constructor
    Vector( double a = 0, double b = 0 );
    // accesores
    bool getModo() const { return modo == Cartesiano; }
    void setValX( double valor ) { x = valor; }
    void setValY( double valor ) { y = valor; }
    // operadores sobrecargados
    Vector operator+( Vector & b ) const; // suma A + B
    Vector operator-( Vector & b ) const; // diferencia A - B
    Vector operator-()             const; // cambio de signo -A
    Vector operator*( double n )   const; // producto nA
    void   operator=( Modo valor );       // asignación del modo
    // amigas, al final de la parte pública
    friend Vector operator*( double n, const Vector & a );         // producto An
    friend double operator*( const Vector & a, const Vector & b ); // producto A.B
    friend double coseno( Vector & a, Vector & b );                // coseno <A,B>
    friend double modulo( const Vector & v );                      // módulo del vector
```

```
    friend double angulo( const Vector & v );                    // ángulo del vector
    // operadores de E/S
    friend std::ostream & operator<<( std::ostream & os, const Vector & v );
    friend std::istream & operator>>( std::istream & is, Vector & v );
private:
    //  miembros
    double x, y; // sus dos coordenadas
    Modo modo;   // estado del vector
};
} // namespace miVector <--

#endif // VECTORES_H_INCLUDED
```

Definiciones

Como se definió un constructor parametrizado, éste actúa también como constructor de oficio y si se obvian todos los valores, pone las coordenadas a cero y el modo en cartesiano. En caso contrario toma los valores pasados y los pone en el vector. El método entrar es el más complejo, pues debe discriminar cómo entran los valores: si el modo es cartesiano, van directo al vector, y si no, entonces son convertidos antes de ponerlos en el vector, pero ambas posibilidades se hacen con una lectura muy flexible que permite entrar un par de coordenadas como **(a,b)**, **(a, b)**, **(a, b)**, etc. y vale la pena estudiarlo.

Esta técnica lee dos elementos que no son números (paréntesis y coma), evita los espacios en blanco (**skipws**) antes de leer cualquier número, y limpia el búfer con una sola instrucción (**cin.ignore(255, '\n')**) que significa: *descarta los primeros 255 caracteres que te encuentres, o hasta que te encuentres con un cambio de línea*.

Usualmente el tamaño de los búferes es de 255 (**0xFF**), 512 (**0x200**) ó 1024 (**0x400**) caracteres

---

Para cambiar el formato de cómo se ve el vector **A**, digamos de cartesiano a polar se pone **A = Polar**. El estado del vector para su vista es controlado por los accesores **get/set** y las operaciones con sobrecarga de operadores. En el caso de los operadores se devuelve el valor correcto construyendo un vector anónimo y pasándole los valores, lo cual es muy eficiente, pues la construcción, pase y destrucción del objeto es en una sola instrucción. La interfaz se completa con funciones amigas que acercan aún más la clase a un tipo nativo de datos. Por ej., para obtener el módulo del vector **A** se pone **modulo(A)** y para sumar dos vectores se pone **A + B**. Al mostrar un vector, éste comprueba su estado y se ve en dispositivo de salida con el formato deseado.

```
#include "vectores.h"
#include <iomanip>
#include <ios>
#include <iostream>

using std::cout;
using std::cin;
using std::string;

namespace miVector { // <--
// el constructor parametrizado toma los valores y los fija
Vector::Vector( double a, double b ) : x( a ), y( b ), modo( Cartesiano ) {}

// operaciones: son la suma y resta, el producto interno,
// el de un vector por un escalar y la negación del vector
Vector Vector::operator-() const { return Vector( -x, -y ); }                      // -A
Vector Vector::operator+( Vector & b ) const { return Vector( x + b.x, y + b.y ); } // A + B
Vector Vector::operator-( Vector & b ) const { return Vector( x - b.x, y - b.y ); } // A - B
Vector Vector::operator*( double n )   const { return Vector( n*x, n*y ); }        // n x A
void   Vector::operator=( Modo valor ) { modo = valor; }                           // A = modo
```

```
// para completamiento de la interfaz
Vector operator*( double n, const Vector & a ) { return a * n; }        // A x n
double modulo( const Vector & v ) { return sqrt( v.x*v.x + v.y*v.y ); } // |A|

// ángulo sita (rad.)
double angulo( const Vector & v ) {
    double resp;

    if ( 0 == v.x ) {
        resp = 0;
    } else {
        resp = atan2( v.y, v.x );
        resp *= GRAD2RAD;
    } // if-else

    return resp;
}

// coseno entre vectores
double coseno( Vector & a, Vector & b ) { return ( a*b ) / ( modulo( a ) * modulo( b ) ); }
double operator*( const Vector & a, const Vector & b ) { return ( a.x*b.x + a.y*b.y ); } // A.B

// muestra un vector en su formato, dependiendo del modo
std::ostream & operator<<( std::ostream & os, const Vector & v ) {
    std::ios::fmtflags oldFlags = os.flags();
    os << std::setprecision( 2 );

    if ( Cartesiano == v.getModo() ) {
        os << "(" << v.x << ", " << v.y << ")";
    } else {
        os << "(" << modulo( v )  << ", ";
        os << std::setprecision( 3 );
        os << angulo( v ) << ")";
    } // if-else

    os.flags( oldFlags );
    return os;
}

std::istream & operator>>( std::istream & is, Vector & v ){
    char parentesis, coma;
    double x, y;
    is.setf( std::ios::skipws );   // evita los espacios en blanco
    is >> parentesis;              // lee el paréntesis
    is >> x;                       // lee la 1ra coordenada
    is >> coma;                    // lee la coma
    is >> y;                       // lee la 2da coordenada;
    is.ignore( 255, '\n' );        // deja limpio el búfer
    is.unsetf( std::ios::skipws ); // retorna al valor por defecto
    v.setValX( x );
    v.setValY( y );
    v = Cartesiano;
    return is;
}
} // namespace miVector <--
```

Uso

El driver usa la clase para emular las soluciones dadas a este problema en (Introducción al C++, tomo 1, pág. 125). Esta abarca más elementos, lleva más código y está más de acuerdo con el acercamiento a un tipo nativo. Una vez depurada y probada, la clase puede ser usada en múltiples aplicaciones sin cambio alguno.

```
#include "vectores.h"
#include <iostream>
#include <cstdlib>
#include <iomanip>

using namespace miVector; // <--
const int PRES = 2;

int main() {
    using std::cout;

    cout << "Problema 4: uso de la clase Vector.\n\n";
    Vector A, B;
    cout << "Datos del vector en formato (x, y)\n";
    cout << "A: ";
    std::cin >> A;
    cout << "B: ";
    std::cin >> B;

    cout << std::fixed << std::setprecision( 2 );
    A = Polar; cout << "\nEn modo polar, A: " << A;
    B = Polar; cout << "\nEn modo polar, B: " << B;

    cout << "\n\n" <<
    "El producto-punto de A y B vale " << A * B << "\n";
    double sita = coseno( A, B );
    cout  << "El coseno entre A y B es de " << sita << " rad o "
          << std::setprecision( 0.5*PRES )  << sita * GRAD2RAD << " grd.\n\n";

    if (  0 == sita ) cout << "Son perpendiculares.\n\n";
    if (  1 == sita ) cout << "Son paralelos.\n\n";
    if ( -1 == sita ) cout << "Son anti-paralelos.\n\n";

    cout << std::setprecision( PRES );
    Vector C( A + B );
    cout << "La suma de ambos es " << C; C = Polar;
    cout << " que en modo\npolar tambi\202n se expresa as\241: " << C;

    cout << "\n\nEn modo cartesiano:"
         << "\nA multiplicado por 3 vale " << A * 3
         << "\nB revertido vale " << -B << "\n\n";

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

Salida

```
Problema 4: uso de la clase Vector.

Datos del vector en formato (x, y)
A: (9.8,6.5)
B: (5.4,2.1)

En modo polar, A: (11.76, 33.555)
En modo polar, B: (5.79, 21.251)

El producto-punto de A y B vale 66.570
El coseno entre A y B es de 0.977 rad o 56.0 grd.

La suma de ambos es (15.20, 8.60) que en modo
polar también se expresa así: (17.46, 29.501)

En modo cartesiano:
A multiplicado por 3 vale (29.40, 19.50)
B revertido vale (-5.40, -2.10)
```

La eficiencia de la respuesta es del 30%.

10-TEMAS DE LA HERENCIA

> *En la OOP, después de la formación de clases por composición, la herencia es el mecanismo utilizado para alcanzar dos de los objetivos más preciados en el desarrollo moderno del software: la reutilización y la extensibilidad. Los diseñadores, partiendo de una clase o de una jerarquía de clases preexistente, ya comprobadas y verificadas, pueden crear nuevas clases evitando con ello el rediseño, la modificación y la verificación de la parte ya implementada. Editado de (Wikipedia en español)*

Herencia simple

¿Qué se quiere decir cuando se indica que algo (o alguien) es un tipo de otra cosa? Se quiere decir que es una *especialización*. Por ej., en Geometría Plana una elipse deriva de una sección cónica, porque es una de sus especializaciones y todo cuanto se aplica a una sección cónica, se aplica a una elipse, incluyendo sus restricciones. Así pues, en este ejemplo la herencia correctamente aplicada indica que la sección cónica es la clase-madre y la elipse es la clase-hija. Pero un círculo es una elipse especializada, donde sus dos ejes son iguales, por lo que aquél deriva de ésta, como indica la figura y no al revés, como pudiera pensarse.

El concepto de herencia permite aplicar la otra gran técnica de creación de clases: *el desarrollo por diferencias*. Las clases se agrupan y en lugar de rescribirlas cada vez que se le añaden nuevas cualidades al proyecto, se reutiliza todo aquello que le sea general y se implementa lo nuevo.

El diagrama ilustra la relación jerárquica de herencia *es-un/es-una*, y se lee así: una Circunferencia es una Elipse; una Elipse es una Sección Cónica. Una Parábola es una Sección Cónica. Una Hipérbola es una Sección Cónica. O sea, la Circunferencia hereda lo suyo a través de la Elipse y esta, junto a las demás figuras, hereda directamente de la Sección cónica.

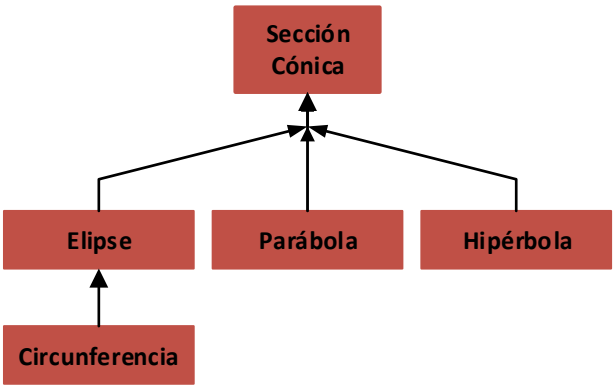


Ilustración 7. Una jerarquía geométrica

La regla es: si una clase (dígase ***B***) hereda de otra (dígase ***A***), necesariamente es porque la primera puede ser considerada y/o manipulada como si fuera la segunda. Debe existir un *vínculo* que propicie el encontrar los datos que les son comunes.

Esta regla permite además conocer el sentido de la herencia: para que ***A*** sea la clase base, superclase o clase-madre, todo lo que se dedica a ella debe poder ser aplicado para la clase ***B***, su clase derivada, subclase o clase-hija, incluyendo las restricciones que pudiera tener ¡Pero mucha atención!

Aunque uno de los objetivos de la herencia es evitar la rescritura de código, no debe ser el que rija su aplicación

¿Cómo se deriva una clase?

C++ ofrece tres tipos de herencia: **`public`**, **`protected`** y **`private`**, o pública, protegida y privada en español. El texto tratará exclusivamente con la herencia pública, donde todo objeto de una clase-hija es considerado también un objeto de su clase-madre; sin embargo, los objetos de la clase-madre no son objetos de sus clases-hijas, a menos que estén protegidos. Bajo el concepto de herencia pública, si un perro (genérico) es descrito como clase-madre y un Pastor Alemán (específico) lo es como clase-hija entonces, aunque todos los pastores alemanes son perros, no todos los perros son pastores alemanes.

Todos los elementos públicos de la clase-madre también son públicos en las clases-hija; todos los elementos protegidos de la clase-madre son visibles desde las clases-hija; y todos los elementos privados de la clase-madre no pueden ser vistos directamente por las clases-hijas. Esto quiere decir que los elementos públicos y protegidos se llaman en la clase-hija sin más, pero los privados requieren de accesores.

Las relaciones de herencia forman estructuras jerárquicas, donde una clase-madre posee una relación con sus clases-hijas. Usualmente los elementos pertenecientes a una clase-madre son mucho más numerosos: perros hay muchos, pastores alemanes hay menos; árboles hay muchos, árboles de naranjas o de plátanos hay menos, etc.

Una vez que se define una clase en una jerarquía, su posición relativa en el árbol de herencia la hace ser (a partir de allí) una clase-madre si suministra más clases, o una clase-hija si es un nodo terminal.

Para derivar públicamente una clase de otra se pone:

```
class ClaseMadre : public ClaseHija { ... };
```

Una clase hereda su constructor de copia

Un constructor de copia, lo hace de un objeto para otro objeto nuevo. Entonces, es usado (entre otras cosas) para crear un objeto durante la inicialización, incluyendo los argumentos, que pasan por valor. No se usa en asignaciones comunes. Para una clase normalmente tiene este prototipo:

```
clase ( const clase & valor );
```

El constructor de copia hace una copia fiel, por lo que no sirve si se usa manualmente la tienda gratis en la clase: hay que hacer uno a mano.

Una clase hereda su operador de direcciones

Un operador de direcciones permite inicializar una clase con un objeto. Tiene este prototipo:

```
clase &rMiObjeto = objeto_de_la_clase;
```

El operador de direcciones hace una asignación fiel, por lo que no sirve si se usa manualmente la tienda gratis en la clase: hay que hacer uno a mano.

¿Qué no se hereda de una clase?

De inicio una clase-hija hereda todos los miembros y métodos de su case-madre menos:

- Sus constructores y su destructor.
- Su operador de asignación.
- Los datos estáticos de la clase.
- Sus amigos.

Un operador de asignación es uno sobrecargado que toma y devuelve una referencia a un objeto de esa clase y lo hace implícitamente. Tiene el siguiente prototipo

```
clase &operator=( const clase &valor );
```

Un operador de asignación hace una asignación fiel, por lo que no sirve si se usa manualmente la tienda gratis en la clase: hay que hacer uno a mano.

Programa 5-desarrollo por herencia pública

Este ejemplo muestra una de las muchas técnicas de desarrollo: partir de una clase muy genérica ir a las específicas en la medida en que el proyecto lo requiera. Para ilustrar (**1**) lo que la clase-hija hereda de su clase-madre; y (**2**) cómo lo hace, se verán una naranja y un plátano como hijos de una clase genérica fruta. Es muy simple debido a su carácter eminentemente didáctico, dada solamente para ilustrar ambos puntos.

Declaración

Esta sencilla clase tiene dos miembros (que poseen todas las futas) un nombre y un color. El color y el nombre son privados y se extraen con sendos **geters**.

```
#ifndef FRUTAS_H_INCLUDED
```

```
#define FRUTAS_H_INCLUDED

#include <string>
typedef std::string Str;

// Clase-madre Fruta
class Fruta {
public: // objetos públicos
    Fruta( Str nb, Str cl ); // constructor
    // accesores
    Str getNombre() const;
    Str getColor()  const;
    void muestra(); // método
private: // miembros privados
    Str nombre;
    Str color;
};

// Clase-hija Naranja
class Naranja : public Fruta {
public: // objetos públicos
    Naranja( Str nb, Str cl, bool co, bool ju );
    void muestra(); // método sobrescrito
private: // miembros privados
    bool comestible;
    bool jugosa;
};

// Clase-hija plátanos
class Platano : public Fruta {
public: // objetos públicos
    Platano( Str nb, Str cl, bool co );
    void muestra(); // método sobrescrito
private: // miembros privados
    bool comestible;
};

#endif // FRUTAS_H_INCLUDED
```

Definición

Usan una lista para entrar sus datos; las clases derivadas los entran después que su clase-madre. Los **getters** y la vista independiente son triviales. El método muestra las partes comunes de las frutas.

```
#include "frutas.h"
#include <iostream>
using std::cout;

// clase base Fruta
Fruta::Fruta( Str nb, Str cl ) : nombre( nb ), color( cl ) {}

//  getters
Str Fruta::getNombre() const { return nombre; }
Str Fruta::getColor()  const { return color;  }

//  método que muestra las partes comunes
void Fruta::muestra() { cout << getNombre() << " es de color " << getColor() << '\n'; }

// Clase específica Naranja
Naranja::Naranja( Str nb, Str cl, bool co, bool ju )
```

```
    : Fruta( nb, cl),                   // primero va la madre
      comestible( co ), jugosa( ju ) {} // luego, la hija

// vista
void Naranja::muestra() {
    Fruta::muestra(); // muestra la parte común
    // y ahora las partes específicas
    jugosa ? cout << "Es " : cout << "No es ";
    cout << "buena para zumo.\n";
    comestible ? cout << "Es " : cout << "No es ";
    cout << "buena para comer.\n\n";
}

// Clase específica Platano
Platano::Platano( Str nb, Str cl, bool co )
    : Fruta( nb, cl ),      // primero va la madre
      comestible( co ) {}  // luego, la hija

// vista
void Platano::muestra() {
    Fruta::muestra(); // muestra la parte común
    // y ahora la parte específica
    comestible ? cout << "Es " : cout << "No es ";
    cout << "muy bueno para comer.\n\n";
}
```

Uso

Las variedades de la frutas tratadas son las de naranja Cadenera y plátano Manzano, cuyos parámetros fueron tomados de (Wikipedia en español) Se muestran la forma de uso para la tienda gratis y la que emplea la pila.

```
#include "frutas.h"
#include <iostream>
#include <cstdlib>

int main() {
    std::cout << "Programa 5: herencia.\n\n";

    // como un puntero a la clase
    Naranja * pNar = new Naranja( "La naranja Cadenera", "naranja", true, true );
    pNar->muestra();
    delete pNar;

    // como una variable de la clase
    Platano Pla( "El pl\240tano Manzano", "amarillo", true );
    Pla.muestra();

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

Salida

```
Programa 5: herencia.

La naranja Cadenera es de color naranja
Es buena para zumo.
Es buena para comer.

El plátano Manzano es de color amarillo
Es muy bueno para comer.
```

La eficiencia de la respuesta es del 38%.

Polimorfismo

La función virtual es un concepto muy importante del polimorfismo en la POO. Si una función es designada virtual, se llamará a la función de la clase derivada, si existe. Si no es virtual, se llamará a la función de la clase base. (Wikipedia en español)

En programación orientada a objetos (POO), una función virtual (método virtual) es una función cuyo comportamiento es determinado por la redefinición de una función con la misma firma en alguna de sus subclases. En C++ una clase se hace virtual anteponiéndole el atributo **`virtual`**. Una clase virtual se hace pura igualándola a cero.

El estudio del polimorfismo y la herencia múltiple, por un lado ya elevan el nivel del texto a medio

Clases virtuales

Uno de los lineamientos básicos de Stroustrup al diseñar C++ fue que el usuario *pagara solamente por lo que deseara*. El enlace tardío tiene su costo de tiempo en la ejecución del programa y de espacio en el código binario, por lo que C++ adopta el enlace temprano por omisión y hay que decirle explícitamente si se requiere del otro.

Si una clase-madre define un método y una clase-hija lo redefine, al llamar este método desde cualquiera de ellas sólo responderá el de la clase-madre, porque fue el que se vinculó primero al momento del enlace. Eso fue enlace temprano, pero si lo que se busca es polimorfismo, cada clase de la jerarquía deberá llamar su método particular y el programa debe reconocerlo a través del enlace tardío.

Si se antepone la palabra-clave **`virtual`** a la declaración de cualquier método de la clase, se dice al compilador que se requiere un enlace tardío para ese método.

Constructores y destructores

Las funciones hacen actuar al objeto según el tipo que recibió cuando fue creado. En cierto sentido, el constructor le da un objeto a su tipo. No se le puede pedir a un objeto que se construya de forma apropiada a su tipo antes de la llamada al constructor, porque el objeto aún no existe.

El constructor para un objeto polimórfico, es decir, de una clase con una función virtual como mínimo, pone “algo” en el objeto para describir su tipo dinámico. Las subsiguientes llamadas de funciones virtuales que se aplican a un objeto de esa

jerarquía, usan ese “algo” para encaminar las peticiones correctamente, por lo que el propio constructor no puede ser virtual ya que el objeto aún no existe.

Los destructores si pueden —y tienen que— ser virtuales, porque cada objeto en la jerarquía puede tener requerimientos diferentes respecto a sus recursos y distintos modos de gestionar su destrucción. De hecho, si se sospecha que una clase que hacemos será usada como clase-madre por terceras personas, es menester que su destructor sea virtual. Es más, no es una mala idea declarar virtuales a todos los destructores.

Si el destructor es virtual puede eliminar al objeto mediante un puntero a su clase-madre con la confianza de que el destructor apropiado será llamado. Si una clase es virtual, su destructor también ha de serlo.

Borrar un puntero virtual con un destructor no-virtual no está definido; el programa pierde la integridad

Sobrescritura y sobrecarga de métodos

Si una clase-hija duplica un método de su clase-madre, pero con un algoritmo diferente, lo ha sobrescrito. Si define métodos con el mismo nombre, pero diferente firma, los ha sobrecargado y ahora hay varios con el mismo. Ambas acciones son similares, pero sus actuares no.

Si se sobrecarga un método, todos los métodos de la clase-madre con dicho nombre son ocultados y esto —si lo que se busca es polimorfismo— es un efecto colateral indeseado y hay que remediarlo. Cuando ese método que se sobrescribe es declarado virtual y se aplica un puntero al objeto, se le puede aplicar polimorfismo.

Sobrescribir un método virtual permite usar polimorfismo; sobrecargarlo no

Es un error común el ocultar un método constante en la clase-madre cuando la intención es de sobrescribirlo, olvidando incluir en la hija la palabra-clave **`const`**, que es parte de su firma. Al duplicarlo allá, se consigue el efecto colateral de ocultarlo e impide también el hacer polimorfismo. ¡Cuidado!

Clase virtual pura

Una función virtual pura no puede ser definida: está allí solamente para obligar a sus hijas a implementarla.

Una clase virtual pura se consigue al afectar un método virtual igualándolo a cero. Por ej., para hacer virtual pura a la clase mamífero basta poner **`virtual void Hablar() const = 0;`**. Si trata de implementarse en una clase que no sea virtual pura ocurre el siguiente error:

```
||=== Build: Release in prob06 (compiler: GNU GCC Compiler) ===
In function 'int main()':
error: invalid new-expression of abstract class type 'Perro'
note:  because the following virtual functions are pure within 'Perro':
note:  virtual void Perro::Hablar() const
||=== Build finished: 1 error(s), 0 warning(s) (0 minute(s), 2 second(s)) ===
```

que establece que la clase **`Perro`** es virtual-pura porque tiene el método **`Hablar()`** igualado a cero. Por otro lado, si se retira la función **`Hablar()`** —o cualquier otra— de la clase **`Perro`**, se obtiene otro error que establece que es abstracto, pero hace referencia a la función virtual pura de su madre y no puede poner ese objeto en memoria.

```
||=== Build: Release in prob06 (compiler: GNU GCC Compiler) ===
In function 'int main()':
error: invalid new-expression of abstract class type 'Perro'
note:  because the following virtual functions are pure within 'Perro':
note:  virtual void Mamifero::Hablar() const
||=== Build finished: 1 error(s), 0 warning(s) (0 minute(s), 2 second(s)) ===
```

Resumiendo:

- Una función virtual pura hace pura a una clase que está allí para ser sobrepasada por sus hijas y no puede ser llamada, aunque puede ser —y es— referida por un puntero.
- Todas las funciones virtuales de la clase-madre deben ser sobrescritas en sus clases-hijas.
- No es menester repetir la palabra-clave **`virtual`** en las clases derivadas: ellas heredan la virtualidad. No obstante, en aras de una mayor claridad de lectura del código, es costumbre el repetirlas.

Eso es la esencia del polimorfismo aplicado a las clases

Programa 6-polimorfismo

Se presenta un programa con funcionalidad mínima, donde se muestra lo básico del polimorfismo: un perro y un gato son referidos como animales con un único puntero.

Declaración

Todos los mamíferos de alguna suerte pertenecen a una raza, por lo que esa cualidad es privada y tiene un **`geter`**. Además, todos los métodos que usarán las clases-hijas deben ser codificados virtuales en la clase-madre, incluyendo sus parámetros si los trae, aunque no hagan nada. Ya por el hecho de ser virtuales en la clase-madre lo son en las hijas, aunque no se les declare, pero por razones de claridad se recomienda hacerlo.

Un perro y un gato heredan de una clase común: **`Mamifero`**, que se hace virtual pura. Dos métodos no reciben parámetros, otro recibe uno. Cada uno de ellos posee un miembro privado (son distintos en identificación, pero también pudieran ser diferentes en tipo) y se saca con su **`geter`**.

```
#ifndef MAMIFERO_H_INCLUDED
#define MAMIFERO_H_INCLUDED

#include <iostream>

class Mamifero {
public:
    Mamifero() : raza( "" ) {}
    Mamifero( std::string rz );
    virtual ~Mamifero() {}
    // acesores
    void setRaza( std::string valor ) { raza = valor; }
    std::string getRaza() const { return raza; };
    // métodos
    virtual void Hablar() const = 0;// <-- clase virtual pura
    virtual void Moverse( int n ) const {};
    virtual void Mostrarse() const {};
private:
    std::string raza;
};

class Perro : public Mamifero {
public:
    Perro() : nombre( "" ) {}
    Perro( std::string rz, std::string nb );
    virtual ~Perro () {}
    void setNombre( std::string valor ) { nombre = valor; }
    std::string getNombre() const { return nombre; }
```

```
    virtual void Hablar() const;
    virtual void Moverse( int n ) const;
    virtual void Mostrarse() const;
private:
    std::string nombre;
};

class Gato : public Mamifero {
public:
    Gato() : color( "" ) {}
    Gato( std::string rz, std::string cl );
    virtual ~Gato () {}
    void setColor( std::string valor ) { color = valor; }
    std::string getColor() const { return color; }
    virtual void Hablar() const;
    virtual void Moverse( int n ) const;
    virtual void Mostrarse() const;
private:
    std::string color;
};

#endif // MAMIFERO_H_INCLUDED
```

Definición

La clase **Mamifero** hace de clase-madre. Se declaran virtual el destructor y todos sus métodos, pero si solo se declarara un método virtual, también sería virtual. Pero además, el método **Hablar()** se iguala a cero, haciendo de esta clase una virtual pura, es decir, sólo sirve de soporte de la jerarquía y no pueden haber objetos directos de ella, aunque sí pueden haber —y hay— punteros. Todos los mamíferos tienen una raza y eso se refleja en la clase. Menos el método **Mostrarse()** que es llamado por las otras clases, en **Mamifero** los demás están vacíos.

Por su parte la clase **Perro** hereda de la case-madre todo, pero además posee un nombre. Se da el **geter** del nombre y se sobrepasan todos los métodos virtuales que le atañen. El cualificador **virtual** no hace falta, pero está ahí por claridad del código. Lo mismo para la clase **Gato**, sólo que esta clase tiene un color en vez de un nombre.

```
#include "mamifero.h"
#include <iostream>
using std::cout;

// los mamíferos
Mamifero::Mamifero( std::string rz ) : raza( rz ) {}

// los perros
Perro::Perro( std::string rz, std::string nb ) : Mamifero( rz ), nombre( nb ) {}
void Perro::Hablar () const { cout << "\255Guau, guau!\n"; }
void Perro::Moverse( int n ) const { cout << "El perro da " << n << " pasos.\n"; }
void Perro::Mostrarse() const {
    cout  << "El perro se llama " << getNombre()
          << " y es un " << Mamifero::getRaza() << '\n';
}

// los gatos
Gato::Gato( std::string rz, std::string cl ) : Mamifero( rz ), color( cl ) {}
void Gato::Hablar () const { cout << "\255Miau!\n"; }
void Gato::Moverse( int n ) const { cout << "El gato trepa " << n << " zancadas.\n"; }
void Gato::Mostrarse() const {
    cout  << "El gato es de color " << getColor()
          << " y " << Mamifero::getRaza() << '\n';
}
```

Uso

El programa declara dos objetos diferentes con solamente un puntero a la clase-madre. Cuando cambia de un objeto a otro se pone a nulo para evitar fugas de memoria. Cuando el programa termina, se devuelve la memoria. ¡La magia del polimorfismo en todo su esplendor!

```
#include "mamifero.h"
#include <iostream>
#include <cstdlib>

int main() {
    std::cout << "Programa 6: polimorfismo.\n\n";
    Mamifero *pPtr; // un sólo puntero a la clase-madre

    pPtr = new Perro( "Doberman", "Bobby" );
    pPtr->Mostrarse();
    pPtr->Hablar();
    pPtr->Moverse( 8 );
    std::cout << '\n';
    pPtr = nullptr; // evita las fugas de memoria

    pPtr = new Gato( "siam\202s", "barcino" );
    pPtr->Mostrarse();
    pPtr->Moverse( 2 );
    pPtr->Hablar();
    std::cout << '\n';
    pPtr = nullptr;

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

Salida

```
Programa 6: polimorfismo.

El perro se llama Bobby y es un Doberman
¡Guau, guau!
El perro da 8 pasos.

El gato es de color barcino y siamés
El gato trepa 2 zancadas.
¡Miau!
```

La eficiencia es de 27%

Aplicación del polimorfismo

Ya se vio que el polimorfismo es una parte integral de la *filosofía* de la OOP, pero la *técnica* de aplicación en C++ sigue los cuatro pasos seguidamente mostrados, que son de carácter obligatorio:

1. Tiene que haber herencia: *el polimorfismo se hace con clases derivadas de una jerarquía*.
2. Tiene que haber clases virtuales: *se consigue haciendo virtual uno o más métodos*.

3. Tiene que manejar los objetos por punteros o referencias: *el polimorfismo se pierde con el pase por valor*.
4. Tiene que definir *como virtual* en la clase-madre todo lo que una clase-hija va a hacer, aunque el cuerpo este vacío.

Costos asociados

El uso del polimorfismo conlleva a un costo espacio-temporal, ese "algo" del cual ya se habló. Para que el programa "sepa" qué hay que invocar, se hace una tabla virtual (V-TABLE en inglés) para cada clase virtual y eso engrosa al código ejecutable. Para que el programa invoque correctamente a una función, cada V-TABLE tiene un puntero adscrito (V-POINTER en inglés) y cuando el programa lo usa, añade tiempo a la ejecución. Es verdad que con los compiladores modernos los costos son nimios, pero a pesar de todo, la regla de oro es:

Si no se vislumbra en el diseño de una clase (o en su desarrollo consecuente) que derive otras clases, no hay razón alguna para que ella misma tenga métodos virtuales.

Los errores lógicos (*bugs* en inglés) provocados por el polimorfismo, las V-TABLE y los V-POINTERS están entre los más difíciles de encontrar y erradicar por el programador novel…y no tan novel.

Siempre que se pueda, desarrollar las clases por agregación antes que por herencia

Una advertencia

La herencia no debe ser una técnica para implementación de trucos o atajos. Debe ser usada para captar las relaciones lógicas entre los objetos presentes en la jerarquía a la cual se aplica. Para ahorrar la escritura de código hay otras técnicas.

Existen usos indebidos de la herencia y su mecanismo de polimorfismo cuando se aplican a una jerarquía heterogénea con el ánimo de ahorrar código escrito, porque debilitan el chequeo fuerte de tipos sobre los objetos contenidos en la familia. Debe entenderse que este juicio aplica en particular a C++ que fue diseñado para favorecer el chequeo estático de tipos y a la vez, minimizar el tiempo de ejecución. Los buenos diseños hechos en C++ se fundamentan en esa virtud.

Herencia múltiple

En C++, una clase se puede derivar de más de una clase base: una técnica conocida como herencia múltiple, en la que una clase derivada hereda los miembros de dos o más clases base. La herencia múltiple es un concepto difícil que sólo deben utilizar los programadores experimentados. De hecho, los lenguajes de programación más recientes como Java, PHP y C#, no la permiten. Editado de (Deitel & Deitel, pág. 1039)

Dicho esto, he aquí un ejemplo que la pide: la mitología griega está llena de ejemplos de animales híbridos: las Sirenas, las Arpías, los Centauros, las Quimeras, los Pegasos y algunos más…Tomemos pues un Pegaso, un híbrido entre caballo y pájaro.

Programa 7-la herencia múltiple

Dada una clase acreditada como **`Pajaro`** y otra conocida por **`Caballo`**, obtenemos un **`Pegaso`** por herencia múltiple, separando por comas las clases-madre en la designación de la clase-hija.

```
class Caballo  { . . . } ;
class Pajaro { . . . };
```

```
class Pegaso  : public Caballo, public Pajaro { . . . };
```

Ilustración 8. Partes de un objeto Pegaso

Cuando un objeto Pegaso es creado en memoria, ambas clases forman parte de él, como ilustran la figura y sus colores. Si Pegaso deriva de ambas clases y cada una de ellas tiene constructores que toman parámetros, la clase Pegaso deberá inicializar estos constructores por turno, como se muestra en el ejemplo.

Declaraciones

Un caballo es medido por manos y relincha; un pájaro vuela y ambos gozan de un color, aunque muy distinto. Un Pegaso posee características mezcladas: es medido por manos, relincha como un caballo y posee un color de caballo, pero además puede volar como un pájaro. Aquí ocurre un problema curioso: cuando se le pregunta al Pegaso por su color ¿cuál retorna? ¿el color del pájaro o el del caballo? Evidentemente, hay una ambigüedad. La ambigüedad se puede quitar diciendo que se desea el color del caballo.

```
#ifndef ANIMAL_H_INCLUDED
#define ANIMAL_H_INCLUDED

#include <iostream>
typedef int MANOS;
enum COLOR { Rojo, Verde, Azul, Amarillo, Blanco, Negro, Bayo, Pinto } ;

class Caballo {
public:
    Caballo( COLOR elColor, MANOS elAlto );
    virtual ~Caballo() { std::cout << "Destructor de un Caballo...\n"; }
    virtual void Relincha()  const;
    virtual MANOS GetAlto()  const { return alto; }
    virtual COLOR getColor() const { return color; }
private:
    COLOR color;
    MANOS alto;
};

class Pajaro {
public:
    Pajaro( COLOR elColor, bool migra );
    virtual ~Pajaro() { std::cout << "Destructor de un P\240jaro...\n"; }
    virtual void Gorjea() const;
    virtual void Vuela()  const { std::cout << "\255Puedo volar!"; }
    virtual COLOR getColor() const { return color; }
    virtual bool  getMigracion() const { return migracion; }
private:
    COLOR color;
    bool  migracion;
};
```

```
class Pegaso : public Caballo, public Pajaro { // <-- primero el caballo
public:
    void Gorjea() const { Relincha(); }
    Pegaso( COLOR elColor, MANOS elAlto, bool migra, short laEdad, short creen );
    virtual ~Pegaso() { std::cout << "Destructor de un Pegaso...\n"; }
    virtual COLOR getColor() const { return Caballo::getColor(); } // <-- quita la ambiguedad
    short getEdad() const { return edad; }
    short getCreyentes() const { return creyentes; }
private:
    short edad;
    short creyentes;
};

#endif // ANIMAL_H_INCLUDED
```

Definiciones

Aquí se muestran los constructores y cómo se cargan. Si en la clase compuesta primero se puso el caballo, al llenar su constructor primero se pone al del caballo. Si ese orden no se respeta aparece una advertencia:

```
||=== Build: Release in prob07 (compiler: GNU GCC Compiler) ===
In constructor 'Pegaso::Pegaso(COLOR, MANOS, bool, short int, short int)':
warning: 'Pegaso::creyentes' will be initialized after [-Wreorder]
warning: base 'Caballo' [-Wreorder]
warning: when initialized here [-Wreorder]
||=== Build finished: 0 error(s), 3 warning(s) (0 minute(s), 3 second(s)) ===
```

```
#include "animal.h"
#include <iostream>
using std::cout;

Caballo::Caballo( COLOR elColor, MANOS elAlto ):
    color( elColor ), alto( elAlto ) {
    cout << "Constructor de un Caballo...\n";
}

void Caballo::Relincha() const { std::cout << "\255Whijijiji!..."; }

Pajaro::Pajaro( COLOR elColor, bool migra ):
    color( elColor ), migracion( migra ) {
    cout << "Constructor de un P\240jaro...\n";
}

void Pajaro::Gorjea() const { std::cout << "Chirp..."; }

Pegaso::Pegaso (
    COLOR unColor,
    MANOS unAlto,
    bool migra,
    short laEdad,
    short creen ):
        Caballo( unColor, unAlto ), // <-- primero el caballo
        Pajaro( unColor, migra ),
        edad( laEdad ),
        creyentes( creen ) {
    cout << "Constructor de un Pegaso...\n";
}
```

Uso

```
#include"animal.h"
#include <iostream>
#include <cstdlib>

int main() {
    using std::cout;
    cout << "Programa 7: herencia m\2431tiple.\n\n";
    Pegaso * pPeg = new Pegaso( Bayo, 4, true, 2, 10 );
    cout << '\n';

    pPeg->Vuela();    cout << '\n';
    pPeg->Relincha(); cout << '\n';
    cout << "Su Pegaso es un ";

    if ( pPeg->getColor() == Bayo ) cout << "bayo ";

    cout << "que tiene " << pPeg->GetAlto() << " manos de altura,\n"
         << pPeg->getEdad() << " a\244os de edad y ";

    pPeg->getMigracion() ? cout << "migra. " : cout << "no migra. ";

    cout << "Un total de " << pPeg->getCreyentes() << " persona(s)\n"
           "cree(n) que existe.\n\n";

    delete pPeg;
    return EXIT_SUCCESS;
}
```

Salida

```
Programa 7: herencia múltiple.

Constructor de un Caballo...
Constructor de un Pájaro...
Constructor de un Pegaso...

¡Puedo volar!
¡Whijijiji!...
Su Pegaso es un bayo que tiene 4 manos de altura,
2 años de edad y migra. Un total de 10 persona(s)
cree(n) que existe.

Destructor de un Pegaso...
Destructor de un Pájaro...
Destructor de un Caballo...
```

La eficiencia es del 27%.

La herencia múltiple incurre en dificultades técnicas muchas veces innecesarias, por eso se advierte que se prefiera usar la simple, si se puede.

Ambigüedades: mismo método

En el problema anterior ambas clases tienen el mismo método (**`getColor`**) y la misma firma, por lo que el Pegaso debe resolver cuál color enviar cuando se le pregunta. Esto crea una ambigüedad que se soluciona con una llamada explícita a

la clase base: **`pPeg->Caballo::getColor()`**. Entonces, si el Pegaso va a controlar su color, se le puede mover el problema, encadenándolo a la función deseada: **`virtual COLOR getColor() const { return Caballo::getColor(); }`** aunque desde luego, un cliente puede violar el problema simplemente escribiendo: **`pPeg->Pajaro::GetColor()`**.

A la acción de subir un problema pasándolo a una clase superior en la jerarquía se le llama “percolarlo”

Ambigüedades: clase en común

Para resolver esta ambigüedad suponer ahora que ambas clases, Caballo y Pájaro, derivan de una clase común: Animal.

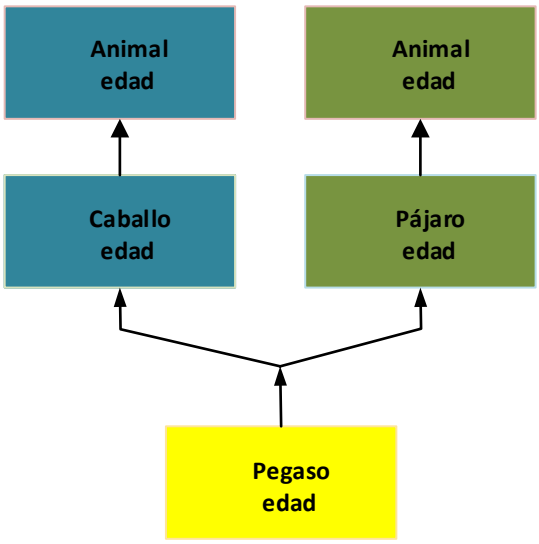


Ilustración 9. Una clase común

Entonces existen dos objetos —como se muestra en la figura— y existe otra clase de ambigüedad; si esa clase-madre tiene un método virtual (dígase **`getEdad()`**) y es llamado por el Pegaso, a cuál llama ¿al que hereda de su clase Caballo, o al que hereda de su clase Pájaro?

La herencia en diamante

Es posible decirle a C++ que se desea una sola copia de la case Animal, como se muestra abajo en la figura. Eso se consigue haciendo Animal una clase-madre virtual de ambos: el caballo y el pájaro. Como es clase común, es lógico que lleve la edad y el color de sus descendientes, y sea virtual pura. Ahora todas sus hijas cambian.

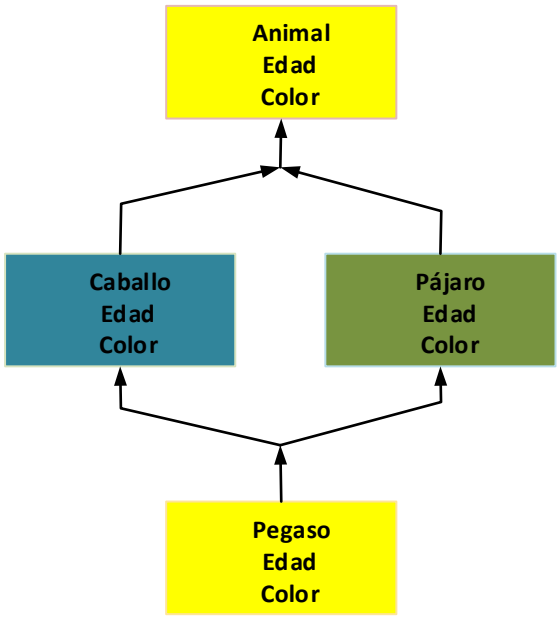


Ilustración 10. Una clase común

Hay que observar bien el ejemplo, sobre todo como se pasan ahora los parámetros comunes, y contrastarlo con el anterior.

Programa 8-la clase en común

Normalmente el constructor de la clase inicializa sus variables y las de sus clases-madre, pero en el caso de la herencia virtual no es así. El constructor se inicializa por su clase común y eso es *la herencia en diamante*. Pero de toda suerte, las clases intermedias todavía han de inicializar sus parámetros, como se muestra en el ejemplo.

Definiciones

```
#ifndef ANIMAL_H
#define ANIMAL_H

#include <iostream>
typedef int MANOS;
enum COLOR { Rojo, Verde, Azul, Amarillo, Blanco, Negro, Bayo, Pinto };

class Animal { // base común, es virtual y va primero
public:
    Animal( COLOR elColor, short laEdad );
    virtual ~Animal() {  std::cout << "Destructor de un Animal...\n"; }
    virtual int getColor()  const = 0;
    virtual short getEdad() const { return edad; }
private:
    COLOR color;
    short edad;
};

class Caballo : virtual public Animal {
public:
    Caballo( COLOR elColor, short laEdad, MANOS elAlto );
    virtual ~Caballo() { std::cout << "Destructor de un Caballo...\n"; }
    virtual void Relincha()  const { std::cout << "\255Whijiji!\n"; }
```

```
    virtual MANOS getAlto()  const { return altura; }
private:
    MANOS altura;
};

class Pajaro : virtual public Animal {
public:
    Pajaro( COLOR elColor, short laEdad, bool migra );
    virtual ~Pajaro() { std::cout << "Destructor de un P\240jaro...\n"; }
    virtual void Vuela() const { std::cout << "\255Puedo volar!\n"; }
    virtual bool getMigracion() const { return migracion; }
private:
    bool migracion;
};

class Pegaso : public Caballo, public Pajaro {
public:
    Pegaso( COLOR elColor, short laEdad, MANOS elAlto, bool migra, short creen );
    virtual ~Pegaso() { std::cout << "Destructor de un Pegaso...\n"; }
    void setEdad( int laEdad ) {}
    void Vuela() const {}
    void Relincha() {};
    int getColor() const { return Animal::getColor(); }
    short getCreyentes() const { return creyentes; }
private:
    short creyentes;
};

#endif // ANIMAL_H
```

Declaraciones

```
#include "animal.h"
using std::cout;

Animal::Animal( COLOR elColor, short laEdad ):
    color( elColor ), edad( laEdad ) { cout << "Constructor de un Animal...\n"; }

int Animal::getColor() const { return color; } ;

Caballo::Caballo( COLOR elColor, short laEdad, MANOS elAlto ):
    Animal( elColor, laEdad ), altura( elAlto ) { cout << "Constructor de un Caballo...\n"; }

Pajaro::Pajaro( COLOR elColor, short laEdad, bool migra ):
    Animal( elColor, laEdad ), migracion( migra ) { cout << "Constructor de un P\240jaro...\n"; }

Pegaso::Pegaso (
    COLOR elColor,
    short laEdad,
    MANOS elAlto,
    bool migra,
    short creen ):
        Animal( elColor, laEdad ),
        Caballo( elColor, laEdad, elAlto ),
        Pajaro( elColor, laEdad, migra ),
        creyentes( creen )
    { cout << "Constructor de un Pegaso...\n"; }
```

Uso

```
#include "animal.h"
#include <iostream>
#include <cstdlib>

int main() {
    using std::cout;
    cout << "Programa 8: herencia en diamante.\n\n";
    Pegaso * pPeg = new Pegaso( Bayo, 2, 4, true, 10 );

    cout << '\n';
    pPeg->Vuela();
    pPeg->Relincha();
    cout << '\n';
    cout << "Su Pegaso es un ";
    if ( pPeg->getColor() == Bayo ) cout << "bayo ";
    cout << "que tiene " << pPeg->getAlto() << " manos de altura,\n"
         << pPeg->getEdad() << " a\244os de edad y ";
    pPeg->getMigracion() ? cout << "migra. " : cout << "no migra. ";
    cout << "Un total de " << pPeg->getCreyentes() << " persona(s)\n"
           "cree(n) que existe.\n\n";

    delete pPeg;
    return EXIT_SUCCESS;
}
```

Salida

```
Programa 8: herencia en diamante.

Constructor de un Animal...
Constructor de un Caballo...
Constructor de un Pájaro...
Constructor de un Pegaso...

Su Pegaso es un bayo que tiene 4 manos de altura,
2 años de edad y migra. Un total de 10 persona(s)
cree(n) que existe.

Destructor de un Pegaso...
Destructor de un Pájaro...
Destructor de un Caballo...
Destructor de un Animal...
```

La eficiencia es del 26%.

La salida es la misma del programa anterior, pero ahora sólo hay una clase común, y eso se refleja con la ganancia de claridad (la pérdida de ambigüedades) en el código.

Aunque en ciertos y determinados casos la herencia múltiple ofrece ventajas sobre la simple, a esta fecha la comunidad de C++ duda de su uso. Aunque la gran mayoría de los compiladores en el mercado la implementan, no todos lo hacen bien; es más difícil de depurar el programa que la emplea; y casi todo lo que necesite herencia múltiple se puede hacer con unos cuantos trucos aplicados a la simple.

Las razones anteriores son válidas. Instalar complejidad innecesaria en los programas nunca es bueno. La herencia múltiple se usa cuando se requiere de atributos y comportamientos de más de una clase y considerar no usarla si no se puede derivar en diamante, en la misma (o parecida) forma en que la da el texto.

Siempre preferir la herencia simple sobre la múltiple

Programa 9-polimorfismo y herencia múltiple

Es posible hacer polimorfismo con la herencia múltiple, aunque el problema se complica porque hay que duplicar las acciones de las clases-hijas en la clase-madre. Por lo demás el mecanismo es idéntico al ya conocido.

Declaración

```
#ifndef ANIMAL_H
#define ANIMAL_H

#include <iostream>
typedef int MANOS;
enum COLOR { Rojo, Verde, Azul, Amarillo, Blanco, Negro, Bayo, Pinto };

class Animal { // base común, es virtual pura y va primero
public:
    Animal( COLOR elColor, short laEdad );
    virtual ~Animal() {  std::cout << "Destructor de un Animal...\n"; }
    virtual int getColor()  const { return color; }
    virtual short getEdad() const { return edad; }
    // todos los comportamientos de las hijas deben ser declarados aquí
    virtual void Relincha() const = 0;
    virtual MANOS getAlto() const { return 1;}
    virtual void Vuela() const {}
    virtual bool getMigracion()  const { return false; }
    virtual short getCreyentes() const { return 1; }
private:
    COLOR color;
    short edad;
};

class Pajaro : virtual public Animal {
public:
    Pajaro( COLOR elColor, short laEdad, bool migra );
    virtual ~Pajaro() { std::cout << "Destructor de un P\240jaro...\n"; }
    virtual void Vuela() const { std::cout << "\255Puedo volar!\n"; }
    virtual bool getMigracion() const { return migracion; }
private:
    bool migracion;
};

class Caballo : virtual public Animal {
public:
    Caballo( COLOR elColor, short laEdad, MANOS elAlto );
    virtual ~Caballo() { std::cout << "Destructor de un Caballo...\n"; }
    virtual void Relincha() const { std::cout << "\255Whijiji!\n"; }
    virtual MANOS getAlto() const { return altura; }
private:
    MANOS altura;
};
```

```
class Pegaso : public Caballo, public Pajaro {
public:
    Pegaso( COLOR elColor, short laEdad, MANOS elAlto, bool migra, short creen );
    virtual ~Pegaso() { std::cout << "Destructor de un Pegaso...\n"; }
    void setEdad( int laEdad ) {}
    void Vuela()     const { Pajaro::Vuela(); }
    void Relincha() const { Caballo::Relincha(); };
    int getColor()  const { return Animal::getColor(); }
    short getCreyentes() const { return creyentes; }
private:
    short creyentes;
};

#endif // ANIMAL_H
```

Definición

```
#include "animal.h"
using std::cout;

Animal::Animal( COLOR elColor, short laEdad ):
    color( elColor ), edad( laEdad ) { cout << "Constructor de un Animal...\n"; }

Caballo::Caballo( COLOR elColor, short laEdad, MANOS elAlto ):
    Animal( elColor, laEdad ), altura( elAlto ) { cout << "Constructor de un Caballo...\n"; }

Pajaro::Pajaro( COLOR elColor, short laEdad, bool migra ):
    Animal( elColor, laEdad ), migracion( migra ) { cout << "Constructor de un P\240jaro...\n"; }

Pegaso::Pegaso (
    COLOR elColor,
    short laEdad,
    MANOS elAlto,
    bool migra,
    short creen ):
        Animal( elColor, laEdad ),
        Caballo( elColor, laEdad, elAlto ),
        Pajaro( elColor, laEdad, migra ),
        creyentes( creen )
    { cout << "Constructor de un Pegaso...\n"; }
```

Uso

```
#include "animal.h"
#include <iostream>
#include <cstdlib>

int main() {
    using std::cout;
    cout << "Programa 13: herencia m\243ltiple y polimorfismo.\n\n";
    Animal * pAnimal = new Pegaso( Bayo, 2, 4, true, 10 );

    cout << '\n';
    pAnimal->Vuela();
    pAnimal->Relincha();
    cout << "Su Pegaso es un ";
    if ( pAnimal->getColor() == Bayo ) cout << "bayo ";
    cout << "que tiene " << pAnimal->getAlto() << " manos de altura,\n"
```

```
        << pAnimal->getEdad() << " a\244os de edad y ";
    pAnimal->getMigracion() ? cout << "migra. " : cout << "no migra. ";
    cout << "Un total de " << pAnimal->getCreyentes() << " persona(s)\n"
           "cree(n) que existe.\n\n";

    delete pAnimal;
    return EXIT_SUCCESS;
}
```

Salida

```
Programa 9: herencia múltiple y polimorfismo.

Constructor de un Animal...
Constructor de un Caballo...
Constructor de un Pájaro...
Constructor de un Pegaso...

¡Puedo volar!
¡Whijiji!
Su Pegaso es un bayo que tiene 4 manos de altura,
2 años de edad y migra. Un total de 10 persona(s)
cree(n) que existe.

Destructor de un Pegaso...
Destructor de un Pájaro...
Destructor de un Caballo...
Destructor de un Animal...
```

La eficiencia es del 24%

Si puede resolver con agregación: ¡hágalo! Si tiene que usar la herencia y puede resolver con la simple: ¡hágalo!

Programa 10-una BD polimórfica

Según (Lipschutz, pág. 131), una lista enlazada la constituye una colección lineal de elementos llamados nodos, donde el orden de los mismos se establece mediante punteros. También se conoce que tomar una estructura adecuada de datos es importante: bien escogida facilita enormemente la tarea.

Las soluciones dadas anteriormente son una LSE manualmente administrada (Introducción al C++, tomo 1, págs. 167-175) y una LSE administrada con la ayuda de la STL (Introducción al C++, tomo 2, págs. 68-72)

Ahora se hará una lista que opera polimórficamente.

La estructura de los datos es compleja; lleva un vector que se encarga de manejar la lista, pero a su vez, cada nodo es un puntero polimórfico que lleva el número común de la pieza y termina en una de dos: el año del modelo de un auto o el número de motores de un avión. La base de datos lo controla todo.

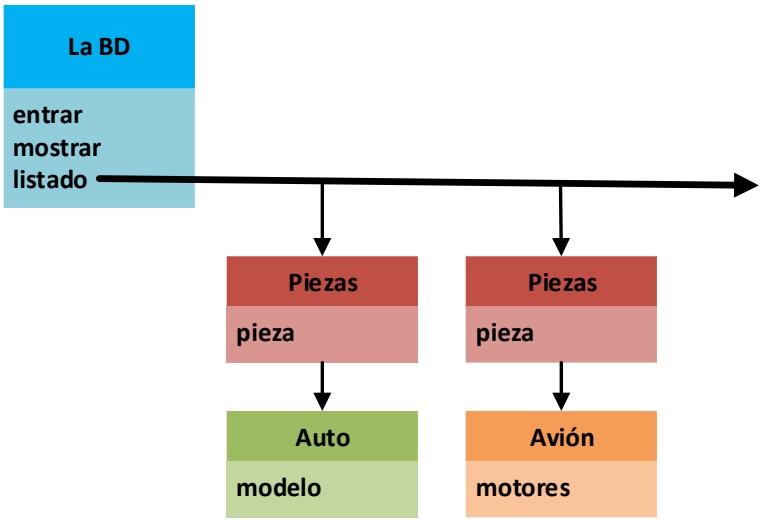


Ilustración 11. La base de datos

Tiene funcionalidad mínima pues lo que se desea es mostrar cómo se hace, pero lleva algo que se ha aprendido y no es nada fácil: cómo manejar la base de datos polimórfica, un listado controlado por la LST y un ordenamiento débil, ya que las piezas son únicas.

Declaraciones

- La parte polimórfica es trivial: un par de clases derivan de ella.

```
#ifndef PIEZAS_H_INCLUDED
#define PIEZAS_H_INCLUDED

typedef unsigned long  uLong;
typedef unsigned short uShort;

// Piezas: base-madre abstracta
class Piezas {
public:
    Piezas(): pieza( 1 ) {}
    Piezas( uLong laPieza ): pieza( laPieza ) {}
    virtual ~Piezas() {}
    virtual uLong getPieza() const { return pieza; }
    virtual void muestra() const = 0; // debe ser sobrescrito
private:
    uLong pieza;
};

// Piezas de un auto
class Auto : public Piezas {
public:
    Auto(): modelo( 2020 ) {}
    virtual ~Auto() {}
    Auto( uLong laPieza, uShort elModelo );
    virtual void muestra() const;
private:
    uShort modelo;
};

// Piezas de un avión
```

```
    class Avion : public Piezas {
    public:
        Avion(): motores( 1 ) {};
        virtual ~Avion() {}
        Avion( uLong laPieza, uShort losMotores );
        virtual void muestra() const;
    private:
        uShort motores;
    };

    #endif // PIEZAS_H_INCLUDED
```

- La parte del listado lo incluye todo, incluso la función de ordenamiento débil. El manejo del vector se deja a la STL

```
#ifndef BASE_H_INCLUDED
#define BASE_H_INCLUDED

#include "piezas.h"
#include <iostream>
#include <algorithm>
#include <vector>
typedef std::vector<Piezas *> Vct;

// función de ordenamiento fuerte
bool menor( const Piezas * n1, const Piezas * n2 ) {
    return n1->getPieza() < n2->getPieza();
}

class Base {
public:
    Base() {};
    virtual ~Base() {};
    void mostrar();
    void ordenar() { sort( vct.begin(), vct.end(), menor ); };
    void insertar( Piezas * pParte ) { vct.push_back( pParte ); }
private:
    Vct vct;
};

void Base::mostrar() {
    for ( auto x: vct ) {
        x->muestra();
    } // for-en-rangos
}

#endif // BASE_H_INCLUDED
```

Definición

- La parte polimórfica

```
#include "piezas.h"
#include <iostream>
using std::cout;

// implementación de una función virtual-pura para
// que pueda encadenarse en las clases-hijas
void Piezas::muestra() const { cout << "Pieza No.: " << pieza; }

// Piezas de auto
Auto::Auto( uLong laPieza, uShort elModelo ) :
```

```
    Piezas( laPieza ), modelo( elModelo ) {}

void Auto::muestra() const {
    Piezas::muestra();
    cout << "\tModelo del a\244o " << modelo << '\n';
}

// piezas de avión
Avion::Avion( uLong laPieza, uShort losMotores ):
    Piezas( laPieza ),
    motores( losMotores ) {}

void Avion::muestra() const {
    Piezas::muestra();
    cout << "\tN\243m. motores = " << motores << '\n';
}
```

Uso

Es importante que la base se ponga en la tienda gratis, si no ocurre un error muy desagradable: el programa compila sin problemas, pero al ejecutarse…se cuelga.

```
#include "base.h"
#include <cstdlib>

int main() {
    using std::cout;
    using std::cin;
    cout << "Programa 10: una BD polim\242rfica.\n\n";
    Base * pBase = new Base;    // debe radicar en la tienda gratis
    Piezas * pParte = nullptr; // un puntero para todos
    uLong laPieza;
    uShort elValor, opcion;

    while ( true ) {
        cout << "(1)Carro; (2)Avi\242n; (0)Fin: ";
        cin >> opcion;

        if ( 0 == opcion )
            break;

        cout << "Nueva pieza: ";
        cin >> laPieza;

        if ( opcion == 1 ) { // <-- aquí discrimina
            cout << "A\244o: ";
            cin >> elValor;
            pParte = new Auto( laPieza, elValor );
        } else {
            cout << "No. de motores: ";
            cin >> elValor;
            pParte = new Avion( laPieza, elValor );
        } // if-else

        pBase->insertar( pParte );
    } // while

    cout << '\n';
    pBase->ordenar();
    pBase->mostrar();
```

```
    cout << '\n';

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

Salida

```
Programa 10: una BD polimórfica.

(1)Carro; (2)Avión; (0)Fin: 1
Nueva pieza: 123
Año: 2019
(1)Carro; (2)Avión; (0)Fin: 2
Nueva pieza: 320
No. de motores: 2
(1)Carro; (2)Avión; (0)Fin: 2
Nueva pieza: 325
No. de motores: 4
(1)Carro; (2)Avión; (0)Fin: 1
Nueva pieza: 118
Año: 2017
(1)Carro; (2)Avión; (0)Fin: 0

Pieza No.: 118  Modelo del año 2017
Pieza No.: 123  Modelo del año 2019
Pieza No.: 320  Núm. motores = 2
Pieza No.: 325  Núm. motores = 4
```

La eficiencia es del 27%

11-PROGRAMACIÓN GENÉRICA

> *Sólo después de que C++ logró algún éxito, fue que Stroustrup adicionó las plantillas, permitiendo la programación genérica. Y sólo después de que las características de las plantillas habían sido usadas y refinadas, se hizo aparente que quizá fueran una adición tan significativa como la OOP. (Prata, p. 15)*

Una de las características de reutilización de software más potentes de C++ son las plantillas, que pueden ser de funciones y de clases, haciendo insensibles al tipo de dato o de contenedor a esas construcciones. A esta técnica se le conoce como programación genérica.

Los compiladores modernos de C++ en general —y el que aquí se usa, de la suite de GNU en particular— requieren que la definición completa de una plantilla aparezca en el archivo de código fuente que la utiliza. Por esta razón y por motivos del rehúso, las plantillas se definen en archivos de cabecera, que después se incluyen en los archivos de código-fuente apropiados. Esto significa que las funciones-miembro también están definidas en dicho archivo.

El estudio de la programación genérica por otro lado, eleva el nivel del texto también a medio

Plantillas de clases

La programación con plantillas es diferente a la OOP tradicional, la cual se centra alrededor del *polimorfismo de tipos*. En cambio, esta se centra en el *polimorfismo paramétrico*, en cuál una función o clase están definidas independientemente de sus parámetros.

Querido lector, la programación genérica es (casi) otro lenguaje de computación. Se parece al C++, pero no lo es; al menos, no totalmente. Es difícil de escribir y aún más de leer, pero EODA vale la pena probarle el gusto. La programación de plantillas de clases y la de los functores, elevan el nivel del texto a medio.

Ahora se verá (elementalmente) la programación genérica de plantillas de clases que pueden usarse en más de una ocasión, parametrizando sus interfaces y retornando los datos de la forma más simple posible. Al pasar el tipo de dato como parámetro, el compilador le produce un código específico y lo integra en el lugar dónde la llama el flujo del programa.

Una clase

Como la programación de plantillas no es trivial, se verá inicialmente una clase que se convertirá a plantilla, una clase MiniMax, muy simple, que toma un contenedor de enteros y produce un par de valores con el mínimo y el máximo.

Programa 11-una clase para plantilla

¿Qué contenedor usar? Los cuatro asociativos típicos se descartan porque ya son ordenados por definición, así pues, quedan los secuenciales. De ellos la doble-cola y sus especializaciones (la pila y la cola simple) quedan fuera porque sus operaciones son especiales. Entonces quedan el vector y la lista. Como el objetivo es programar una plantilla a partir de la clase de enteros, el autor tomó el vector y mantuvo la funcionalidad mínima, y la codificación en un bloque.

```
#include "../deposito/rnd.h"
#include <iostream>
#include <cstdlib>
#include <vector>
#include <iterator>
#include <algorithm>
#include <ctime>

class MiniMax {
public:
    MiniMax( std::vector<int> vct ) : v(vct), minInt(0), maxInt(0) {}
    std::pair<int, int> buscar();
private:
    std::vector<int> v;
    int minInt;
    int maxInt;
};

std::pair<int, int> MiniMax::buscar() {
    minInt = *min_element( v.begin(), v.end() );
    maxInt = *max_element( v.begin(), v.end() );
    return std::make_pair( minInt, maxInt );
}

const int N = 10, P = 20, Q = 80;
```

```
int main() {
    std::cout << "Problema 11: una clase MiniMax.\n\n";
    std::vector<int> vct( N );
    srand( time( nullptr ) );
    RELLENA_I( vct, P, Q );
    MiniMax mm( vct );
    std::pair<int, int> par = mm.buscar();
    std::copy( vct.begin(), vct.end(), std::ostream_iterator<int>( std::cout, ", " ) );
    std::cout << "\b\b \n";
    std::cout << "M\241n = " << par.first  << '\n'
              << "M\240x = " << par.second << "\n\n";
    system( "pause" );
    return EXIT_SUCCESS;
}
```

Salida

```
Problema 11: una clase MiniMax.

42, 70, 65, 76, 79, 58, 80, 29, 79, 64
Mín = 29
Máx = 80
```

La eficiencia es del 37%.

El problema de esta clase es que sólo sirve con vectores de enteros.

Una plantilla de clase

Una plantilla de clases toma uno o más parámetros y cuando se va a instanciar, el compilador le suministra valores para ellos. Centrada en el polimorfismo paramétrico, es definida independientemente de sus parámetros, que pueden ser valores, tipos o incluso, otras plantillas, pero que no se conocen al momento de la compilación.

Recordar que:

- La plantilla no se define: se implementa. Por eso es considerada unidad atómica de código en C++.
- En tiempo de codificación no se conoce su forma definitiva: el compilador sustituye la plantilla por una versión adaptada al entorno donde se llama y la inserta *in situ*, haciendo enlace temprano.
- La programación genérica de clases no es nada trivial y tradicionalmente se considera difícil de escribir o interpretar: es prácticamente un lenguaje *per se*.

El vector es un contenedor secuencial junto con la lista y la doble-cola. Todos son de tipo genérico y los métodos cargar y buscar no cambian excepto por su generalidad. Habrá que hacer cambios menores —tal vez quitarle unos **`#include`** o añadirle otros— y afinar el comportamiento. Luego un pequeño **`driver`** la probará con un vector de **`double`**s, una lista de **`int`**s y aún otro de doble-colas. Se ilustra cómo separar las partes del código dentro del fichero cabecera.

Programa 12-la plantilla de la clase

Se procede a transformar la clase **`MiniMax`** en una plantilla. Se insensibilizan el tipo de datos y el contenedor, que ahora puede ser cualquiera que sea secuencial. Su formato de escritura es:

`template class<lista_de_Tipos> tipo_de_retorno NOMBRE(lista_de_parámetros) {...}`

Definición y declaración

Todo va junto ya que una plantilla es unidad atómica de código C++. Lo primero es el encabezamiento que es muy parecido a la declaración de funciones (vista en el tomo anterior), pero sin la palabra-clave **`inline`**: es un error ponerla porque sólo la pueden llevar las funciones y esta es una clase.

Luego hay que localizar el tipo y cambiarlo por uno genérico. Donde quiera que esté la palabra-clave **`int`**, poner **`T`**, previamente declarada cono **`typename T`** en el encabezamiento. La costumbre es poner los nombres de los miembros con una adenda (ahora se llaman **`min_`** y **`max_`**). Por último, hay que sobrecargarla para que trabaje con los tres contenedores secuenciales, porque las colas sólo trabajan por las puntas y tanto ellas como las listas no pueden ser accedidas aleatoriamente. No obstante, lo que se pide sólo puede hacerlo un vector, por lo que se la asignan los valores de los otros contenedores y él hace todo el trabajo. Se ilustra cómo se separan las partes de la plantilla en la misma cabecera.

```
#ifndef PLANTILLA_H_INCLUDED
#define PLANTILLA_H_INCLUDED

#include <vector>
#include <list>
#include <deque>
#include <algorithm>

template <typename T> class MiniMax {
public:
    MiniMax( std::vector<T> &vct ) { v.assign( vct.begin(), vct.end() ); }
    MiniMax( std::list<T>   &lst ) { v.assign( lst.begin(), lst.end() ); }
    MiniMax( std::deque<T>  &dek ) { v.assign( dek.begin(), dek.end() ); }
    std::pair<T, T> buscar();
private:
    std::vector<T> v;
    T min_;
    T max_;
};

template<typename T> std::pair<T, T> MiniMax<T>::buscar() {
    min_ = *min_element( v.begin(), v.end() );
    max_ = *max_element( v.begin(), v.end() );
    return std::make_pair( min_, max_ );
}

#endif // PLANTILLA_H_INCLUDED
```

Uso

```
#include "../deposito/rnd.h"
#include "../deposito/print.h"
#include "plantilla.h"
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <iomanip>
#include <iterator>

const int N = 10, P = 1, Q = 50;

int main() {
    using std::cout;
    cout << "Problema 12: una plantilla de clases.";
    srand( time( nullptr) );
```

```
    cout << "\n\nCon una lista de 15 enteros:\n";
    std::list<int> lst( 1.5 * N );
    RELLENA_I( lst, P, Q );
    LISTA( lst );
    MiniMax<int> mi( lst );
    std::pair<int, int> pi = mi.buscar();
    cout << "\nM\241n = " << pi.first  << '\n'
         << "M\240x = "   << pi.second;

    cout << "\n\nCon una doble cola de 12 cortos:\n";
    std::deque<short> dek( 1.2 * N );
    RELLENA_I( dek, P, Q );
    LISTA( dek );
    MiniMax<short> ms( dek );
    std::pair<short, short> ps = ms.buscar();
    cout << "\nM\241n = " << ps.first  << '\n'
         << "M\240x = "   << ps.second;

    cout << "\n\nCon un vector de 6 reales:\n";
    std::vector<double> vct( 0.6 * N );
    RELLENA_R( vct, P, Q );
    LISTA( vct, "", 2 );
    MiniMax<double> md( vct );
    std::pair<double, double> pd = md.buscar();
    cout << std::fixed << std::setprecision( 2 )
         << "\nM\241n = " << pd.first  << '\n'
         << "M\240x = "   << pd.second << "\n\n";

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

Salida

```
Problema 12: una plantilla de clases.

Con una lista de 15 enteros:
13, 35, 3, 17, 34, 8, 3, 31, 6, 47, 31, 40, 10, 39, 34
Mín = 3
Máx = 47

Con una doble cola de 12 cortos:
17, 19, 28, 10, 24, 11, 8, 34, 23, 32, 26, 28
Mín = 8
Máx = 34

Con un vector de 6 reales:
20.99, 13.29, 26.24, 25.72, 12.07, 34.10
Mín = 12.07
Máx = 34.10
```

La eficiencia es de 22%.

Ahora la plantilla es insensible al contenedor, al tipo y al tamaño.

Este es el momento de volver al segundo tomo de la serie y revisar las plantillas. Si aún no las comprende, ahora sí que lo va a hacer.

Functores

Los argumentos funcionales para los algoritmos no tienen por qué ser funciones. Pueden ser objetos que se comportan como funciones. Tales objetos son llamados functores.

Los functores son un ejemplo del poder de la programación genérica y el concepto de abstracción pura. Puede decirse que cualquier cosa que se comporta como una función es una función. Entonces, si se define un objeto que se comporta como una función, puede ser utilizado como una función.

Un functor es simplemente una función que define al operador `()`

Functores predefinidos

La STL contiene varios functores predefinidos que usan bastante y que cubren operaciones fundamentales de la biblioteca estándar. Utilizándolos, en la mayoría de los casos no hay porque escribir uno nuevo.

Tabla 7. Functores predefinidos

Functores unarios	
`negate<T>()`	`-parámetro`
`logical_not<T>()`	`!parámetro` `not parámetro`
Functores binarios	
`plus<T>()`	`parámetro1 + parámetro2`
`minus<T>()`	`parámetro 1 - parámetro2`
`multiplies<T>()`	`parámetro1 * parámetro2`
`divides<T>()`	`parámetro1 / parámetro2`
`modulus<T>()`	`parámetro1 % parámetro2`
`equal_to<T>()`	`parámetro1 == parámetro2`
`not_equal_to<T>()`	`parámetro1 != parámetro2`
`less<T>()`	`parámetro1 < parámetro2`
`greater<T>()`	`parámetro1 > parámetro2`
`less_equal<T>()`	`parámetro1 <= parámetro2`
`greater_equal<T>()`	`parámetro1 >= parámetro2`
`logical_and<T>()`	`parámetro1 && parámetro2` `parámetro1 and parámetro2`
`logical_or<T>()`	`parámetro1 || parámetro2` `parámetro1 or parámetro2`

Notas:

- Para usar los functores predefinidos, se tiene que incluir el archivo de encabezamiento `<funcional>`
- `less<T>()` es el criterio predeterminado por omisión cada vez que son ordenados los objetos de un contenedor. Las operaciones predeterminadas de clasificación también producen un orden ascendente por omisión.

¿Para qué usarlos?

¿Para qué usar un functor? Si bien ellos complican el código, sin embargo, tienen cinco ventajas:

1. Son funciones “inteligentes”, ya que además de definir al operador `()` pueden tener otros miembros y otros métodos.
2. Tienen enlace temprano, que es más eficiente.
3. Tienen sus propios tipos, aunque tengan una sola firma: se pasa el comportamiento funcional como si fuera un parámetro ordinario.

4. Son al menos tan rápidos como las funciones ordinarias, pero casi siempre más.
5. Pueden ser sobrecargados para ampliarles su alcance.

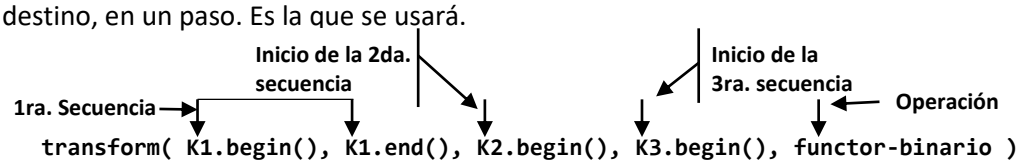
Programa 13-una clase functor

Ya se conocen los estadísticos descriptivos principales, pero por razones pedagógicas sólo se computarán la cantidad de elementos, su suma, promedio y varianza. Usando valores ya conocidos, calcularlos con una clase functor de enteros.

Declaración

La clase usara la STL como apoyatura. Se cambia el cálculo de la varianza a $s^2 = \frac{\sum x^2 - \frac{(\sum x)^2}{n}}{n-1}$ donde $\sum x^2$ es la suma de los cuadrados o **SDC(),** expresión que es idéntica a la dada anteriormente, pero que —aunque luce imponente— realmente cambia los ciclos de la otra por una expresión lineal que permite la transformación de la expresión. Se aprovecha el momento para volver a ilustrar el algoritmo **transform()** calculando la suma de los cuadrados de una sola vez.

El algoritmo **transform** tiene dos interfaces: la primera lleva tres elementos y un functor unario; y transforma los elementos de una fuente a un destino en un paso, por lo que sólo puede negar la expresión, ya sea aritmética, ya sea lógicamente. La segunda espera cuatro argumentos y un functor binario y combina los elementos de dos secuencias a un destino, en un paso. Es la que se usará.



Es de observar que la cuenta de los valores siempre es un número entero y el promedio, la mediana y la varianza siempre son números reales pues son (o pueden ser) producto de una división. El resto del cálculo depende del tipo de dato.

```
#ifndef ESTADS_H_INCLUDED
#define ESTADS_H_INCLUDED

#include <vector>
#include <algorithm>

class Estads {
public:
    Estads();
    Estads( std::vector<int> & vct );
    // al sobrecargar el operador la clase se convierte en un functor
    void operator()( int x ) {}
    int cuent()    const;
    int suma()     const;
    int SDC()      const;
    double media() const;
    double medin() const;
    double varza() const;
private:
    int n;
    std::vector<int> v;
};

#endif // ESTADS_H_INCLUDED
```

Definiciones

```
#include "estads.h"
Estads::Estads() : n( 0 ) { v.clear(); }
Estads::Estads( std::vector<int> & vct ) : v( vct ) { n = v.size(); }
int Estads::cuent() const { return n; }
int Estads::suma() const { return accumulate( v.begin(), v.end(), 0.0 ); }
double Estads::media() const { return static_cast<double>( suma() ) / n; }

double Estads::varza() const {
    return static_cast<double>( SDC() - suma() * suma() / n ) / ( n - 1 );
}

int Estads::SDC() const {
    std::vector<int>aux( v );
    transform( aux.begin(), aux.end(), aux.begin(), aux.begin(), std::multiplies<int>() );
    return accumulate( aux.begin(), aux.end(), 0 );
}

double Estads::medin() const {
    double md;
    int idx = 0.5 * n;
    n % 2 ? md = v[idx] : md = 0.5 * ( v[idx - 1] + v[idx] );
    return md;
}
```

Uso

```
#include "estads.h"
#include <iostream>
#include <cstdlib>
#include <fstream>

int main() {
    using namespace std;
    cout << "Programa 13: una clase functor.\n\n";
    std::ifstream leeEntero( "C://temp//data.txt" );

    if ( leeEntero.fail() ) {
        std::cerr << "Fall\242 la lectura. Abortando.\n";
        return EXIT_FAILURE;
    }

    int valor;
    std::vector<int> vInt;

    while ( leeEntero >> valor ) {
        vInt.push_back(valor);
    } // while

    Estads stI = for_each( vInt.begin(), vInt.end(), Estads(vInt) );
    cout << "Media  =  " << stI.media() << '\n';
    cout << "Mediana = " << stI.medin() << '\n';
    cout << "Varz.  =  " << stI.varza() << '\n';
    cout << "Suma   =  " << stI.suma()  << '\n';
    cout << "Cuenta =  " << stI.cuent() << "\n\n";

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

Salida

- Con error: se mandó a leer un archivo inexistente.

```
Programa 13: una clase functor.

Falló la lectura. Abortando.
```

- Sin error: son los mismos valores que sus homónimos del Programa 29 (pág. 196) del tomo 1 y del Programa 16 (pág. 111) del tomo 2.

```
Programa 13: una clase functor.

Media  =  48.765
Mediana = 46.5
Varz.  =  821.518
Suma   =  9753
Cuenta =  200
```

La eficiencia fue del 27%

Programa 14-la plantilla

Una vez comprobado que el mecanismo de la case-functor trabaja, es tiempo de hacer el siguiente movimiento: transformar esa clase en una plantilla de cálculo de modo que se insensibilicen los tipos y los contenedores. Como estamos hablando de una plantilla, todo va en una cabecera. El resultado es el mismo que al aplicarle las reglas de transformación a las clases comunes y corrientes.

Declaración

Al transformar la clase functor a una plantilla, se insensibilizan los tipos, resolviendo uno de los problemas que se arrastraron hasta acá. Y bien mirado, se puede insensibilizar también al tipo de contenedor, pero se impone un análisis parecido al anterior: para este problema sólo sirven los tres contenedores asociativos básicos: el vector, la lista y la doble cola.

La plantilla se hizo para todos ellos insensibilizándola al contenedor, pero el vector hace todo el trabajo ya que se le asignan la lista o la doble cola. También hay que considerar que la suma de todos los valores depende del tipo de datos: no es lo mismo sumar enteros que sumar reales.

```
#ifndef ESTADS_H_INCLUDED
#define ESTADS_H_INCLUDED

#include <vector>
#include <list>
#include <deque>
#include <algorithm>

template <typename T> class Estads {
public:
    Estads( std::vector<T> & vct ) {
        v.assign( vct.begin(), vct.end() );
        sort( v.begin(), v.end() );
    }
```

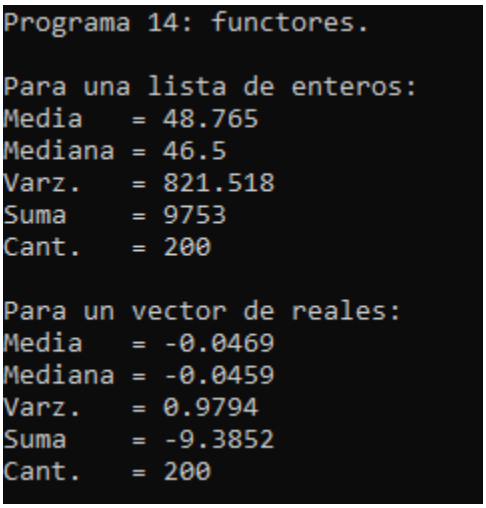
```
    Estads( std::list<T> & lst ) {
        v.assign( lst.begin(), lst.end() );
        sort( v.begin(), v.end() );
    }
    Estads( std::deque<T> & dek ) {
        v.assign( dek.begin(), dek.end() );
        sort( v.begin(), v.end() );
    }
    // al sobrecargar al operador () la plantilla es un functor
    void operator()( T & x ) {}
    // estos estadígrafos dependen del tipo dado
    T suma() const { return accumulate( v.begin(), v.end(), 0.0 ); }
    T SDC() const;
    // este estadígrafo siempre es entero
    int cuent() { return v.size(); }
    // estos estadígrafos siempre son dobles
    double media() const;
    double varza() const;
    double medin() const;
private:
    std::vector<T> v;
};

template<typename T> T Estads<T>::SDC() const {
    std::vector<T>aux( v );
    transform( aux.begin(), aux.end(), aux.begin(), aux.begin(), std::multiplies<T>() );
    return accumulate( aux.begin(), aux.end(), 0.0 );
}

template<typename T> double Estads<T>::media() const {
    return static_cast<double>( suma() ) / v.size();
}

template<typename T> double Estads<T>::varza() const {
    int n = v.size();
    return static_cast<double>( SDC() - suma() * suma() / n ) / ( n - 1 );
}

template<typename T> double Estads<T>::medin() const {
    int idx = 0.5 * v.size();
    double md = v[idx];

    if ( 0 == v.size() % 2 )
        md = 0.5 * ( v[idx - 1] + md );

    return md;
}

#endif // ESTADS_H_INCLUDED
```

Uso

```
#include "estads.h"
#include <fstream>
#include <iostream>
#include <cstdlib>
#include <iomanip>

int main() {
    using std::cout;
    std::cout << "Programa 14: functores.\n\n";
```

```
    cout << "Para una lista de enteros:\n";
    std::ifstream leeEntero( "C://temp//datos.txt" );

    if ( leeEntero.fail() ) {
        std::cerr << "Fallo de lectura. Abortando.\n";
        return EXIT_FAILURE;
    }

    std::list<int> lInt;
    int valor1;

    while ( leeEntero >> valor1 )
        lInt.push_back(valor1);

    Estads<int> stI = for_each( lInt.begin(), lInt.end(), Estads<int>(lInt) );
    cout << "Media   = " << stI.media() << '\n';
    cout << "Mediana = " << stI.medin() << '\n';
    cout << "Varz.   = " << stI.varza() << '\n';
    cout << "Suma    = " << stI.suma()  << '\n';
    cout << "Cant.   = " << stI.cuent() << "\n\n";

    cout << "Para un vector de reales:\n";
    std::ifstream leeDoble( "C://temp//gauss.txt" );

    if ( leeDoble.fail() ) {
        std::cerr << "Fallo de lectura. Abortando.\n";
        return EXIT_FAILURE;
    }

    std::vector<double> vDbl;
    double valor2;

    while ( leeDoble >> valor2 )
        vDbl.push_back(valor2);

    Estads<double> stD = for_each( vDbl.begin(), vDbl.end(), Estads<double>(vDbl) );
    cout << std::setprecision( 4 ) << std::fixed;
    cout << "Media   = " << stD.media() << '\n';
    cout << "Mediana = " << stD.medin() << '\n';
    cout << "Varz.   = " << stD.varza() << '\n';
    cout << "Suma    = " << stD.suma()  << '\n';
    cout << "Cant.   = " << stD.cuent() << "\n\n";

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

Salida



La eficiencia fue del 29% y los valores concuerdan.

Ahora el functor es insensible al contenedor, al tipo y al tamaño.

PROBLEMAS PROPUESTOS

No hay nada nuevo bajo el sol. Se tomarán algunos problemas ya resueltos bajo una óptica, que se harán con el auxilio de la STL y la POO. Es todo.

1. 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%; y 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. Se pide llevar una BD con los impuestos mensuales de una comunidad. La salida ordenada debería verse así:

```
Ejercicio #1: una BD.

El Reino del Revés.
Sistema de impuestos
1) Insertar una partida
2) Mostrar la BD
3) Escribir la BD a disco
4) Contar las partidas
5) Ver si una partida está
6) Eliminar una partida
7) Vaciar la BD
0) Terminar
Entre una opción: 2

El Reino del Reves - Sistema de Impuestos.
Thu Nov 19 03:16:08 2020
Nombre                                  Ganó       Impuesto
Pastas Alimenticias, Lmtd.............38951.00........4790.20
Aves y Huevos, SA.....................19873.00........1730.95
Dulces & Bebidas, Cía.................9513.00.........451.30

Presione una tecla para continuar . . .
```

2. 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 registra las cantidades de cada una, además el costo del préstamo, quién hizo el pedido y en qué fecha lo hizo. Diseñar una clase adecuada al problema y escribir un pequeño manejador para probarla. La salida deberá verse así:

```
Problema 30: 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 - d/mm/aaaa: 8/10/2020
   Sierra de calar = 13
 Sierra de trocear = 13
      Atornillador = 15
    Lijadora plana = 2

Hoja de pedidos.
Salvador Aldorso - Fecha: 8/10/2020
Herramienta         Precio   Cant.      Costo
Sierra de calar       6.80      13       88.40
Sierra de trocear     8.99      13      116.87
Atornillador          3.55      15       53.25
Lijadora plana       11.22       2       22.44
Costo total:.............................280.96$
```

3. En las calculadoras modernas se puede buscar la equivalencia en el cambio de unidades entre magnitudes físicas. Codificar un programa bajo los conceptos de la OOP, que emule parte de de una calculadora de conversión. 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 al menor ángulo posible. La salida deberá verse así:

```
Problema 9: calculadora de conversión.

1) Galón USA(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 Celsius(C)
7) Radián(rad) y grado(grado)
0) Terminar

Entre una opción: 7
Unidades a convertir: 39

 39.0000 rad   -->  74.5354 grado
 39.0000 grado -->   0.6807 rad
```

4. 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. Al ver su éxito, aprobó pasar la demo anterior a producción. Las tente-en-pie llevan cada una su marca, peso (g), cantidad de calorías (cal), precio de venta (CUC) y cantidad de unidades (u) en inventario. Realizar la tarea, usando estos datos:

Marca	Peso	Cals	Precio	Unids
Mocha Munch	65	350	0.9	208
Krispy Krunch	70	340	1.05	411
Honey Beez	90	390	1.25	789

La salida deberá verse así:

```
Ejercicio 4: una base de datos.

1) Imprimir inventario
2) Adicionar un producto
3) Eliminar un producto
4) Editar un producto
5) Vaciar la base de datos
6) Ayuda
0) Terminar
Seleccione una opción: 1
================================================
Marca                 Peso   Cals.   Cant. Precio
------------------------------------------------
Mocha Munch            65     350     208    0.90
Krispy Krunch          70     340     411    1.05
Honey Beez             90     390     789    1.25
DeCeballos            400     100     753    1.00
================================================

Hay 4 marcas diferentes,
con un total de 2161 u en almacén,
para un total de 2358.00 CUC
```

5. Para formar una bibliografía bajo los conceptos de la OOP, crear una lista ordenada de cadenas C++ y manejarla por menú. El programa tomará nombre y apellido del autor, y título de la obra. Tomar estos nombres y entrarlos en este orden: Richard Halterman. *C++ programming*; Rex Jaeschke. *Punteros Void*; 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. *Introducción al C++*. No olvidar que la bibliografía debe listar cada autor en línea aparte. Una salida parcial sería:

```
Ejercicio 5: manejo de una biblioteca.

 BIBLIOTECA
 1) Insertar una o varias obras.
 2) Mostrar la biblioteca.
 3) Contar las obras.
 4) Buscar un autor.
 5) Buscar una obra.
 6) Eliminar una obra.
 7) Eliminar un autor.
 8) Vaciar la biblioteca.
 9) Ayuda.
 0) Terminar.
Entre una opción: 2
Autor              Título
Eckel, B.............Pensar en C++
Halterman, R........C++ programming
Jaeschke, R.........Punteros void
```

6. Se desea codificar un programa bajo los conceptos de la OOP que tome las horas totales trabajadas en el mes por los obreros de la CPA Santos García y emita una nómina mensual de pago, sabiendo que la empresa toma cada mes como de 4 semanas. La aplicación será manejada por menú. La nómina debe ir a consola y al archivo C:/temp/BD/nomina.tx, la nómina estará fuertemente ordenada y nuevos obreros podrán ser insertados en cualquier momento. Para el desarrollo y puesta a punto de esta solución, usar los datos siguientes:

Trabajador	Horas
Salvador Mancuso	167
Maricusa Avogadro	158
Daniel Montero	170
Carlos Avogadro	161
Santos Aldorso	163
Fabia Almudena	148

Después de tomar la opción 2 y luego tomar la 6, la salida debiera verse así:

```
CPA S. García - Nómina
Fecha: Mon Oct 26 12:43:27 2020

Obrero              Horas          Salario
Aldorso, Santos.......163..........2056.25  _____________
Almudena, Fabia.......148..........1850.00  _____________
Avogadro, Carlos......161..........2018.75  _____________
Avogadro, Maricusa....158..........1975.00  _____________
Mancuso, Salvador.....167..........2131.25  _____________
Montero, Daniel.......170..........2187.50  _____________
Total.............................12218.75$ Firma
```

7. Ya se conocen los estadísticos descriptivos principales: los valores máximo y mínimo, la varianza, la mediana y la moda. El rango y la desviación típica se calculan a partir de estos. Usando valores enteros ya conocidos, calcularlos con una clase functor para enteros. La salida debiera verse así:

```
Ejercicio 7: una clase functor.

Media    =   48.7650
Mediana =   46.5000
Moda     =    6
Varz.    =  821.5176
Desviac.=   28.6621
Máximo   =   99
Mínimo   =    1
Rango    =   98
Suma     = 9753
Cuenta   =  200
```

8. Mediante su transformación a plantilla, insensibilizar la clase-functor anterior a tipos y contenedores secuenciales. La salida debiera verse así:

```
Ejercicio 8: una plantilla functor.

Para una lista de enteros:
Media   =   48.7650
Mediana =   46.5000
Moda    =    6
Varz.   =  821.5176
Desviac.=   28.6621
Máximo  =   99
Mínimo  =    1
Rango   =   98
Suma    = 9753
Cuenta  =  200

Para un vector de reales:
Media   =   -0.0469
Mediana =   -0.0459
Moda    =  N/A
Varz.   =    0.6762
Desviac.=    0.8223
Máximo  =    2.8108
Mínimo  =   -2.1563
Rango   =    4.9671
Suma    =   -9.3852
Cuenta  =  200
```

9. Completar la base de datos polimórfica de modo que efectúe las siguientes acciones: borrar, editar, encontrar un valor y contar los nodos en existencia. Manejarla por menú. La salida debiera verse así:

```
Ejercicio 9: BD polimórfica.

1) Entrar un nodo
2) Mostrar la base
3) Encontrar un valor
4) Borrar un nodo
5) Contar los nodos
0) Terminar

Entre una opción: 2

Pieza No.: 120  Modelo del año 2016
Pieza No.: 123  Modelo del año 2019
Pieza No.: 345  Núm. motores = 2
Pieza No.: 348  Núm. motores = 4

Presione una tecla para continuar . . .
```

10. 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.). Use su cronógrafo para cronometrar estos tiempos.

```
Ejercicio 10: 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 0.699217 min.

Este sistema es:
76.6928% mejor que la AT.
91.2598% mejor que la PC.
```

Los tiempos son consistentes con los hallados anteriormente, pero varían levemente entre sucesivas ejecuciones.

11. En el programa 15 de (Introducción al C++, tomo 1, pág. 104) se tenía como objetivo el trabajar un arreglo numérico lineal estático y se pedía calcular y mostrar la nota promedio (de cero a 100 puntos) de un alumno que cursaba 5 asignaturas y en el programa 16 (pág. 108) se pedía hacer lo mismo, pero para una tabla con seis notas y tres alumnos. Ese escalamiento y reúso se resolvió en el programa 6 del (Introducción al C++, tomo 2, pág. 62). Usar estos datos.

Asignat.	Matemáticas	Español	Física		
Alumno	63	97	88		
Asignat.	**Matemáticas**	**Español**	**Física**	**Química**	
Alumno 1	63	97	88	77	
Alumno 2	96	87	89	78	
Asignat.	**Matemáticas**	**Español**	**Física**	**Química**	**Inglés**
Alumno 1	63	97	88	77	93
Alumno 2	96	87	89	78	80
Alumno 3	70	90	86	81	90

Rehacer el programa, pero ahora bajo los conceptos de la OOP. La salida sería con dos alumnos sería así:

```
Ejercicio 11: una solución reúsable.

¿Cuántos alumnos? 2

Tabla de resultados
=============================================
Asig.    Mat.    Esp.    Fis.   Quim.   Prom.
---------------------------------------------
Al. 1    63.0    97.0    88.0    77.0    81.2
Al. 2    96.0    87.0    89.0    78.0    87.5
---------------------------------------------
Prom.    79.5    92.0    88.5    77.5
=============================================
```

12. Se resolvió una LSE en el problema 24 de (Introducción al C++, tomo 1, pág. 125) y en la (Introducción al C++, tomo 2, pág. 68) se solucionó con la STL pura en el programa 7, pero en ISO C++ el manejo de una lista se hace mediante una clase adecuada y los recursos de la STL. Crear una solución insensible tanto al tipo como al contenedor secuencial. La salida se vería así:

```
Ejercicio 12: manejo de una LSE.

Con una lista de 10 enteros.
  15, 10, 30, 32, 1, 31, 48, 19, 46, 44
  El valor 25 no está.

Con un vector de 5 reales.
  30.2312, 2.71762, 25.3654, 35.5065, 31.5855
  Ahora son 6 nodos ordenados.
  2.2222, 2.71762, 25.3654, 30.2312, 31.5855, 35.5065

Con una doble cola de 8 cortos.
  41, 16, 21, 47, 28, 1, 15, 33
  Ahora la lista está vacía.
```

13. Montar la jerarquía de figuras geométricas manejándola por menú y poniendo sus características. Como no se puede dibujar una figura geométrica en modo texto, se anuncia. La salida se vería así:

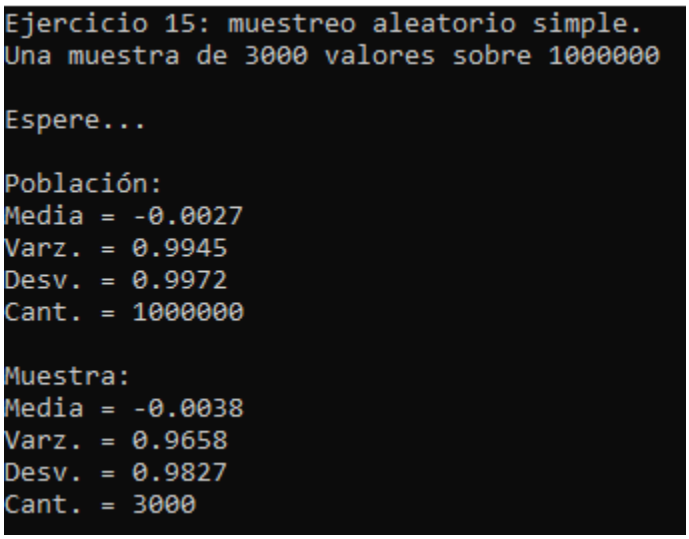
```
Ejercicio 13: las secciones cónicas.

(1)Hiperbola, (2)Parabola, (3)Elipse
(4)Circunferencia, (0)Terminar
Entre una opción: 4

La circunferencia es el lugar geométrico de los puntos del plano
que equidistan de otro punto fijo y coplanario llamado centro, en
una cantidad constante llamada radio.

Circunferencia dibujada.

Presione una tecla para continuar . . .
```

14. He aquí un caso en que el tipo de dato **char** del ANSI C es indispensable para entrar el dato. Una calculadora aritmética RPN es aquella que recibe su entrada en notación polaca en reverso o notación posfija. Por ejemplo, para sumar 3 + 2 se entra 3 2 +. Se implementa usando una pila simple y si ya la STL trae una, ¿por qué no usarla?

```
Ejercicio 14: ejemplo de calculadora RPN.
Una expresión legal RPN es: 12 5 + que
suma 12 con 5. Termine su cálculo con la
letra F

> 12 5 + 2 * 6 + 8 /f
5
```

El cálculo muestra $12 + 5 * 2 + \frac{6}{8} = 17 * 2 + \frac{6}{8} = 34 + \frac{6}{8} = \frac{40}{8} = 5$

15. La estadística inferencial comprende los métodos y procedimientos que determinan las propiedades de una población a partir de una pequeña parte de esta. El muestreo aleatorio simple o MAS es el que mejor se aconseja y si la muestra es verdaderamente grande, se puede suponer que cada lectura es única. Codificar un programa que genere un millón de valores gaussianos y les calcula los verdaderos valores o estadígrafos. Entonces tomar una muestra al azar de 2000 valores, calcular sus estadísticas y constatar contra las respuestas reales. ¿Está la realidad en concordancia con la teoría?



La teoría predice una buena concordancia entre muestra y población cuando se hace un MAS con el 3% de la misma. Y eso pasa aquí.

APÉNDICES

1-Organizaciones normativas para C++

Tabla 8. Normas norteamericanas que afectan a C++

ANSI	Instituto norteamericano nacional de normas
X3	Comité ANSI para la información de sistemas de procesamiento
X3J16	El comité técnico que normaliza a C++ en Norteamérica

Tabla 9. Normas internacionales relacionadas con C++

ISO	La organización internacional para la normalización, que engloba sucursales tales como ANSI (USA), AFNOR (Francia), BSI (Reino Unido), DIN (Alemania), JTSC (Japón), SCC (Canadá), etc.
IEC	La comisión electrotécnica internacional
JTC1	Joint Task Committee 1: el comité conjunto (ISO & IEC) en tecnología de la información
SC22	Sub Committee 22: el subcomité de JTC1 para lenguajes de programación.
WG21	Work Group 21: el grupo de trabajo de SC22 para la normalización de C++

2-Bibliotecas de terceras personas

Bibliotecas BOOST

Es un conjunto de bibliotecas de software libre y revisión por pares preparadas para extender las capacidades del lenguaje de programación C++. Las Bibliotecas BOOST de C++ son gratuitas, de código fuente abierto, creadas por miembros de la comunidad de C++ y pueden ser utilizadas por los programadores de C++ que trabajan en una amplia variedad de plataformas, con muchos compiladores distintos.

La idea de un repositorio en línea de bibliotecas de C++ gratuitas, de código fuente abierto, se propuso por primera vez en un artículo escrito por Beman Dawes: *Proposal for a C++ Library Repository Web Site*, Beman G. Dawes, mayo 6, 1998, a ver en **`http://www.boost.org/more/proposal.pdf`**. Él y Robert Klarer obtuvieron la idea mientras asistían a una reunión del comité de normalización de C++.

El artículo sugería emplazar un sitio Web en el que los programadores de C++ pudieran buscar y compartir bibliotecas, y patrocinar el posterior desarrollo de C++. Esa idea se desarrolló eventualmente para convertirse en las Bibliotecas BOOST en **`http://www.boost.org`**.

BOOST ha crecido hasta tener 70 bibliotecas, y se agregan más con frecuencia

Los creadores originales Beman Dawes, Robert Klarer y David Abrahams siguen activos en la comunidad y alrededor de 3.000 personas están suscritas a la lista de correo, donde decenas de ellos son muy activos.

Bibliotecas Blitz++

El proyecto Blitz++ aporta computación numérica de alto rendimiento al lenguaje C++. En ciertos aspectos, es lo que debería haber sido el **`valarray`**. El Blitz++ tiene un potente conjunto de operadores que trabajan sobre los arreglos numéricos y las matrices, y funciona con zancadas, subconjuntos, etc. El paquete está escrito para minimizar el número de objetos temporales innecesarios y aprovecharse del tiempo de compilación, vía meta-programación de plantillas, siempre que sea posible.

Una de las optimizaciones clave es los operadores aritméticos y las funciones matemáticas involucradas en el cómputo de arreglos Blitz++ no computan los valores inmediatamente. En lugar de eso, devuelven objetos de expresión. Cuando un objeto de expresión es asignado a un arreglo, la expresión es computada, almacenando los resultados directamente en el blanco de la asignación, sin necesidad de crear grandes arreglos temporarios.

STLport

El proyecto STLport es una implementación libre de la biblioteca estándar de C++. Aunque cada compilador moderno viene más o menos con una biblioteca estándar completa, permanecen diferencias, omisiones y errores en la mayoría de las suministradas por los proveedores. Si la portabilidad a través de compiladores y plataformas es una prioridad, Ud. podría querer usar la implementación de esta biblioteca en todas las plataformas.

Ud. también podría querer usar STLport por sus características adicionales, tales como su modo de depuración, lo cual ayuda detectar errores de los programadores. Incluye también extensiones a la biblioteca estándar, como contenedores por hash, LSE y cuerdas, cadenas de C/C++ que escalan muy bien para muy grandes tamaños.

3-El fichero makefile

Es un fichero utilitario que permite la compilación de los ficheros-fuente y los transforma en **.exe** El sistema de compilación reconoce varios mandatos y obra en consecuencia. Makefile es un fichero de texto con extensión **.mak** y se invoca desde el **make** correspondiente, que en este sistema se llama **mingw32-make.exe** y está localizado en C:\MinGW\bin\.

La versión aquí utilizada es:

```
GNU Make 3.82.90
Built for i686-pc-mingw32
Copyright (C) 1988-2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

El **make** tiene muchos símbolos, pero los más importantes son:

Tabla 10 (simplificada) Los símbolos del make

Comando	Resultado
-c	La salida
-C DIRECTORY **--directory=DIRECTORY**	Cambia al directorio DIRECTORY antes de compilar.
-f FILE **--file=FILE** **--makefile=FILE**	Lee el fichero como si fuera un **makefile**
-g	Pone los símbolos de la depuración
-I DIR (es una i mayúscula)	Incluye al directorio DIR
-o	Cómo se llama el fichero objeto
-O2	Optimiza para velocidad
-s	Quita los símbolos de la depuración
--silent **--quiet**	No muestra las reglas.
-std=c++11	Cumple con la norma C++11 del ISO C++
-Wall	Activa todas las advertencias
-Wnon-virtual-dtor	Clase virtual con destructor no virtual
-Wunreachable-code	Código que no se ejecuta
-Wzero-as-null-pointer-constant	Puntero nulo inicializado con cero

Para hacer esta prueba se organizó un directorio test debajo del directorio temp y se copiaron los ficheros **main.cpp**, **shell.h** y **shell.cpp** del ordenamiento Shell. El resultado sigue:

```
c:\temp\test>dir
 El volumen de la unidad C es Simon
 El número de serie del volumen es: 9027-2A1B

 Directorio de c:\temp\test

02/11/2020  05.59 PM    <DIR>          .
02/11/2020  05.59 PM    <DIR>          ..
02/11/2020  04.52 PM                39 compilar.bat
06/09/2020  06.12 AM               794 main.cpp
02/11/2020  01.16 PM               175 Makefile.mak
06/09/2020  06.12 AM             1,234 shell.cpp
06/09/2020  06.12 AM               833 shell.h
               5 archivos          3,075 bytes
               2 dirs  58,753,712,128 bytes libres
```

El fichero compilar.bat está compuesto por dos comandos DOS: el primero compila y el segundo borra los enlaces.

```
c:\temp\test>type compilar.bat
mingw32-make -f makefile.mak
del *.o
```

El **`makefile`** está compuesto por reglas y comandos. El más simple es una regla que llamamos PRUEBA que usa los ficheros shell.cpp y shell.h. Enlaza a main.cpp con main.o; a shell.cpp con shell.o y todo lo compila en un fichero ejecutable llamado shell.exe

```
c:\temp\test>type makefile.mak
# Regla
PRUEBA: shell.cpp shell.h
        g++ -Wall -O2 -std=c++11 -c "main.cpp" -o main.o
        g++ -Wall -O2 -std=c++11 -c "shell.cpp" -o shell.o
        g++ -o shell.exe main.o shell.o -g
```

El resultado sigue:

```
c:\temp\test> compilar

c:\temp\test>mingw32-make -f makefile.mak
g++ -Wall -O2 -std=c++11 -c "main.cpp" -o main.o
g++ -Wall -O2 -std=c++11 -c "shell.cpp" -o shell.o
g++ -o shell.exe main.o shell.o -g

c:\temp\test>del *.o
```

Y al ejecutar el fichero shell.exe tenemos:

```
c:\temp\test>shell
Problema 39: ordenamiento Shell.

Trabajando...tardé 2.798 s en ordenar 1500000 enteros.

Presione una tecla para continuar . . .
```

El directorio ahora tiene:

```
c:\temp\test>dir
 El volumen de la unidad C es Simon
 El número de serie del volumen es: 9027-2A1B

 Directorio de c:\temp\test

02/11/2020  06.00 PM    <DIR>          .
02/11/2020  06.00 PM    <DIR>          ..
02/11/2020  04.52 PM                39 compilar.bat
06/09/2020  06.12 AM               794 main.cpp
02/11/2020  01.16 PM               175 Makefile.mak
06/09/2020  06.12 AM             1,234 shell.cpp
02/11/2020  06.00 PM         1,925,517 shell.exe
06/09/2020  06.12 AM               833 shell.h
               6 archivos      1,928,592 bytes
               2 dirs  58,751,782,912 bytes libres
```

Con la ayuda de reglas mucho más complejas el trabajo es más eficiente que con el IDE, pero aquél lo automatiza todo y nos quita las preocupaciones de encima.

4- Métodos y Miembros Estáticos

> *Los objetos son instancias de clases y aunque la clase es una, puede haber dos o más objetos de la misma. La clase de dato es común a todas, pero las instancias guardan su propio valor.*

Si se tiene cinco objetos Perro, cada uno de ellos tiene su propio nombre, su edad y peso, etc. Un objeto no afecta a otro. Sin embargo, hay momentos en que se desea seguirle la pista a un conjunto de objetos derivados de una clase. Por ejemplo, querer saber cuántos los objetos de una clase específica han sido creados en el programa, o de una clase dada cuántos están en existencia en un instante dado, etc. Una buena manera de hacerlo es mediante los miembros estáticos.

Las clases pueden tener miembros estáticos —datos globales de la clase disponibles en todo su programa. Esta clase controla los animales en común y sólo se sabe cuál es por su habla. Al ser público el dato, no está protegido. Observar dónde se declara el miembro estático y cómo se llama.

Programa auxiliar 1-miembro estático público

Definición y declaración

```
#ifndef ANIMAL_H
#define ANIMAL_H

#include <iostream>

class Animal {
public:
    Animal() {
        edad = 0;
        ++cuantosSon;
    }
    virtual ~Animal() { --cuantosSon; }
    virtual short getEdad() const { return edad; }
    virtual void  setEdad( short valor ) { edad = valor; }
    virtual void Habla() const {}
    static int cuantosSon;
private:
    short edad;
};

int Animal::cuantosSon = 0;

class Perro : public Animal {
public:
    Perro() {}
    virtual ~Perro() {}
    virtual void Habla() const { std::cout << "\255Guau, Guau!\n"; }
};

class Gato : public Animal {
public:
    Gato() {}
    virtual ~Gato() {}
    virtual void Habla() const { std::cout << "Miau...\n"; }
};

#endif // ANIMAL_H
```

Uso

```
#include "animal.h"
#include <iostream>
#include <cstdlib>

const int TAM = 4;

int main() {
    std::cout << "Programa auxiliar 1: m\202todos est\240ticos.\n\n";
    Animal *camada[TAM];

    for ( int i = 0; i < TAM; i++ ) {
        if ( 0 == i % 2 ) {
            camada[i] = new Perro;
            camada[i]->setEdad( i + 2 );
        } else {
            camada[i] = new Gato;
            camada[i]->setEdad( i + 2 );
        } // if-else
    } // for

    for ( int i = 0; i < TAM; i++ ) {
        1 == Animal::cuantosSon
            ? std::cout << "Hay un animal.\n"
            : std::cout << "Hay " << Animal::cuantosSon << " animales.\n";

        std::cout << "Borrando a uno de " << camada[i]->getEdad() << " a\244os. ";
        camada[i]->Habla();
        std::cout << '\n';
        delete camada[i];
        camada[i] = nullptr;
    } // for

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

Salida

```
Programa auxiliar 1: métodos estáticos (1).

Hay 4 animales.
Borrando a uno de 2 años. ¡Guau, Guau!

Hay 3 animales.
Borrando a uno de 1 años. Miau...

Hay 2 animales.
Borrando a uno de 4 años. ¡Guau, Guau!

Hay un animal.
Borrando a uno de 3 años. Miau...
```

Programa auxiliar 2-método público

El dato está en un miembro global hecho público y puede llamarse directamente, o desde una función normal. Pero ese dato no está protegido. Mejor es hacerlo privado y acceder a él mediante su geter. Muy poco cambia, pero ahora el dato está protegido.

Definición y declaración

```
#ifndef ANIMAL_H
#define ANIMAL_H

#include <iostream>

class Animal {
public:
    Animal() {
        edad = 0;
        ++cuantosSon;
    }
    virtual ~Animal() { --cuantosSon; }
    virtual short getEdad() const { return edad; }
    virtual void  setEdad( short valor ) { edad = valor; }
    virtual void Habla() const {}
    static int getAnimal() { return cuantosSon; }
private:
    short edad;
    static int cuantosSon;
};

int Animal::cuantosSon = 0;

class Perro : public Animal {
public:
    Perro() {}
    virtual ~Perro() {}
    virtual void Habla() const { std::cout << "\255Guau, Guau!\n"; }
};

class Gato : public Animal {
public:
    Gato() {}
    virtual ~Gato() {}
    virtual void Habla() const { std::cout << "Miau...\n"; }
};

#endif // ANIMAL_H
```

Uso

```
#include "animal.h"
#include <iostream>
#include <cstdlib>

const int TAM = 4;

void cuantos();

int main() {
    std::cout << "Programa auxiliar 2: m\202todos est\240ticos.\n\n";
    Animal *camada[TAM];
```

```
    for ( int i = 0; i < TAM; i++ )
        if ( 0 == i % 2 ) {
            camada[i] = new Perro;
            camada[i]->setEdad( i + 2 );
        } else {
            camada[i] = new Gato;
            camada[i]->setEdad( i + 2 );
        } // if-else

    for ( int i = 0; i < TAM; i++ ) {
        1 == camada[i]->getAnimal()
            ? std::cout << "Hay un animal.\n"
            : std::cout << "Hay " << camada[i]->getAnimal() << " animales.\n";

        std::cout << "Borrando a uno de " << camada[i]->getEdad() << " a\244os. ";
        camada[i]->Habla();
        std::cout << '\n';
        delete camada[i];
        camada[i] = nullptr;
    } // for

    system( "pause" )     ;
    return EXIT_SUCCESS;
}

void cuantos() {
    std::cout << Animal::getAnimal();
}
```

Salida

```
Programa auxiliar 2: métodos estáticos.

Hay 4 animales.
Borrando a uno de 2 años. ¡Guau, Guau!

Hay 3 animales.
Borrando a uno de 3 años. Miau...

Hay 2 animales.
Borrando a uno de 4 años. ¡Guau, Guau!

Hay un animal.
Borrando a uno de 5 años. Miau...
```

5-Manejo de errores

> *Una excepción es la indicación de un problema que ocurre durante la ejecución de un programa. El nombre "excepción" implica que el problema ocurre con poca frecuencia…El manejo de excepciones permite a los programadores crear aplicaciones que puedan resolver (o manejar) las excepciones…Las características que presentamos…permiten a los programadores escribir programas tolerantes a fallas, que (a) puedan, sin dejar de ejecutarse, tratar con problemas que puedan surgir, o (b) que terminen de una manera no dañina. El estilo y los detalles sobre el manejo de excepciones en C++ se basan, en parte, en el trabajo que Andrew Koenig y Bjarne Stroustrup presentaron en su artículo* Exception Handling for C++ *(versión revisada).* Proceedings of the Usenix C++ Conference*, pp. 149-176, San Francisco, abril de 1990. (Deitel & Deitel, pág. 683)*

C++ ofrece un Sistema de tratamiento de errores basado en la biblioteca <stdecept> y una clase-madre esception. De ella derivan las clases (más bien, plantillas de clases) que construyen objetos que maneja los errores.

El manejo de excepciones NO está diseñado para procesar los errores asociados con los eventos síncronos, como son el completar la E/S de disco, la llegada de mensajes de red, los clics del ratón y las pulsaciones de tecla, los cuales ocurren en paralelo con —y de manera independiente a— el flujo de control del programa. Está diseñado para procesar los errores que ocurren cuando se ejecuta una instrucción. Por ej., los subíndices de arreglos fuera de rango, el desbordamiento aritmético (es decir, un valor fuera del rango de valores representables), la división entre cero, los parámetros de función inválidos y la asignación fallida de memoria (debido a la falta de memoria).

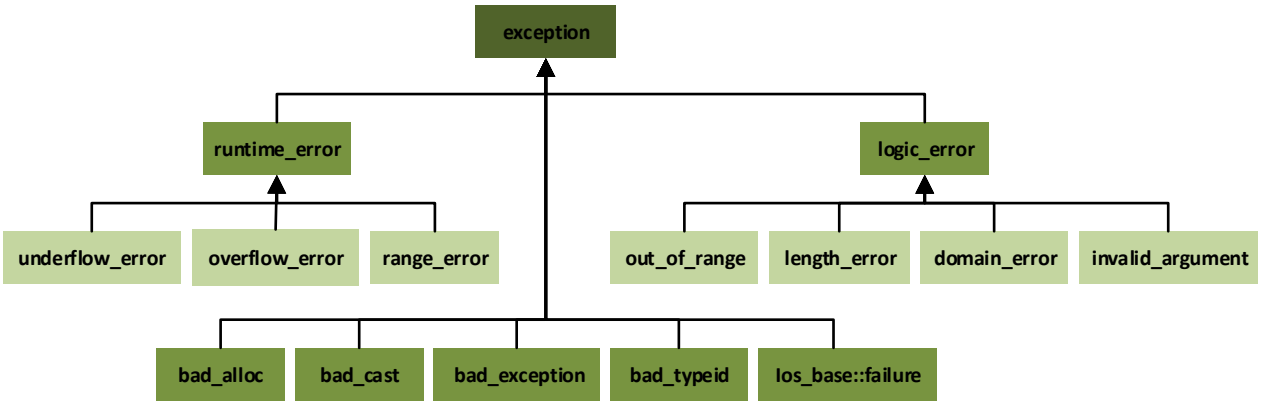


Ilustración 12. Clases de error

Para ello se va a derivar de la clase **`runtime_exception`** por herencia pública a una clase que va a "hablar" en español. La clase-madre se encuentra en la biblioteca **`<stdexcept>`**

Bloque try-catch

El sistema de tratamiento de errores se apoya en un mecanismo especial de C++: el bloque **`try-catch`**. Aunque ya se habló de él, ahora se puntualizará.

1. La palabra-clave **`try`** ejecuta su bloque y si alguna otra instrucción de allí lanza una excepción (y no es atrapada y manipulada por otro), los manipuladores de excepción son probados para ver si cualquier de ellos puede manejarla. Cada manipulador **`catch`** es probado a su vez. El primero que corresponda a la excepción, la maneja. Si ningún **`catch`** casa, la excepción sube por la pila de llamada a la siguiente declaración. Si no hay más y no ha casado, llama a la instrucción **`terminate()`** y el programa termina su ejecución.

   Nota: el bloque es verdadero, significando que, si se declara alguna variable allí, cuando este termina, ella se va.

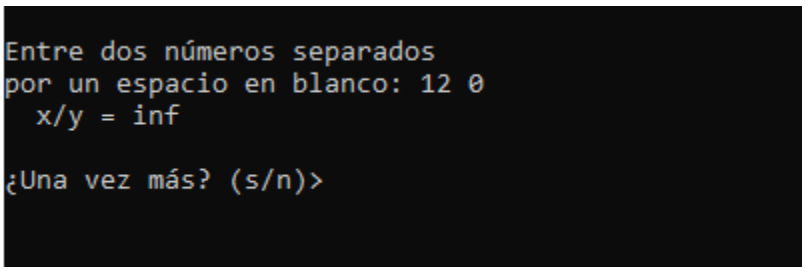
2. La palabra-clave **catch** introduce un manipulador de excepción en una declaración **try**, a la cual sigue inmediatamente y quien tiene una o más de ellas. Cada declaración **catch** declara un objeto que maneja a esa excepción. Si una excepción es lanzada, el tipo del objeto creado es comparado con el tipo de cada declaración **catch** del bloque y el primero que le case es ejecutado. El último bloque **catch** puede tener una elipsis (…), para corresponder a todas las excepciones.
3. **throw** es un operador que lanza una expresión tipo **void** como excepción desde cualquier parte de la aplicación. Sin operando, **throw** relanza la excepción actual que está pendiente. Si no hay ninguna, llama a la instrucción **terminate()** y el programa termina su ejecución.

C++ declara una constante **what** que devuelve un mensaje con la descripción del sistema acerca de la naturaleza del error:

```
virtual const char* what() const throw();
```

Programa auxiliar 3-división por cero

Se mostrará cómo puede usarse el sistema, de modo que un programa reaccione a una excepción "hablando" en español. Para ello se intentará dividir por cero. En este SO, la aplicación muestra:



Ocurre un error de desbordamiento u **overflow**, que es cuando el sistema origina un número muy grande, pero que no puede manejar, de ahí que produzca la salida **inf** que quiere decir infinito. Del árbol jerárquico de los errores vemos que éste se origina a partir de la clase **runtime_error**, así que se va a producir un error con mayor información. Para ello se va a heredar públicamente de esa clase.

Declaración

Se declara la biblioteca que contiene a la clase de la cual se va a heredar, y también a la clase propiamente.

```
#ifndef ERROR_H_INCLUDED
#define ERROR_H_INCLUDED

#include <stdexcept>       // contiene el sistema de clases
using std::overflow_error; // la clase del error por desbordamiento

// el objeto DivisionPorCero hereda públicamente de la clase
// runtime_error y será lanzado cuando se detecte este error
struct DividePorCero : public overflow_error {
    // el constructor especifica el mensaje de error
    DividePorCero() : overflow_error( "  No puedo dividir por cero." ) {}
};

// entrada: dos números reales
// salida: la división
double entre( double x, double y );

#endif // ERROR_H_INCLUDED
```

Definición

```
#include "error.h"

double entre( double x, double y ) {
    if ( 0 == y )
        throw DividePorCero();
    else
        return x / y;
}
```

Uso

```
#include "error.h"
#include <iostream>
#include <cstdlib>
#include <iomanip>

const std::string PROMPT( "\nEntre dos n\243meros separados\npor un espacio en blanco: " );

int main() {
    std::cout << "Programa auxiliar 3: manejo de excepciones.\n";
    double n1, n2; // los dos números
    char sn;       // ¿si o no?
    std::cout << PROMPT
              << std::fixed << std::setprecision( 2 ) << std::setw( 8 );

    while ( std::cin >> n1 >> n2 ) {
        try {
            std::cout << "x/y = " << entre( n1, n2 );
        } catch ( DividePorCero & e ) {
            std::cerr << e.what();
        }

        std::cout << "\n\n\250Una vez m\240s? (s/n)> ";
        std::cin >> sn;

        if ( 'N' == toupper( sn ) )
            return EXIT_SUCCESS;
        else
            std::cout << PROMPT;
    } // while cin
}
```

Salida

```
Programa auxiliar 3: manejo de excepciones.

Entre dos números separados
por un espacio en blanco: 12.3 7.8
  x/y = 1.58

¿Una vez más? (s/n)> s

Entre dos números separados
por un espacio en blanco: 12.3 0
  No puedo dividir por cero.

¿Una vez más? (s/n)> n
```

¿Cuál es la ganancia?

¿Cuál es la ganancia sobre el programa original que planteaba directamente el error?

```
const std::string ERR("No puedo dividir por cero.\n");

double entre( double x, double y ) {
   if ( 0 == y )
      throw ERR;
   else
      return x/y;
}

try {
   pFunc = entre;
   std::cout << "x/y = " << std::setw(2) << pFunc(n1, n2);
} catch (std::string) {
   std::cerr << '\n' << ERR;
}
```

Lo más obvio es que la solución dada aquí emplea las clases C++ de error, probadas y validadas hasta la saciedad, afinando el tipo de error, evitándole al lector alargamiento por gusto del código y pérdidas innecesarias de tiempo.

BIBLIOGRAFÍA

Budd, T. A. (2005). *Introducción a la Programación Orientada a Objetos.* La Habana, Habana, Cuba: Ediciones R.

Deitel, H. M., & Deitel, P. J. (2008). *Como programar en C++.*

Deitel, H., & Deitel, P. (1995). *Cómo programar en C/C++.* Naualcapán, Juárez, MX: Prentice Hall Hispanoamericana, SA.

Dubois, L. (1996). *Delphi 2.0* (Primera ed.). (A. Brugués, Trad.) Barcelona, España: Ediiones Gestión 2000 SA.

Eckel, B. (2000). *Pensar en C++* (Vol. I). Madrid, España: McGraw-Hill Interamericana de España SAU.

Galliano Vidal, S. A. (2020). *Introducción al C++, tomo 1* (Primera ed., Vol. I). (D. G. Orozco, Ed.) Manzanillo, Granma, Cuba. Obtenido de http://www.enciclopedia-manzanillo.cu

Galliano Vidal, S. A. (2020). *Introducción al C++, tomo 1* (Primera ed., Vol. I). (D. G. Orozco, Ed.) Manzanillo, Granma, Cuba. Obtenido de http://www.enciclopedia-manzanillo.cu

Galliano Vidal, S. A. (2020). *Introducción al C++, tomo 2* (Primera ed., Vol. II). (D. G. Orozco, Ed.) Manzanillo, Granma, Cuba. Obtenido de http://www.enciclopedia-manzanillo.cu

García de Jalón, J., Rodríguez, J., Sarriegui, J., & Brazález, A. (1998). Aprenda C++ como si estuviera en primero. (J. G. Jalón, Ed.) Pamplona, Navarra, España: ESIISS. Recuperado el 2019

Halterman, R. L. (2018). *Fundamentals of C++ programming.* USA: Southern Adventist University, School of Computing.

Hazzah, A. (1990, july). Encapsulation, Inheritance and Late-Binding in C++. *C++ User Journal, VIII*(7). Retrieved july 2020

Josuttis, N. M. (1999, august). *The C++ Standard Library: A Tutorial and Reference.* USA: Addison Wesley.

Lipschutz, S. (1991). *Estructura de Datos.* Habana, Habana, Cuba: Ediciones R.

Lischner, R. (2003, may). *C++ in a Nutshell.* USA: O'Reilly.

Prata, S. (2012). C++ Primer Plus. (Sexta ed.). Upper Saddle River, New Jersey, USA: Pearson Education, Inc. 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* (Tercera ed.). USA: McGraw-Hill/Osborne.

Stroustrup, B. (1991). *The C++ Programming Language* (2nd ed.). USA: Addison-Wesley.

Sutter, H. (2001). Exceptional C++. (I. Addison Wesley Longman, Ed.) Indianápolis, IN, USA.

Sutter, H. (2001). More Exceptional C++. (I. Addison Wesley Longman, Ed.) Indianápolis, IN, USA.

VOX: diccionario de uso del español de América y España. (s.f.). Recuperado el marzo de 2019

Wiegand, G. (1998). *Teach yourself C++ in 21 days.* Obtenido de http://www.mcp.com.

Wikipedia en español. (2016). *https://es.wikipedia.org/*, Español. Recuperado el 2018

noviembre 2020

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 que cierra una trilogía.

Con la precondición de que el lector sabe programar en ISO C++, pero ya pasando a un nivel medio, su objetivo pedagógico es enseñar las buenas prácticas básicas que debe usar el programador novato de C++ para lograr crear clases correctas.

El texto despliega catorce temas: el primero se dedica a presentar el sistema de desarrollo, con un estilo de programación típica de ANSI C, pero conservando la claridad de lectura humana del código. El segundo tema revisa brevemente la filosofía de aplicación de la OOP y el tercero delinea una muy breve historia de la misma. El cuarto revisa la jerga de los programadores de OOP y como se ve en C++, y el quinto introduce el uso los espacios de nombres.

El sexto tema va a la construcción correcta de una clase y el séptimo produce una clase compuesta a partir de la anterior. El octavo tema crea una clase utilitaria y el noveno enseña cómo hacer la sobrecarga de operadores. Luego viene el tema diez de la herencia dividido en cuatro partes: la herencia simple, el polimorfismo, la herencia múltiple y el polimorfismo junto a la herencia múltiple. Cierra el texto el tema once dividido en dos partes: la programación genérica de plantillas de clases y la programación genérica de functores.

Lleva cinco apéndices que son: (1) Las organizaciones normativas de C++; (2) Las bibliotecas BOOST, Blitz++ y STLport; (3) Un fichero **`makefile`** y cómo se compila desde la línea de comandos del DOS; (4) Métodos y Miembros estáticos, que tiene dos programas auxiliares; y (5) Manejo de errores, el cual a su vez tiene un programa auxiliar.

El texto se complementa con 12 ilustraciones aclaratorias, 10 tablas para consultas, 14 programas resueltos y concluye con 15 ejercicios propuestos. Se adjuntan los programas y las soluciones a los ejercicios planteados. Para cualquiera de ellos casi siempre hay más de una solución, mejor o al menos diferente. Ante la duda el lector debe buscar una personal, que es de las mejores formas de aprender alegre y eficazmente: ¡mediante la experimentación propia!

Se puede contactar al autor por e-mail en mailto: simongv@infomed.sld.cu?subject=texto3 de Cpp para cualquier queja, consulta o sugerencia.

¡Buena suerte y muchas gracias!