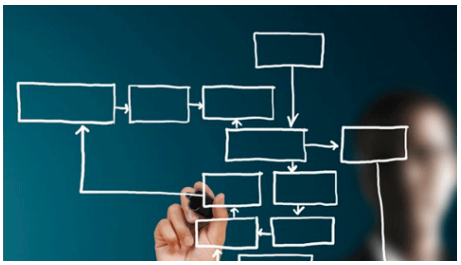
PABLO R. RAMIS

DISEÑO ORIENTADO A OBJETOS

[April 8, 2024 at 13:47 – classicthesis version 1.0]

DISEÑO ORIENTADO A OBJETOS

PABLO R. RAMIS



Una introducción

Departamento de Informática
Instituto Politécnico Superior
Universidad Nacional de Rosario

Mayo 2019 – version 1.0

Hablar es barato.
Enséñame el código.

— Linus Torvalds

La educación en computación no puede hacer a nadie un experto programador más que el estudio de pinceles y pigmentos puede hacer a alguien un pintor experto.

— Eric S. Raymond

RESUMEN

Este trabajo se recopila el material generado para la cátedra de Diseño Orientado a Objetos de la carrera Analista Universitario en Sistemas del Instituto Politécnico Superior.

La cátedra la cual lleva casi 15 años de dictado ha ido cambiando, evolucionando hasta quedar como se encuentra plasmado aquí.

Se verá un balance entre practica y teoría, entre el diseño y la programación. Se busco incorporar ejemplos en C++ y no solo en Java como fue originalmente.

LICENCIA

PUBLICACIÓN

Este trabajo fue realizado en LaTeX. Fue editado en texmaker http://www.xm1math.net/texmaker/download.html

Atención: se utilizó la plantilla `ClassicThesis`. Puede ser descargada en https://www.ctan.org/tex-archive/macros/latex/contrib/classicthesis/.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

UML Unified Modeling Language

OMG Object Management Group

Part I

CONCEPTOS FUNDAMENTALES

1 INTRODUCCIÓN

Ei choro aeterno antiopam mea, labitur bonorum pri no. His no decore nemore graecis. In eos meis nominavi, liber soluta vim cu. Sea commune suavitate interpretaris eu, vix eu libris efficiantur.

1.1 EL MODELO DE OBJETOS

Los principios básicos del modelo de objeto son: Abstracción, Encapsulamiento, Modularidad, Jerarquía, tipos, concurrencia y persistencia.

De la filosofía planteada por el modelo de objetos se desprenden el diseño y la programación orientada a objetos.

El diseño orientado a objetos representa un desarrollo evolutivo, no una revolución. No rompe con el pasado, sino que se basa en avances ya probados.

1.1.1 *Programación Orientada a Objetos*

La programación orientada a objetos es un método de implementación en el que los programas se organizan como colecciones cooperativas de objetivos, cada uno de los cuales representa una instancias de alguna clase, y cuyas clases son, todos ellas, miembros de una jerarquía de clases unidas mediante relaciones de herencia.

En resumen:

- Utiliza objetos en vez de algoritmos para sus bloques lógicos de construcción fundamentales
- Cada objeto es una instancia de alguna clase
- Las clases están relacionadas con otras clases por medio de relaciones de herencia.

La programación puede parecer orientada a objetos, pero si falta cualquiera de estos elementos ya no lo es. Sería simplemente un programa que utiliza tipos de datos abstractos.

Por lo tanto un lenguaje orientado a objetos es aquel que soporta los siguientes requisitos:

- Soporta objetos que son abstracciones de datos con una interfaz de operaciones con nombre y un estado local oculto.
- Los objetos tienen un tipo asociado (clase)

- Los tipos pueden heredar atributos de los supertipos

Durante el curso, trabajaremos con C++, aunque no será el único lenguaje con el cuál veremos ejemplos ya que en algunas situaciones podremos ver que hay ejemplos en ***java*** por adecuarse mejor al tema que se trate.

1.1.2 *Diseño Orientado a Objetos*

Es un método de diseño que abarca el proceso de descomposición orientada a objetos y una notación para describir los modelos lógicos y físicos, así como los modelos estático y dinámico del sistema que se diseña.

Cabe resaltar dos cosas: El poder descomponer al sistema y la utilización de notaciones.

Como notación usaremos Unified Modeling Language (UML) - Lenguaje de Modelado Unificado.

1.1.3 *Análisis Orientado a Objetos*

El análisis orientado a objetos es un método de análisis que examina los requisitos desde la perspectiva de las clases y objetos que se encuentran en el vocabulario del dominio del problema.

Generalmente, los productos del Análisis Orientado a Objetos sirven como punto de partida para realizar el Diseño Orientado a Objetos, los productos de dicho diseño pueden utilizarse como anteproyecto para la implementación completa de un sistema utilizando métodos de programación orientada a objetos.

1.2 ELEMENTOS DEL MODELO OBJETO

Elementos fundamentales:

- Abstracción
- Encapsulamiento
- Modularidad
- Jerarquía

Estos son los rasgos distintivos, hay otros tres que podrían considerarse secundarios:

- Tipos (tipificación)
- Concurrencia
- Persistencia

1.2.1 *Abstracción*

Una abstracción denota las características esenciales de un objeto que lo distinguen de todos los demás tipos de objeto y proporciona así fronteras conceptuales nítidamente definidas a la perspectiva del observador.

Se centra en la visión externa de un objeto, y por tanto sirve para separar el comportamiento esencial de un objeto de su implantación.

En resumidas palabras, conozco al objeto, se que hace o para que sirve pero no profundizo en como lo hace o funciona.

Se persigue construir abstracciones de entidades, porque imitan directamente el vocabulario de un determinado dominio del problema.

1.2.2 *Encapsulamiento*

Es el proceso de almacenar en un mismo compartimiento los elementos de una abstracción que constituyen su estructura y su comportamiento; sirve para separar el interfaz contractual de una abstracción y su implementación.

El interfaz de una clase captura sólo su vista externa, abarcando la abstracción que se ha hecho del comportamiento común de todas las instancias de la clase. La implementación de una clase comprende la representación de la abstracción así como los mecanismos que consiguen el comportamiento deseado. La interfaz de una clase es el único lugar en el que se declaran todas las suposiciones que un cliente puede hacer acerca de todas las instancias de la clase; la implantación encapsula detalles acerca de los cuales ningún cliente puede realizar suposiciones.

Hay autores que han llamado a esos elementos encapsulados los *secretos de la abstracción*.

En conclusión, la abstracción y el encapsulamiento son conceptos complementarios: la abstracción se centra en el comportamiento observable de un objeto, mientras el encapsulamiento se centra en su implementación que da lugar a este comportamiento. El encapsulamiento se consigue a menudo mediante la ocultación de información, que es el proceso de ocultar todos los secretos de un objeto que no contribuyen a sus características esenciales, típicamente la estructura de un objeto está oculta, así como la implantación de sus métodos.

1.2.3 *Modularidad*

La modularidad es la propiedad que tiene un sistema que ha sido descompuesto en un conjunto de módulos cohesivos y débilmente acoplados.

Es el acto de fragmentar un programa en componentes para reducir su complejidad en algún grado. De esta forma se generan fronteras bien definidas y documentadas del sistema.

Encontrar las clases y objetos correctos y organizarlos después en módulos separados con decisiones de diseño.

Más adelante veremos en detalle características de esta particularidad.

1.2.4 *Jerarquía*

Es una clasificación u ordenación de las abstracciones.

Las dos jerarquías mas importantes son su estructura de clases (jerarquía de clases) y su estructura de objetos (jerarquía "de partes").

Estos conceptos son los que define cuando un objeto ***es un*** en caso del primero o ***es parte de*** en el segundo.

1.2.5 *Tipos (tipificación)*

Los tipos son la puesta en vigor de la clase de objetos, de modo que los objetos de tipos distintos no pueden intercambiarse o, como mucho, pueden intercambiarse sólo de formas muy restrictivas.

El significado de tipos deriva de la teoría sobre tipos de datos abstractos. Un tipo es una caracterización precisa de propiedades estructurales o de comportamiento que comparten una serie de entidades. Para nuestro uso general, los términos tipo y clase son iguales.

Aquí además del diseño, influye mucho el lenguaje con el que se trabajará y la comprobación de tipos que este realice, puede ser estricta o débil o incluso no tener tipos. Esta última, es la más flexible pero a su vez la de mayor riesgo, ya que no puede conocerse si hay incongruencias de tipo hasta el momento de su ejecución. La comprobación estricta de tipos permite utilizar el lenguaje de programación para imponer ciertas decisiones a nivel de diseño, esto comienza a tener mucha importancia en la medida que el sistema es complejo y de gran tamaño. La desventaja que se puede sufrir es que produce una dependencia semántica muy importantes al punto que el cambio de una interfaz en una clase base de la que heredan otras, provocaría la recompilación de todas las subclases.

Ventajas del uso de lenguajes fuertemente tipados:

- Sin la comprobación de tipos, un programa puede "estallar" en plena ejecución
- En la mayoría de los sistemas, el ciclo editar-compilar-depurar es tedioso y conviene la detección de errores en forma temprana.

- La declaración de tipos ayuda a documentar mejor los programas.
- En general, se genera un código mas eficiente si se declaran los tipos.

De la teoría de tipos se desprende un concepto muy importante, el polimorfismo, en el que un solo nombre puede denotar objetos de muchas clases diferentes que se relacionan por alguna superclase común. Ej: Se posee un objeto TransporteDeCarga, este será clase base de los subtipos de transportes del sistema: Camión, Tren y Avión. TransporteDeCarga posee una función que es llenarBodega(), por lo tanto, los tres subtipos también disponen de esa funcionalidad por heredarla pero como cada uno con características diferentes, o sea, dicha función se comportará de manera distinta en cada uno, y en caso que alguno de esos vehículos tenga la necesidad de poseer distintos modos de cargas, por ejemplo dependiendo el material a guardar el modo de llenado es otro, pueden especificarse estas formas siempre invocando a llenarBodega(materialLiquido) o llenarBodega(materialSólido).

1.2.6 *Concurrencia*

La concurrencia se centra en la abstracción de los procesos y la sincronización de los mismos. Permite a diferentes objetos actuar al mismo tiempo.

Podemos ver tres enfoques de concurrencia dentro del modelo de objetos:

- La concurrencia es una característica de ciertos lenguajes de programación, por ejemplo: en Ada para expresar un proceso concurrente se utiliza task, en Smalltalk se posee la clase Process.
- Se puede usar una biblioteca de clases que soport alguna forma de procesos ligeros, es el enfoque usado por C++ con los threads.
- Pueden usarse interrupciones para dar la ilusión de concurrencia, esto exige mucho conocimiento de ciertos detalles a nivel de hardware.

1.2.7 *Persistencia*

Es la propiedad de un objeto por la que su existencia trasciende el tiempo (es decir, el objeto existiendo después de que su creador deja de existir) y/o el espacio (es decir, la posición del objeto varía con respecto al espacio de direcciones en el que fue creado).

Generalmente la persistencia de los datos se logra con el uso de Bases de Datos, pero también se posee y se necesita que variables y

datos de las mismas se conserven para la comunicación de los distintos procesos. Cada lenguaje tendrá una forma de implementar esto: variables propias, globales, elementos del montículo (heaps), etc.

Un enfoque moderno y razonable para la persistencia es proporcionar una capa (en la arquitectura de la implementación del proyecto) orientada a objetos bajo la cual se oculta una base de datos relacional, ejemplo: el uso del frameworks Hibernate.

2

ELEMENTOS BÁSICOS Y CONCEPTOS

ntoSi bien, dependiendo el autor y el contexto (en ocaciones temporal) las definiciones y conceptos pueden ser muchos y variados. Sin embargo, todos estarán dentro del mismo sentido general.

Veremos y analizaremos a varios de ellos, buscaremos ser objetivos en los significados para que la idea del mundo de objetos quede definida.

Antes de comenzar con el significado de un ***objeto*** propiamente dicho, introduciremos otro el cual nos resulta fundamental:

CONCEPTO O TIPO DE OBJETO: Es una idea o noción que nosotros aplicamos a las cosas u objetos en nuestro conocimiento. Es por lo cual reconocemos a los objetos.

La existencia de conceptos lleva a la existencia de un conjunto de objetos que lo satisfacen o lo componen:

ESCRITORES	POLÍTICOS	FAMOSOS	MÚSICOS	P.PERFECTA
Borges	Menem	Borges	McCartney	
Cortazar	Alfonsín	Cortazar		
Gorodicher	Macri	McCartney		
	Kirschner	Alfonsin		
		Macri		
		Kirschner		

Table 1: Conceptos y objetos que responden a ellos.

De un universo de objetos, tenemos tipos de objetos que los agrupan, esto no implica que un objeto no pueda responder a varios tipos, o que existan tipos que no tienen un objeto que lo satisface (podría existir en un futuro Table 1).

En el momento de analizar un sistema, habrá que plantearse, definir, o encontrar los conceptos que existan para dicho sistema y a partir de esto, agrupar a los objetos que existen en ellos. Esto nos ayudará a comprender el modelo que intentamos construir.

Como vemos en Figure 1, además de tener el concepto en si mismo, tenemos una definición, una idea o comprensión del mismo, y además, a los objetos que lo componen.

Para representar a los conceptos, utilizaremos como símbolo a rectángulos, y dentro de ellos estará el nombre del concepto Figure 2.

De las tres partes mencionadas arriba, podría ocurrir que:

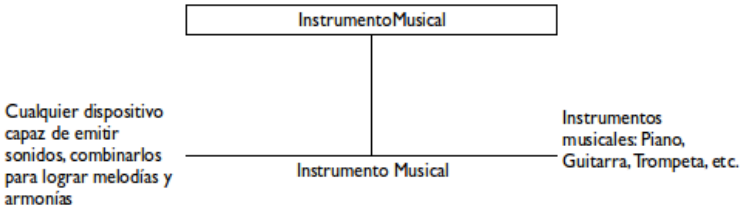


Figure 1: Representaciòn del Concepto. Definido por extensión y comprensión

Figure 2: Concepto

1. Nos falte el nombre. O sea, tengamos la definición, y los objetos que la integran pero no un nombre significativo que lo represente.
2. Nos falte la definición. Tenemos el nombre del concepto, los objetos que responden a el, pero no una definición o idea de su significado.
3. Nos falten los objetos. Tenemos el nombre y la definición del concepto pero ningún objeto existente lo satisface.

De estas tres opciones mencionadas, solo la última es válida. Las dos anteriores son errores o falta de análisis que puede resolverse con mejores consultas a las personas que posean el conocimiento del sistema que se está analizando.

SINÓNIMOS O ALIAS: Es cuando un tipo de objeto puede tener dos nombres diferentes, o sea responden a la misma definición y por lo tanto comparten el conjunto de objetos. Figure 3

HOMÓNIMOS DE TIPOS DE OBJETOS: Cuando un concepto puede estar definido de dos formas diferentes. Esto implica que posee dos conjuntos posibles de objetos que responden a el, uno por cada definición. Figure 4

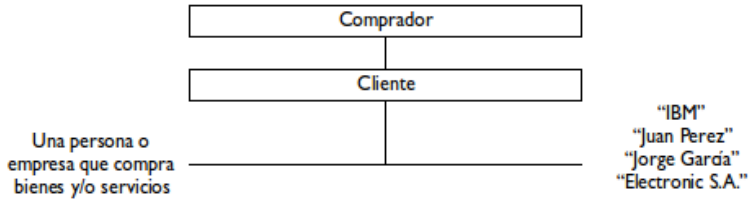


Figure 3: Sinònimo o Alias

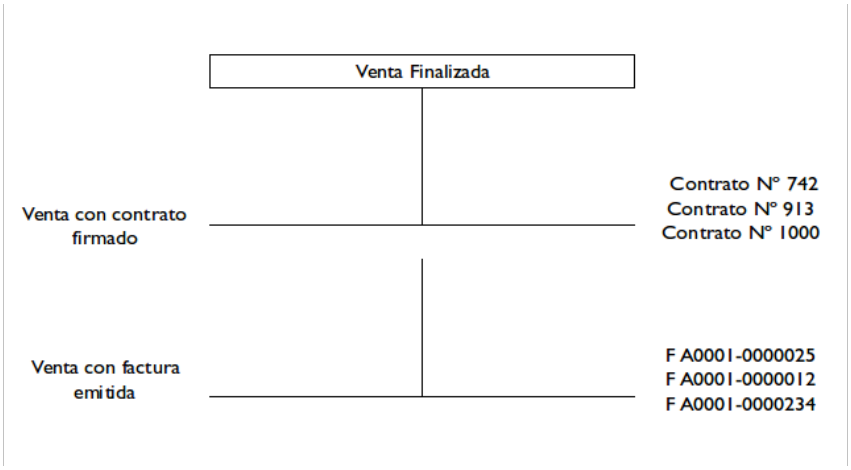


Figure 4: Homonimos

2.1 OBJETOS

Son muchas las definiciones que se pueden dar de la orientación a objetos o de los objetos en si mismos. Por ejemplo:

> Un sistema construido con métodos orientados a objetos es uno cuyos componentes son bloques encapsulados de datos y funciones, que pueden heredar atributos y comportamiento de otros componentes de este tipo, y cuyos componentes se comunican a través de mensajes entre sí.

En la definición encontramos conceptos que iremos viendo con mucho mas detalle, como encapsulamiento, herencia, etc. Antes, repasaremos varios de diferentes autores:

OBJECT Una abstracción de algo en un dominio de problema, que refleja las capacidades del sistema para mantener información sobre él, interactuar con él o ambos; Una encapsulación de los valores de los atributos y sus servicios exclusivos (sinónimo de una instancia)[Coad and Yourdon:1990] [2]

CLASE Una colección de uno o mas Objetos con un conjunto uniforme de atributos y servicios incluyendo la descripción de como crear nuevos Objetos de esa Clase. [Coad and Yourdon:1990] [2]

El consorcio de vendedores de software Object Management Group (OMG) nos da la siguiente definición:

UN OBJETO ES UNA COSA. Este es creado como la instancia de un tipo de objeto. Cada objeto tiene una identidad única que lo distingue independientemente de sus características. Cada objeto ofrece una o mas operaciones. [OMG:1992] [7]

Otra que podemos encontrar es:

OBJETO se define como una abstracción del software que modela todo aspecto relevante de una simple entidad tangible o conceptual o cosa de un dominio de aplicación o espacio de solución. Un objeto es una de las entidades de software primarias en una aplicación orientada a objetos, típicamente corresponde a un módulo del software, y consiste en un conjunto de atributos relacionados, atributos, mensajes, operacione y objetos componentes opcionales. [Firesmith:1996] [3]

Un referente de la orientación a objetos nos da otra definición muy particular:

OBJETO SEGÚN PERSPECTIVA DEL CONOCIMIENTO HUMANO es cualquier cosa que cumple con:

1. Una cosa tangible y/o visible.
2. Algo que puede ser aprehendido intelectualmente.
3. Algo hacia el cual se dirige el pensamiento o la acción.

Un objeto tiene un estado, comportamiento e identidad; la estructura y el comportamiento de los objetos similares están definidos en una clase común; el término instancia y objeto son intercambiables. [Booch:1991] [1]

Shlaer y Mellor ofrecen una explicación similar:

OBJETOS es una abstracción de un conjunto de cosas tal que:

1. Todos los objetos del conjunto (Instancias) tienen las mismas características y,
2. Todas las instancias están sujetas y se ajustan al mismo conjunto de reglas y políticas.

Ver: [Shlaer and Mellor:1992] [8]

Ivar Jacobson, otro referente de la talla de Booch, junto a otros colaboradores nos dan una concisa definición:

OBJETO esta caracterizado por un número de operaciones y un estado que recuerda los efectos de dichas operaciones. [Jacobson et al.:1992] [4]

Finalmente, James Martin y James Odell ofrecen estas dos:

OBJETO esta caracterizado por un número de operaciones y un estado que recuerda los efectos de dichas operaciones. [Martin and Odell:1992] [5]

OBJETO es toda cosa que responde a un ***concepto o tipo de objeto*** [Martin and Odell:1994] [6]

En definitiva es *cualquier cosa* que "pueda ser pensada en", "referida a", "descripta por".

Hay más autores y todos tienen una variante o posible definición.

Sigue siendo importante el notar que todos están observando similitudes, en mayor o menor detalles. Hay dos puntos en este momento que si vale la pena resaltar:

1. Un ***objeto*** es una instancia de un ***concepto***
2. Todo ***concepto*** puede ser ***objeto***

Veremos que ***Instancia*** será utilizada en varias ocaciones y dependerá del contexto no siempre significará lo mismo. Ocurre lo mismo con ***Objeto*** y ***Clase***.

El concepto de ***Objeto*** y el de ***Clase*** están ligados y bajo un concepto amplio, pueden ser considerados sinónimos.

La representación de la ***clase*** es la siguiente:

*Nota: Si bien, la representación mas clásica y común del un objeto dentro de los lenguajes de programación es la clase, en algunos lenguajes otras estructuras también son objetos. Por ejemplo, en C++ los **namespace** y las **struct** son también objetos.*

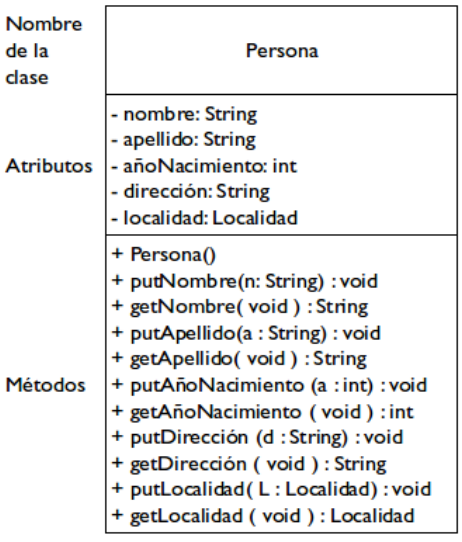


Figure 5: clase

La representación en código la vemos en Listing 1

Listing 1: Declaración de una clase

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

class Persona{
    private:

        string nombre, apellido, direccion;
        int aNacimiento;
        Localidad localidad;

    public:

        Persona();
        void putNombre(string);
        string getNombre();
        void putApellido(string);
        string getNombre();
        void putDireccion(string);
        string getDireccion();
        void setANacimiento(int);
        int getANacimiento();
        void setLocalidad(Localidad);
        Localidad getLocalidad();
}
```

Suele ocurrir que se encuentren diferencias en algunos términos según las bibliografías usadas o las costumbres. Por ejemplo, en una visión exclusivamente de código, del lenguaje, es común ver a la clase como "la declaración del objeto" en una visión estática del mismo, y a su instancia, o sea al objeto creado en memoria como al objeto en forma dinámica.

2.2 RELACIONES

Son "vínculos" entre objetos, El vinculo esta representado por una línea que une a dos objetos, dependiendo el tipo de asociación, los extremos de dicha línea finalizarán de distintos modos, flechas u otro gráfico.

Veremos que existen distinto tipos de relaciones entre objetes, La asociación, la generalización y la dependencia.

2.2.1 *Asociación*

Se utiliza para especificar que un conjunto de objetos esta conectado con otros objetos. Dada la asociación entre dos clases, se puede establecer la relación entre dos dos objetos (instancias) de ambas.

Una relación binaria se compone de dos extremos, los cuales pueden ser los mismos, o sea, coincidir en la misma clase.

Figure 6: Asociación

Aquí vemos que el alumno está asociado de algún modo con la materia, cuando existan dos objetos que correspondan a cada uno de esos conceptos, ese vinculo puede formarse y generar a una tupla, ej: (Martín, Análisis Matemático) que representa a esa asociación, la cual puede ser, "cursa"

En algunas ocasiones, la relación posee un concepto lo suficientemente importante como para que deba ser considerada un objeto también, por ejemplo, en una relación laboral, entre un empleado y su empresa, existe un contrato de trabajo, el cual posee atributos propios que deben ser representados, esto hace que la relación este representada por el objeto "Contrato".

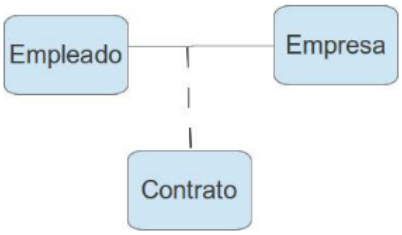


Figure 7: Objeto relación

A este objeto, a esta nueva clase la llamaremos ***Clase Relación***

2.2.2 *Adornos*

2.2.2.1 *Nombre y Dirección*

Para una mejor lectura y modelización, las asociaciones pueden tener un nombre para describir la naturaleza de la relación y una dirección para eliminar la interpretación ambigua en caso que sea necesario.

2.2.2.2 *Rol*

Es un adorno que indica el papel que juega la clase en la asociación. La clase puede jugar el mismo o diferentes roles en otras asociaciones. Se vuelve importante el explicitar un rol cuando existen varias asociaciones entre los mismos objetos Figure 8.

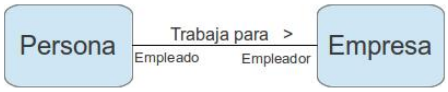


Figure 8: Rol

El uso del rol y la dirección no es meramente decorativo, puede llegar a tener mucha importancia cuando los objetos tienen múltiples relaciones con los mismos tipos Figure 9.

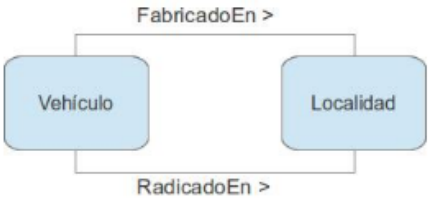


Figure 9: Roles y dirección

Vemos como Vehículo tiene dos referencias al mismo tipo de objeto, Localidad. Gracias a la dirección queda mucho más claro el motivo de esta doble relación.

El código desarrollado para esto podría verse como en Listing 2.

Otra posible situación podría ser Figure 10

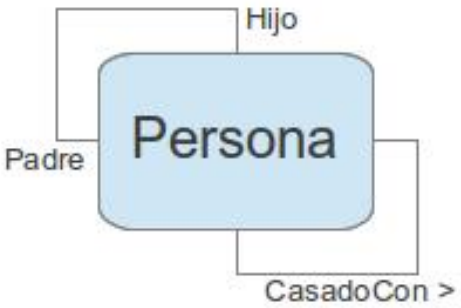


Figure 10: Relaciones Humanas

Aquí vemos otra forma de plantear la relación entre objetos, en este caso, del mismo tipo. Es algo muy común sobre todo cuando se modelizan relaciones humanas

Listing 2: Relación entre clases

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

class Localidad{

    private:
        string codigoPostal;
        string nombre;

     // Aqui sigue el resto de las declaraciones
}

class Vehiculo {
    private:
        Localidad fabricadoEn;
        Localidad radicadoEn;

     // Aqui sigue el resto de las declaraciones
}
```

2.2.2.3 *Multiplicidad o Cardinalidad*

Con este adorno se indica cuantos objetos pueden conectarse a través de una instancia de asociación. Se expresa con un número entero o un rango de enteros. Cuando no se explicita, se asume que es de cero a muchos Table 2.

POSIBLES EXTREMOS	POSIBLES EXTREMOS
1 (uno)	2...5 (de dos a cinco)
1...* (uno o más)	* (muchos)
0...1 (cero o uno)	3 (hasta 3)
0...3 (cero a tres)	

Table 2: Relaciones con canitdad de objetos.

Como vemos en Figure 11, el gráfico indica que la persona trabajará para 1 y como máximo 5 Empresas que serán sus Empleadores, y la Empresa podrá tener 1 o muchos Empleados

Este adorno es fundamental en el momento del diseño, la omisión o error en el mismo provocará que el desarrollo del código o del modelo de datos (la estructura de la base de datos) se realice con error y por lo tanto que el sistema no cumpla con los requisitos impuestos.

Los ejemplos **??** muestran como se plantean las relaciones tanto en el código como en el modelo de datos. Analizando el diseño el mapeo

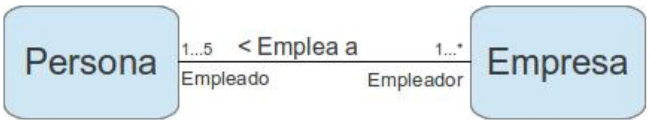


Figure 11: Cardinalidad

que se realiza esta centrado sobre la empresa que emplea a personas (según la dirección que se indica).

En estos ejemplos, aunque incompletos vemos como se establece la relación entre los objetos. La clase Empresa tiene un contenedor para guardar a los objetos Empleados que vaya contratando.

Debe tenerse en cuenta que este no es el único modo de modelizar, todo depende de la interpretación del diseño y en que se acentúa, si en la empresa o en el empleado. Si fuera el segundo caso, veriamos a la Empresa como información del Empleado, en vez de declarar el *vector* donde se encuentra, en los atributos del Empleado podríamos encontrar dentro del bloque *private:* el siguiente código Listing 4

De este modo se define un arreglo para ubicar las 5 posibles empresas en las que puede trabajar indicadas en la cardinalidad del Empleado, si así fuera indicada en el diseño.

Si vemos un posible diseño de la base de datos, la estructura será diferente. Como se ve en Figure 12, la relación se materializa en una tercera tabla a través de las claves primarias de Persona y Empresa. Cabe la mención que las bases de datos relacionales no son orientadas a objetos pero será la situación más común el que tengan que convivir ambos diseños.

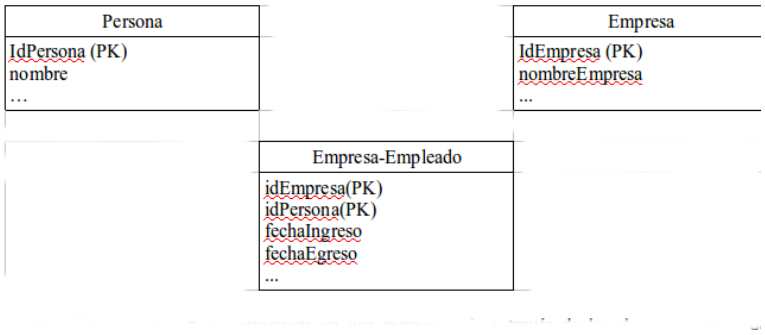


Figure 12: Normalizaciòn de tablas

2.2.3 *Agregación*

Es un caso especial de asociación, las asociaciones normales indican la relación entre dos clases de la misma jerarquía, En la agregación

Listing 3: Relación entre clases

```
#include<iostream>
#include<string>
#include<vector>
using namespace std;

class Persona {

    private:
    //atributos
    string idPersona;
    string nombre;
    //aqui continuan las declaraciones

    public:
    //metodos
        Persona (string, string);
    //continuan los metodos

}

class Empresa {

   private:
   //atributos
       vector <Empleado> empleados;
   //continua la definicion de atributos

   public:
   //metodos
   //Lista de metodos y constructores necesarios para manipular
       al objeto

}
```

una de las clases tendrá el papel del "todo" y la otra el papel de "parte".

Resumiendo, esta asociación se utiliza para modelizar objetos complejos que están formados por "partes" las cuales son otros objetos como se ve en Figure 13.

Vale mencionar que los adornos indicados antes, principalmente la cardinalidad son validos también en este tipo de asociación.

Un ejemplo más completo es el que vemos en Figure 14

El modelo que se realiza dependerá del grado de detalle que se pretende alcanzar. Por ese motivo podrán existir mas o menos cantidades de clases componentes.

Listing 4: Relación entre clases

```
class Empleado{
    private:
        Empresa empresas[5];
    //resto del codigo.
}
```

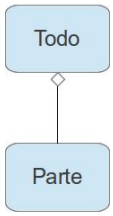


Figure 13: Agregación

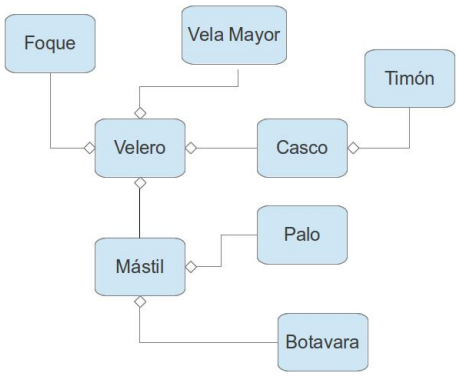


Figure 14: Partes componentes de un velero

2.2.4 *Composición*

El hecho de usar estas relaciones, tanto la agregación como la composición, no implica el no usar el resto de los adornos vistos. Es más, es recomendable que además de indicar estas relaciones cuando corresponden, se las complete con los roles y la cardinalidad principalmente.

Es una variante de la agregación, con una fuerte relación de pertenencia y vida coincidentes de las partes con el todo. Esto significa que un objeto puede formar parte de sólo un "Todo" compuesto a la vez y que ese todo debe su existencia a que posee esa parte como se ve en el ejemplo de la Figure 15.

2.3 CLASIFICACIÓN

Es el acto que resulta de aplicar conceptos (tipos de objetos) a un objeto coo se puede ver en la Figure 16

Podemos decir que es un mecanismo que permite manejar, ordenar a la complejidad del mundo.

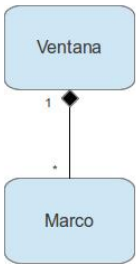


Figure 15: Partes que componen la ventana

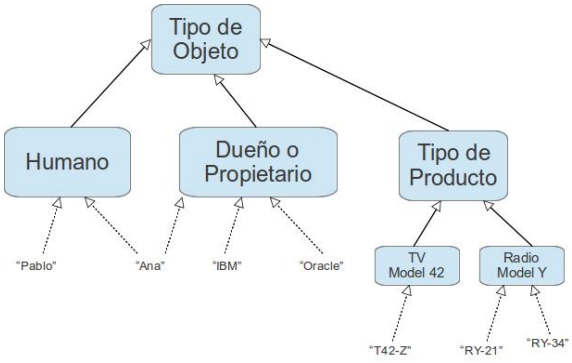


Figure 16: Jerarquia de conceptos clasificados

2.3.1 *Herencia - Generalización*

Es el acto o resultado de distinguir un concepto que se encuentra incluido otro como se ejemplifica en Figure 17, es una forma particular de ver a la *clasifiación* enunciada antes.

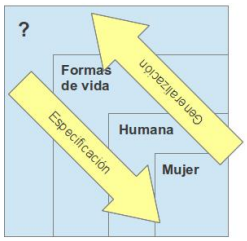


Figure 17: Generalización y Especifiación

Opuesto a la ***generalización*** es la ***especificación*** de la cual se parte de un concepto más general hasta llegar a otros que forman parte de ese tipo de objeto.

De este concepto se desprende dos términos importantes:

Subtipo : refiere a un tipo de objeto que:

- El conjunto de miembros están todos incluidos dentro de otro conjunto.
- La definición es más específica que otra.

Supertipo : se refiere a un tipo de objeto que:

- El conjunto incluye a todos los miembros de uno o mas conjuntos.
- La definición es más general que otras.

Cuando hablamos que un objeto hereda de otro, que es subtipo de otro, o hijo de otro, estamos diciendo que ese objeto "es un" objeto del tipo del padre, o del supertipo. Esto implica que posee todas las características del objeto padre más las suyas propias particulares.

Como vemos tenemos nombres que son sinónimos para los objetos de nivel superior en la generalización o la herencia podemos usar *padre*, *supertipo*; para los objetos de niveles inferiores o que heredan de uno superior: *hijo* o *subtipo*.

En el gráfico del ejemplo de clasificación, Figure 16, podemos notar que el objeto instanciado ***Ana*** hereda de ***Humano*** y de ***Dueño o Propietario***. Esto significa que ***Ana*** **ES** un objeto de tipo ***Humano*** y de tipo ***Dueño o Propietario***.

2.3.1.1 *Particiones*

Una partición es una división de un tipo de objetos en subtipos. Ejemplo: Figure 18

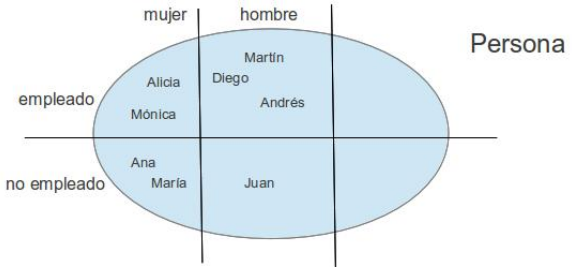


Figure 18: Partición

Entre particiones podrían existir superposiciones pero dentro de una partición no pueden solaparse los objetos, deben ser exclusivos. Ejemplo: Figure 19

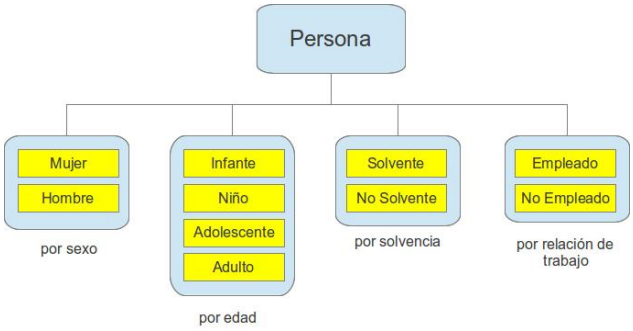


Figure 19: Posibles particiones de un objeto

Con esto, lo que se pretende decir es que la *persona* que este en la partición *Por Solvencia* o es *Solvente* o es *No Solvente* pero no puede ser ambos.

2.3.1.2 *Particiones completas*

Una partición es completa si tiene todos los subtipos especificados como se ve en la Figure 20.

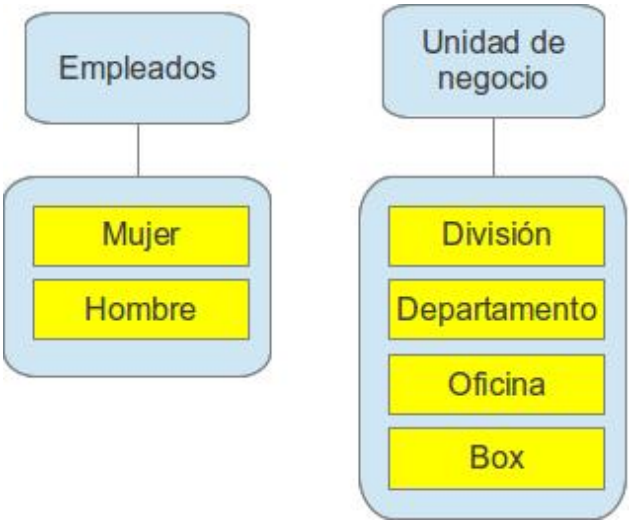


Figure 20: Particiones completas de objetos

Las particiones completas especifican una clasificación obligatoria: El empleado es hombre o mujer; no ambas.

2.3.1.3 *Particiones incompletas*

Una participación incompleta es aquella que tiene una lista parcial de los subtipos que abarca.

No es tan extraño el que se utilicen igual, teniendo en cuenta que uno modeliza solo lo fundamental o importante para el sistema.

2.4 ATRIBUTOS, ESTADO, EVENTOS Y MÉTODOS

2.4.1 *Atributos*

Un atributo es un valor de datos lógico de un objeto. Se incluirán solo aquellos que que los requisitos sugieran o implican una necesidad de registrar la información.

En la Figure 21 vemos un ejemplo de notación en UML.

Los atributos poseen las siguientes características:

visibilidad / nombre: *tipo* [multiplicidad] = valor por defecto propiedadesyconstraints

La *visibilidad* puede ser +, -, #, y significan *public, private, protected o package* respectivamente.

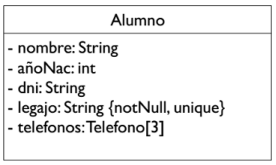


Figure 21: Atributos de la clase Alumno

/ indica que el atributo es derivado. Esto significa que es un atributo que puede ser calculado de otros atributos de la clase.

Nombre: es el nombre o frase corta con la que se nombra al atributo. Comienza en minúscula y en caso de ser varias palabras se separan con la mayúscula correspondientes sin espacios ni guiones.

Tipo: es el tipo de dato al que pertenece el atributo, puede ser una clase, una interface, o un tipo de dato primitivo.

Multiplicidad: especifica cuantas instancias de ese tipo de atributo referenciado.

Valor por defecto: es el valor que tomará el atributo en caso que no se complete en forma manual.

propiedades y constraints: son aquellas que el atributo debe cumplir una vez instanciado el objeto, ejemplo: no puede ser nulo, o debe ser mayor a 0, debe ser constante.

En el modelo conceptual es conveniente mantener los atributos de forma simple y utilizar la asociación para indicar a los atributos complejos y no es imprescindibles que se expliciten como se muestra en la . Figure 22.

Figure 22: Atributos complejos

2.4.1.1 *La diferencia con la implementación*

El último ejemplo se centra en la descripción conceptual del dominio, del modelo de negocio. Esto no implica que en la visión de la implementación haya diferencias o sea conveniente la descripción completa y detallada. Hay que ser conciente que esto tambien podrá variar de acuerdo al lenguaje que se utilice y las herramientas que este brinde.

2.4.1.2 *Tipos de datos*

Podemos diferenciar dos tipos de datos esencialmente, los primitivos y los no primitivos (objetos). Los primeros son aquellos que no son objetos, no poseen métodos propios para trabajarlos no es necesario instanciarlos con metodos constructores, más adelante veremos este destalle para los objetos.. En C++ dichos tipos primitivos son:

bool : tiene un tamaño de 8 bits y un rango entre 0 y 1, en pocas palabras es cero o es uno (falso o verdadero). Este tipo de dato, es comúnmente usado en condicionales o variables que solo pueden tomar el valor de falso o verdadero. Las variables de tipo bool no suelen llevar modificadores, pues son innecesarios, ya que su rango es solo 0 y 1.

int : El tipo de dato int, tiene un tamaño de 32 bits y un rango entre -2.147.483.648 y 2.147.483.647. Este tipo de dato, es usado para números enteros (sin cifras decimales). A continuación alguna combinaciones con los modificadores:

short int : Tiene un tamaño de 16 bits y un rango entre -32.768 y 32.767.

unsigned short int : Tiene un tamaño de 16 bits y un rango entre 0 y 65535.

unsigned int : iene un tamaño de 32 bits y un rango entre 0 y 4.294.967.295.

long long int : Tiene un tamaño de 64 bits y un rango entre -9.223.372.775.808 y 9.223.375.775.807.

unsigned long long int : Tiene un tamaño de 64 bits y un rango entre 0 y 2exp64.

float : tiene un tamaño de 32 bits, es usado comúnmente en números con 6 o menos cifras decimales. Tiene un rango entre $1,17549 * (e^{-}38)$ hasta $3,40282 * (e^{+}38)$.

double : tiene un tamaño de 64 bits, es usado para números de menos de 15 cifras decimales. Tiene un rango entre $2,22507 * (e^{-}308)$ hasta $1,79769 * (e^{3}08)$.

long double : Tiene un tamaño de 96 bits y una precisión de 18 cifras decimales. Tiene un rango entre $3,3621 * (e^{-}4932)$ hasta $1,18973 * (e^{4}932)$.

char : es simplemente un caracter. la forma mas simple de declaración es char nombre = 'valor' notar que va entre comillas simples.

Todos los tipos de datos pueden ser declarados como vectores o arrays, esto se logra dando un tamaño máximo entre corchetes luego del nombre de la variable de forma idéntica como se realiza en C. Esto no se justifica en caso de los caracteres ya que C++ en su librería *string.h* posee el tipo *string* con una serie de métodos que facilitan mucho su uso.

Otro modificador importante es el que define a la variable como *constante*

Declarado de esta forma el valor no puede ser cambiado, mientras el programa esté en ejecución el valor de dicha variable no puede alterarse por lo tanto no podría hacerse lo siguiente:

En otros lenguajes estos tipos de datos y modificadores pueden llegar a variar.

Listing 5: tipo constante

```
#include<iostream>

using namespace std;

int main()
{
    const float PI  = 3.1416; //Definimos una constante llamada
        PI
    cout << "Mostrando el valor de PI: " << PI << endl;

    return 0;
}
```

Listing 6: tipo constante - uso erroneo

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

int main()
{
    const float PI  = 3.1416; //Definimos una constante llamada
        PI
    cout << "Mostrando el valor de PI: " << PI << endl;
    PI = 2; //Esto generara un error pues PI es de solo lectura (
        constante)
    return 0;
}
```

2.4.1.3 *Visibilidad de los atributos*

En el diseño de la clase, al declarar a los atributos que la conforman debemos indicar que visibilidad poseen: públicos, privados, protegidos, de acuerdo a esto, el objeto será manipulado de diferentes maneras:

public : Se permite que la variable sea accedida de modo directo, o sea, puede dársele valor, cambiarlo o borrarlo con una simple asignación.

private : Solo se permite la manipulación de la variable a través de métodos (y estos deben ser públicos). De este modo uno puede garantizarse que el seteo, el borrado o los cambios de los valores guardados en la misma se realicen de forma correcta.

protected : es un estado intermedio a los anteriores, Las clases derivadas de la que se esta definiendo, osea que heredan esos atributos, pueden manipularlos directamente como si fueran propios y no a través de los métodos de la clase padre.

2.4.1.4 *Resumiendo*

Los atributos de un objeto son sus características, es lo que lo describe, conforma. Dichos atributos se definen a traves de tipos, estos pueden ser primitivos como los descriptos anteriormente o complejos, o sea objetos con sus propios atributos.

2.4.2 *Estado de un objeto*

Es la situación en un momento determinado (cuando se esta observando al objeto) del objeto. Sus atributos, sus relaciones, sus asociaciones existentes, etc.

Dicho de otra forma: Un conjunto de circunstancias o atributos que caracterizan a una persona o cosa en un momento dado; manera o forma de ser: condición [V.A.:1980] [9].

Mas avanzados ahondaremos un poco más sobre este tema ya que estará muy relacionado con los dos que siguen y con la importancia e ingeniería que se necesita para describirlos y representarlos.

2.4.2.1 *Cambio de estado*

Un cambio de estado es la transición de un objeto desde un estado a otro.

Según el punto de vista:

- El cambio de estado es el cambio de la asociación del objeto
- o es el cambio en uno o mas atributos o relación del objeto.

2.4.3 *Eventos*

Un evento en la vida de un objeto es un cambio de estado significativo.

si bien estos estarán definidos según el modelo de negocio estudiado, algunos son generales y estarán presente siempre como por ejemplo:

Evento de creación En el evento de creación aparece enteramente un nuevo objeto. Se recibe la indicación que un objeto es creado como la "instancia" de un tipo de objeto. Sin su tipo apropiado de objeto, ningún objeto puede ser reconocido. Una vez creado este objeto pasa a formar parte del conjunto de tipos de objeto.

Evento terminación Es cuando un objeto es borrado de nuestro "conocimiento", deja de existir la instancia del tipo de objeto.

2.4.3.1 *Pre Estado y Post Estado de un Evento*

En muchas ocaciones, al analizar un objeto en un determinado ***estado***, se puede relevar que para llegar a dicha situación se tendrían que haber cumplido ciertos requisitos o que ese objeto cumpla con determinadas condiciones, a esto lo llamaremos ***pre-estado***.

Un pre estado es un estado que debe considerarse en un objeto antes que ocurra un evento.

Un post estado es un estado que debe poseerse en un objeto despues que ocurra un evento.

2.4.3.2 *Eventos externos, internos y temporales*

Evento externo: es cuando el resultado de la operación realizada es externa al dominio analizado.

Evento interno: es aquel en que el resultado de la ocurrencia del mismo queda dentro del dominio analizado.

Evento temporal: es el resultado de operaciones fijadas por tiempos de reloj.

2.4.4 *Métodos*

Los distintos lenguajes orientados a objetos han influenciado mucho en los nombres de los conceptos generales del paradigma, según los autores estos nombres pueden ir cambiando a pesar que referencien a las mismas ideas.

Una operación o método es un proceso que puede ser requerido como una unidad, invocado por o al objeto en que pertenece.

Es la implementación del *evento*. Ejemplo: el evento de *Persona Empleada* es el resultado de ejecutar el método *emplearPersona*

Hemos estado hablando que el encapsulamiento es una de las características principales de la orientación a objetos. Esta provee protección alrededor de los datos de un objeto almacenados en sus variables/atributos. Podemos definir el nivel de acceso a esos datos y por lo tanto estos se podrán manipular solo en determinados métodos definidos con el objeto.

2.4.4.1 *Variables de entrada de una operación*

Cada operación puede requerir objetos con quien operar. Estos pueden ser recibidos en los métodos al inicio del mismo, en el momento en que se invocan.

2.4.4.2 *Variables de salida de una operación*

Esto también puede ser conocido como *Valores de retorno*. Una operación o método es una función que dada una entrada retorna una salida.

2.4.4.3 *Operaciones con múltiples eventos*

En muchas situaciones las operaciones son modeladas con un evento simple. Esto suele ser lo ideal, lo recomendable: un método sirve para hacer una cosa y solo esa cosa. Pero a veces multiples eventos pueden ocurrir cuando una operación se completa, dependiendo la complejidad de lo requerido.

2.4.4.4 *Pre y Post condición*

Una pre condición especifica una restricción bajo la cual una operación se desempeña correctamente y una post condición especifica aquellas condiciones que deben resultar cuando se completa el método.

2.4.4.5 *Sintaxis*

La forma de definir en UML a los métodos es la siguiente:

visibilidad nombre (parámetros) : tipo de retorno (propiedades)

Donde: parámetros es:

nombre : tipo = valor por defecto

La visibilidad es idéntica a los conceptos de los atributos. Y los parámetros se encadenan con , cuando se requieren mas de uno.

Cuando el método no retorna ningún valor se declara la palabra *void*.

3 LA CLASE

Ahondaremos un poco mas en el concepto de ***Clase*** ya que para nosotros será la estructura fundamental en el momento de programar el diseño.

3.1 LA CLASE COMO ESTRUCTURA

Un concepto fundamental es el de *Tipo de Dato Abstracto*, es lo que permite tener flexibilidad en el diseño, modularizar correctamente. Uno de los principales tipos es la Clase. Ya la hemos mencionado y ahora veremos sus características principales.

¿Cuál es el concepto central de la tecnología de objetos? Hay que pensar dos veces antes de contestar "objeto". Los objetos son útiles, pero no son nuevos en sí mismo: desde que Cobol ha tenido estructuras, desde que Pascal ha tenido registros, C tiene estructuras. Obviamente los objetos siguen siendo importantes para describir la ejecución de un sistema orientado a objetos. Pero la idea básica, de la que todo deriva de la tecnología de objetos, es la de *clase*.

Una clase es un tipo abstracto de datos equipado posiblemente con una implementación parcial de la aplicación.

Como todo tipo de dato abstracto (TDA), una clase es un tipo: describe un conjunto de posibles estructuras de datos, llamadas instancias de una clase. Hablar de tipos de datos abstractos es hablar en forma muy genérica, pero hablar de una clase es hablar de una estructura de datos que se puede representar en el memoria de un ordenador y manipulada por un sistema de software.

Por ejemplo, si se ha definido una clase **STACK**, añadiéndole la información adecuada para su representación, las instancias de esa clase serán estructuras de datos que representan las pilas individuales.

Otro ejemplo, desarrollado es una clase **POINT** modelizando la noción de punto en un espacio bidimensional, bajo alguna representación adecuada, una instancia de esa clase es un estructura de datos que representa un punto. En una representación cartesiana, cada instancia de **POINT** es un registro con dos campos representando las coordenadas horizontal y vertical, X e Y, de un punto.

Una definición que ya hemos dicho es que el concepto de *clase* esta enlazado con el de *objeto*. A modo de diseño, análisis, las clases son los objetos del sistema. Y siendo mas precisos, mas técnicos, un objeto es simplemente la instancia de una clase. Es la materialización de esa clase en la memoria de la computadora.

Una clase es un modelo y un objeto es una instancia de ese modelo. Esta propiedad es tan obvia que normalmente no merecen comentarios más allá de las definiciones anteriores para los que están acostumbrado a la tecnología de objetos, pero en general esto es un error, es facil caer en confusiones.

Veamos el siguiente ejemplo de un libro:

"Se puede identificar un objeto "Usuario" en el espacio de un problema donde el sistema no necesita mantener ningún tipo de información acerca del usuario. Este caso, el sistema no necesita el número habitual de identificación, nombre, privilegios de acceso, y similares. Sin embargo, el sistema no necesita supervisar el usuario, en respuesta a peticiones y proporcionar información de manera oportuna. Y así, debido al servicio requerido en beneficio de la cosa del mundo real (en este caso, el Usuario), tenemos que añadir un objeto que corresponde al modelo del espacio del problema."

En el ejemplo vemos que el objeto representado por la palabra "usuario" se usa en dos sentidos diferentes: El usuario típico del sistema interactivo en discusión y el concepto de usuario en general.

3.1.1 *El molde y la Instancia*

Pensemos en un libro, considérelo como un objeto en el sentido mas común del término. Tiene sus propias características individuales: puede ser nuevo o ya manoseado por los lectores anteriores, tal vez escrito por dichos lectores (si fue usado para estudiar o le pusieron el nombre) si es de una biblioteca cuenta con un código de identificación. Sin embargo, las propiedades básicas como título, editor, autor y contenidos, están determinados por una descripción general que se aplica a cada copia individual. Este conjunto de propiedades no define a un objeto, sino a una clase de objetos (un tipo de objeto, en este sentido, las nociones de tipo y clase son idénticas).

Llame a la clase Libro A. Este se define como un cierto molde. Los objetos construidos a partir de este molde, serían las copias del libro, se llaman instancias de la clase. Otro ejemplo de molde sería el molde de yeso que un escultor hace para obtener una versión invertida del diseño de un conjunto de estatuas idénticas; cualquier estatua derivada del molde es una instancia del molde.

Lo que va a escribir en su sistemas de software es la descripción de las clases, tal como una clase **LINKED_STACK** describiendo propiedades de las pilas en una cierta representación. Cualquier ejecución particular de su sistema puede utilizar las clases para crear objetos; cada uno de estos objetos se deriva de una clase, y se llama una instancia de esa clase. Por ejemplo, la ejecución puede crear un objeto de la pila vinculado, derivado de la descripción dada en la clase **LINKED_STACK**; tal objeto es una instancia de la clase **LINKED_STACK**.

La clase es un texto de software. Es estática, es decir, que existe independientemente de cualquier ejecución. Por el contrario, un objeto derivado de la clase es un conjunto de datos creados de forma de estructura dinámica, que sólo existe en la memoria de la computadora durante la ejecución de un sistema.

Esto, por supuesto, está en línea con la discusión anterior sobre los tipos abstractos de datos: cuando especificando a un **STACK** como un **TDA**, no se describe ninguna pila en particular, pero la noción general de pila, un molde del que se puede derivar casos individuales.

Las declaraciones de ***X es una instancia de T*** y ***X es un objeto de tipo T*** serán consideradas como sinónimos para esta discusión. Igualmente hay que tener en cuenta que esto no siempre será así, ya que cuando incorporemos el concepto de *herencia* se tendrá que distinguir el concepto directo de instancia de una clase (construido a partir del patrón exacto definido por la clase) y su instancia en el sentido más general (casos directos de la clase o de cualquiera de sus particularizaciones o casos heredados).

Como vemos, las ideas si bien no son complejas a veces se prestan a una confusión, generalmente debido al uso semántico de las palabras, se puede tener la tentación de usar la palabra objeto por simple en más oportunidades de las convenientes. A veces, cuando nos referimos a la manipulación informática de cosas materiales y claramente visibles, como cuentas bancarias, aviones, documentos y en otras ocasiones de otras tantas que no tienen fuera del sistema representación alguna como pueden ser listas o estados. Lo importante es contextualizar a las palabras y definir claramente su significado para no confundir los términos reales y no confundir al objeto con la clase o la instancia de la misma según su uso.

Algunos lenguajes orientados a objetos, especialmente Smalltalk, han introducido una noción de **MetaClass** para manejar este tipo de situaciones. Un meta-clase es una clase cuyas instancias son clases mismas. Un poco más adelante veremos mejor este tipo especial de clases, al que llamaremos Clase Abstracta.

3.1.1.1 *El rol de la clase*

Para entender el enfoque orientado a objetos, es esencial darse cuenta de que las clases juegan dos funciones que los enfoques pre-OO siempre habían tratado por separado: el módulo y el tipo.

Los lenguajes de programación y otras anotaciones utilizadas en el desarrollo de software (diseño lenguas, lenguajes de especificación, notaciones gráficas para análisis) siempre incluyen tanto alguna facilidad para indicar módulos y algún sistema de tipos.

Un módulo es una unidad de descomposición de software. Hay varias posibles formas de módulo, tales como rutinas y paquetes. Independientemente de la elección exacta de la estructura del módulo,

ya que la descomposición en módulos sólo afecta a la forma del software, y no lo que el software puede hacer, de hecho es posible, en principio, escribir cualquier programa Ada como un solo paquete o cualquier programa de Pascal como un único programa principal. No se recomienda tal enfoque, cualquier desarrollador de software competente utilizarán los módulos del lenguaje que este usando para descomponer su software en partes manejables. Pero si tomamos un programa existente, por ejemplo en Pascal o C, que siempre puede combinar todos los módulos en uno solo y aún así obtener un sistema de trabajo con la semántica equivalente y buena performance cumpliendo con lo que debe hacer. Así que la práctica de la descomposición en módulos es dictado por la ingeniería de software y los principios de gestión de proyectos y no por necesidad intrínseca.

El concepto de Tipo, a primera vista, es muy diferente. Un tipo es la descripción estática de ciertos objetos dinámicos: los diversos elementos de datos que se procesarán durante la ejecución de un sistema de software. El conjunto de tipos por lo general incluye los tipos predefinidos, tales como CHARACTER o INTEGER y así como aquellos que pueden ser definidos por el desarrollador o por los lenguajes: estructuras, punteros, matrices, listas, etc.

El concepto de tipo es un concepto semántico, ya que cada tipo influye directamente en la ejecución de un sistema de software mediante la definición de la forma de los objetos que en el sistema se crean y manipulan en tiempo de ejecución.

3.1.2 *La clase como módulo y tipo*

En los enfoques no-OO, el módulo y el concepto de tipo son distintos como digimos antes. La más notable propiedad de la noción de clase es que subsume estos dos conceptos, la fusión en una sola construcción lingüística. Una clase es un módulo, o una unidad de descomposición de software, pero también es un tipo (o, en los casos de generalidad, un patrón de tipo).

Gran parte de la potencia del método orientado a objetos se deriva de esta identificación. La Herencia, en particular, sólo puede entenderse plenamente si la miramos para proporcionar tanto a Módulos de ampliación y el tipo de especialización.

3.1.3 *Sistema de tipos uniforme*

Un aspecto importante de la orientación a objetos es por la simplicidad y uniformidad del sistema de tipos, eso se puede resumir en la siguiente propiedad:

REGLA DEL OBJETO Cada objeto debe provenir de una clase.

Esta regla no solo se aplicará a los objetos definidos por el desarrollador (por ejemplo, estructuras de datos con varios campos), sino también a los objetos básicos tales como números enteros, números reales, valores booleanos y caracteres, que todos serán considerados como casos de predefinido clases de biblioteca (INTEGER, REAL, DOUBLE, BOOLEAN, CHARACTER).

Insistir en esto ultimo tiene ventajas:

- Siempre es conveniente contar con un marco simple y uniforme en lugar de muchos casos especiales. Aquí el sistema de tipos se basa completamente en la noción de clase.
- Describir los tipos básicos como TDA y por lo tanto las clases es simple y natural. No es difícil, por ejemplo, ver cómo definir la clase INTEGER, la cual debería tener funciones que abarcan operaciones aritméticas tales como: operaciones booleanas de comparación, suma, resta, y las propiedades asociadas, derivadas de los axiomas matemáticos correspondientes.
- Mediante la definición de los tipos básicos como clases, les permitimos participar en todo el juego de características de la Orientación a Objetos, especialmente de herencia y generalidad. Si no tratamos a los tipos básicos como clases, tendríamos que introducir limitaciones severas y muchos casos especiales.

3.2 LA ESTRUCTURA CLASS

Veremos algunas características que son generalmente compartidas por todas las clases.

Veamos el siguiente ejemplo, simple y común: un Punto.

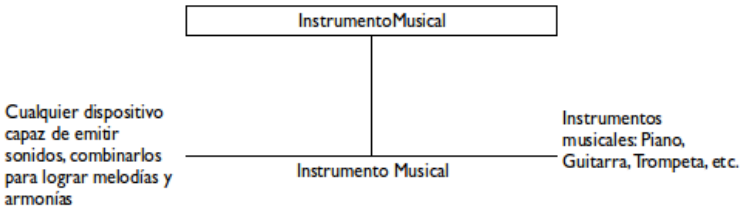


Figure 23: Representaciòn del Concepto. Definido por extensión y comprensión

Para la caracterización de un tipo **POINT** como un tipo abstracto de datos, necesitaríamos las cuatro funciones de consultas para x, y, ρ, Θ. (Los nombres de los dos últimos se especificarán como ro y theta en nuestro sistema). La función x nos da la abscisa de un punto (coordenada horizontal), Y es la ordenada (coordenada vertical), ρ su distancia al origen, Θ el ángulo con el eje horizontal.

Los valores de x y y para un punto se llaman sus coordenadas cartesianas, los de Θ y ρ son las coordenadas polares. Otra función útil es la de distancia, lo que dará lugar a la distancia entre dos puntos.

La especificación podría contar con comandos como traducir (para mover un punto por un desplazamiento horizontal y vertical dada), rotar (girar el punto por un cierto ángulo, alrededor del origen) y la escala (para llevar el punto más cerca o más lejos de la origen por un factor determinado).

Cualquier tipo de datos abstracto como **POINTS** caracterizados por un conjunto de funciones, que describe el operaciones aplicables a los casos de la ADT. En las clases de implementaciones (ADT), funciones producirán funciones -las operaciones correspondientes a las instancias de la clase.

Además de las funciones u operaciones (que a nivel de especificación irían acompañadas por una explicación de como cada función se implementa: por el espacio o el tiempo, se requiere atributos que serán los datos con los que trabajarán las funciones.

Podríamos definir una clasificación, esto es una definición algo arbitraria en el sentido de los nombres, ya que varían según las bibliografías, pero las características de la clase podría ser así:

1. los atributos, los cuales instanciados estarán en la memoria de la computadora

2. las rutinas: las cuales realizan una acción de cómputo. De las cuales podríamos diferenciar los procedimientos, que no tienen retorno alguno, o sea ejecutan acciones internas; y las funciones, las cuales retornan un resultado.

En general, resumiremos a las operaciones, funciones o rutinas, con el nombre de método, ya que no nos resultará necesario el distinguirlos.

Un tema asociado a esto, es el acceso que se debe tener a los atributos de una clase, ya se hizo una aproximación a esto. El acceso a los atributos debe ser uniforme en todo el diseño e implementación, debe estar definido. Puede ser tanto accediendo directamente, o mediante funciones. Esta última opción es la aconsejable, la que se debe usar siempre. De este modo se utiliza la potencialidad de la clase, y principalmente le da mayor consistencia y seguridad en la manipulación del objeto, ya que solo se hará a través de la interfaz que se publique.

3.2.1 *Las clases según los distintos lenguajes*

El primer lenguaje en usar la clase fue Simula. Dependiendo el lenguaje, obviamente la sintaxis cambiará, pero es importante siempre tener en claro el concepto de lo que implica la clase, la rep-

resentación del objeto en si y no el modo en que se escribe en el lenguaje.

Hay muchos lenguajees orientados a objetos, solo veremos algunos ejemplos de como se implementa:

3.2.1.1 *Simula*

Fue el primer lenguaje orientado a objetos, creado por Ole Johan Dahl y Kristen Nygaard en mayo de 1967, fue el que introdujo los conceptos de clase, objeto, herencia, polimorfismo, etc.

Listing 7: Declaración de una clase en Simula

```
Class nombre_clase (lista_ parametros);
        tipo_param_1 param1,...,  tipo_param_n param_n;
Begin
        declaracion_y_especificacion_de_atributos_y_metodos
        cuerpo_de_la_clase
End of nombre_clase;
```

Este sería un ejemplo completo y real para que puedan apreciar:

Listing 8: Clase Rectangulo en Simula

```
Class Rectangulo (Ancho, Alto);
        Real Ancho, Alto;
Begin ! Comienzo de la definicion de la clase;
        Real Area, Perimetro; ! Declaracion de atributos;

        Procedure Actualizar; ! Declaracion de metodos;
        Begin
                Area := Ancho * Alto;
                Perimetro := 2*(Ancho + Alto)
        End of Actualizar;

        Boolean Procedure EsCuadrado;
                EsCuadrado := Ancho=Alto;

        Update; ! Comienzo del cuerpo de la clase;
        OutText("Rectangulo creado: "); OutFix(Ancho,2,6);
        OutFix(Alto,2,6); OutImage
 End of Rectangulo; ! Fin de la clase;
```

3.2.1.2 C++

Este lenguaje es una variedad enriquecida y bajo el paradigma de objetos del lenguaje C. Si bien tiene una complejidad elevada, el código y la potencia del mismo es enorme. Fue creado a principio de 1980 por Bjarne Stroustrup. C++ admite tanto la programación orientada a objetos como la procedural, bajo la visión estricta es un híbrido por admitir y permitir mezclar ambos paradigmas.

En Listing 9 vemos el esquema de la clase en C++.

Listing 9: Declaracion de una clase en C++

```
class <identificador de clase> [<:lista de clases base>] {
public:
        //declaracion de atributos publicos
        tipo_1 nombre_1;
        tipo_2 nombre_2

        //declaracion de metodos publicos
        void f_1 (tipo_1 parametro, tipo_2 parametro_2);

protected:
        // declaracion de atributos y metodos protegidos

private:
        // declaracion de atributos y metodos privados
}
```

Un ejemplo completo sería Listing 10:

3.2.1.3 *Smalltalk*

Los orígenes de Smalltalk se encuentran en las investigaciones realizadas por Alan Kay, Dan Ingalls, Ted Kaehler, Adele Goldberg y otros durante los años setenta en el Palo Alto Research Institute de Xerox (conocido como Xerox PARC), para la creación de un sistema informático orientado a la educación. El objetivo era crear un sistema que permitiese expandir la creatividad de sus usuarios, proporcionando un entorno para la experimentación, creación e investigación.

Python

Fue diseñado por Guido Van Rossum en 1991. Es multiparadigma, por lo tanto también se lo considera híbrido. Es un lenguaje interpretado.

Listing 10: clase Tupla en C++

```
class Tupla {
private:
        // Datos miembro de la clase "tupla"
        int a, b;
public:
        // Funciones miembro de la clase "tupla"
        void Lee(int &a2, int &b2);
        void Guarda(int a2, int b2) {
                a = a2;
                b = b2;
        }
};

void Tupla::Lee(int &a2, int &b2) {
        a2 = a;
        b2 = b;
}
```

Listing 11: clase Empleado en Smalltalk

```
"Una subclase de Empleado que agrega un protocolo que se
    necesita para los empleados con salario"

 Empleado subclass: #EmpleadoConSalario
        instanceVariableNames: 'posicion salario'
        classVariableNames: ' '
        poolDictionaries: ' ' !
 ! EmpleadoConSalario publicMethods !
        posicion: unaPosicion
                posicion := unaPosicion !
        posicion
                ^posicion !
        salario: unSalario
                salario := unSalario !
        salario
                ^salario ! !
```

3.2.1.4 *Java*

El lenguaje de programación Java fue originalmente desarrollado por James Gosling de Sun Microsystems (la cual fue adquirida por la compañía Oracle) y publicado en el 1995. Su sintaxis deriva mucho de C y C++, pero tiene menos facilidades de bajo nivel que cualquiera de ellos. Las aplicaciones de Java son compiladas a bytecode (.class) que puede ejecutarse en cualquier máquina virtual Java (JVM) sin

Listing 12: Clase Persona en Python

```
class Persona(object):
     def __init__(self, nombre, edad):
         self.nombre = nombre # Una Propiedad cualquiera
         self.edad = edad # Otra propiedad cualquiera
     def mostrar_edad(self): # Es necesario que, al menos, tenga
         un parametro, generalmente: "self"
         print self.edad # mostrando una propiedad
     def modificar_edad(self, edad): # Modificando Edad
         if edad < 0 or edad > 150: # Se comprueba que la edad no
              sea menor de 0 (algo imposible), ni mayor de 150 (
             algo realmente dificil)
             return False
         else: # Si esta en el rango 0-150, entonces se modifica
             la variable
             self.edad = edad # Se modifica la edad
```

importar la arquitectura de la computadora subyacente. Java es un lenguaje de programación de propósito general, concurrente, orientado a objetos y basado en clases.

Listing 13: Definicion de clase en Java

```
[propiedad] class [nombre] extends [clase base]{
        //declaracion de atributos
        [propiedad] [tipo] nombreVariable;
        // declaracion de metodos
        [propiedad] [tipo de retorno] nombreMetodo ( variable1,
            variable2);
}
```

3.2.2 *Clientes y Proveedores*

Veremos el modo de uso de una clase. Siguiendo con nuestro ejemplo anterior, sólo hay dos maneras de utilizar una clase como POINT. Una forma es heredar de ella. La otra es convertirse en un punto en cliente de.

CLIENTE - PROVEEDOR Sea S una clase. Una clase C, que contiene una declaración de la forma a: S se dice que es un cliente de S. Entonces S es un proveedor de C.

3.3 EL OBJETO COMO ESTRUCTURA DINÁMICA

Hemos visto que hay diferentes significados para algunas de las palabras según el contexto o la generalización que se haga, principalmente esto se da con las palabras como *objeto* o con *instancia*. No son las únicas pero si tal vez las más significativas.

Conceptualmente, y por eso el paradigma se denomina así, el objeto de diseño en nuestro sistema es algo que debe ser representado en el mismo, dentro de un diseño técnico, ya no simplemente funcional, hablamos *clases*, como vimos antes, la cual es algo mas inmaterial, es el código de nuestro sistema. En este enfoque, el objeto está representado por la instancia de la clase, o sea, la materialización de la misma. A esto nos referimos como una estructura dinámica en tiempo de ejecución.

Daremos nuevos conceptos bajo este enfoque.

3.3.1 *Objetos*

En cualquier momento durante su ejecución, un sistema orientado a objetos habrá creado un cierto número de objetos. La estructura de tiempo de ejecución es la organización de estos objetos y de sus relaciones.

Veremos algunas de sus características.

3.3.1.1 *¿Qué es un objeto?*

En primer lugar, debemos recordar lo que significa la palabra *objeto* para esta discusión y en este contexto. El concepto es simple y nada ambiguo:

OBJETO Es la instancia en tiempo de ejecución de una clase.

Un sistema de software que incluye una clase C puede, en diversos puntos de su ejecución, crear (a través de las operaciones de creación y la clonación, por ejemplo) instancias de C; tal caso es una estructura de datos construida de acuerdo con el modelo definido de C.

Igualmente, tenemos que tener cuidado, el término *objeto* está tan sobrecargado de significados cotidianos que a pesar de los beneficios que esto puede tener, en un sentido técnico de software ha dado su cuota de confusión. En particular:

- No todas las clases corresponden a tipos de objetos del dominio del problema. Las clases introducidas para el diseño y aplicación no tienen homólogos inmediatos en el sistema modelado. Ellos son a menudo entre los más importantes en la práctica, y el más difícil de encontrar (Ejemplo: Clases Abstractas).

- Algunos conceptos del dominio del problema pueden ser clases en el software (y objetos en la ejecución del programa) a pesar de que no serían necesariamente clasificados como objetos en el sentido habitual del término. Una clase como ESTADO o COMANDO entran en esta categoría por ejemplo.

Consideremos el ejemplo dado con anterioridad. El del Punto, poseía dos atributos y sus métodos. El código sería así :

Listing 14: Declaracion de una clase en C++

```
class Punto {
private:
        float x, y;

public:
        // declaracion de atributos y metodos privados
}
```

La instancia del mismo o sea, el objeto generado en tiempo de ejecución lo podríamos representar de la siguiente forma Figure 24

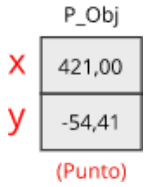


Figure 24: Representaciòn de la instancia de la clase punto.

Debe quedar claro que esto es una simple representación del espacio en memoria de la computadora que estaría siendo usado por la instancia de la clase Punto.

Aquí vemos que el objeto se conforma con el espacio de memoria para los dos atributos que lo conforman, el que posean valores implica que en algún momento de la ejecución del programa ocurrió lo siguiente Listing 15:

Listing 15: Seteo de atributos

```
Point.setX(421.00);
Point.setY(-54.41);
```

3.3.1.2 *Referencias*

Hasta ahora, hemos visto atributos que son tipos de datos primitivos, pero estos limitan mucho la potencia de la clase al momento de definirla como una estructura, como un tipo de dato abstracto y complejo.

Este sería el caso por ejemplo de una clase *Libro*, el cual uno de los campos es el atributo *Autor* de tipo *Escritor* como se ve en Listing 16.

Listing 16: Tipo Abrstracto y Complejo

```
class Escritor {
private:
     string nombre, nombreReal;
     int anioNacimiento, anioMuerte;
  // continua el codigo
}

class Libro {
     string titulo;
     int fecha;
     int cantPaginas;
     Escritor autor;
  // continua el codigo
}
```

Ejecutando al sistema, cada instancia de estas clases serán objetos diferentes, los objetos *libros* que se creen harán *referencia* a un objeto *Escritor*.

Las instancias podrian ser de la siguiente forma Figure 25

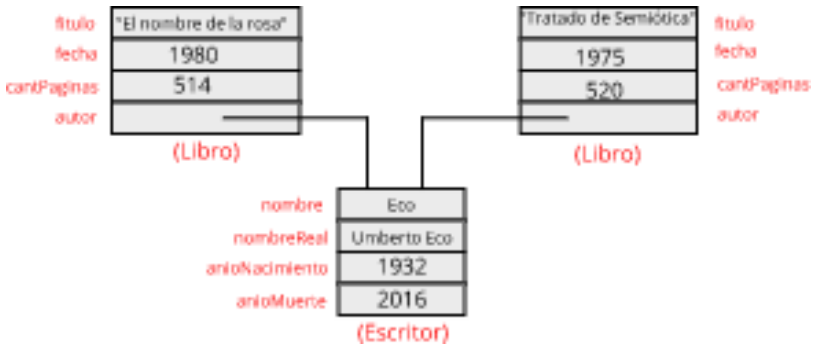


Figure 25: Instancias de estructuras complejas.

Como vemos, ambos objetos libros hacen referencia al mismo autor y por lo tanto al mismo objeto instanciado. En los libros, el valor guardado en la variable author es la dirección de memoria del objeto Escritor que se creó.

Una referencia es un valor en tiempo de ejecución. Puede ser nulo o no. En este último caso, la referencia señala, identifica a un objeto que queda *adjuntado*. Teniendo en cuenta esta definición y siguiendo el ejemplo, si se instanciara un libro de autor anónimo la referencia al objeto Escritor seria nula como vemos en Figure 26.

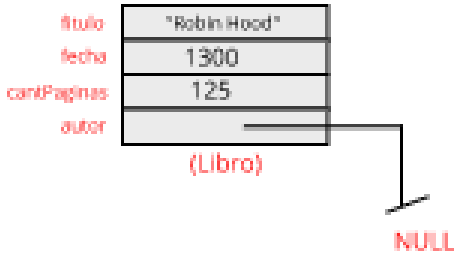


Figure 26: Referencia nula.

3.3.1.3 *La identidad de los objetos*

La noción de referencia trae el concepto de la identidad del objeto. Cada objeto creado durante la ejecución de un sistema orientado a objetos tiene una identidad única, independiente de el valor del objeto definido por sus campos. En particular:

1. Dos objetos con diferentes identidades pueden tener campos idénticos.
2. Por el contrario, los campos de un determinado objeto puede cambiar durante la ejecución de un sistema, pero esto no afecta a la identidad del objeto.

3.3.1.4 *Sub Objetos*

Por ejemplo podríamos pensar en un objeto libro, en una nueva versión LIBRO2 de la clase de libro, como tener un campo de autor que es en sí un objeto, como informalmente sugiere el siguiente cuadro Figure 27

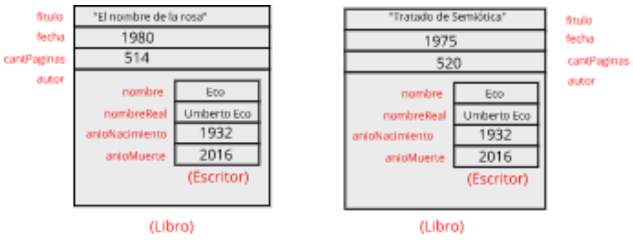


Figure 27: Sub-Objetos.

Esta situación, esta opción para este ejemplo en particular tiene serias desventajas:

- Es una pérdida de espacio en la memoria.

- Aún más importante, esta técnica no tiene en cuenta la necesidad de expresar el compartir la información. Independientemente de las opciones de representación, los campo author de los dos objetos se refieren a la misma instancia del escritor, si se actualiza el objeto ESCRITOR (por ejemplo, para registrar la muerte del autor), es deseable que el cambio afecta a todos los objetos de libros asociado con el autor dado.

3.3.1.5 *Auto-Referencias*

Algunos de estos conceptos los hemos visto en un sentido de diseño, como un modo de relación.

Un objeto puede contener un campo referencia que se une a sí mismo. Este tipo de auto-referencia también puede ser indirecta. En Figure 28 que se muestra a continuación, el objeto con *Almaviva* en su campo de nombre es su propio dueño (ciclo de referencia), el objeto *Figaro* ama *Susana*, que le encanta *Figaro* (ciclo de referencia indirecta).

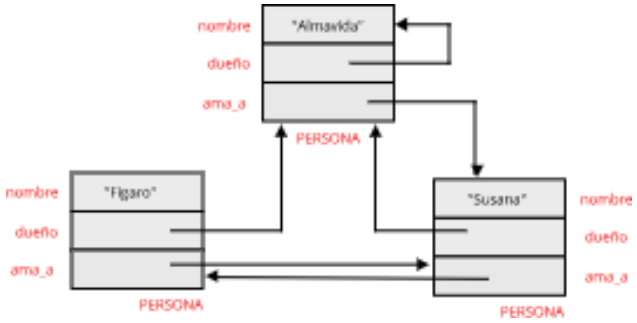


Figure 28: Auto-Referencia directa e indirecta.

Si lo pensamos en el modo de diseño veriamos lo siguiente Figure 29:

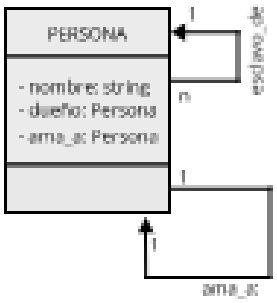


Figure 29: Auto-Referencia vista de diseño.

3.3.2 *Manipulando Objetos y Referencias*

3.3.2.1 *La creación*

Hemos visto anteriormente, en forma rápida, como son los eventos de creación, Ahondaremos un poco mas en detalle y veremos como funciona el tema de las referencias en sí misma.

Haremos un paralelo con respecto a Java y C++. En general esto nos mostrará los mismos conceptos pero con las diferencias de implementación de los lenguajes.

El evento de creación, del que hemos hablado anteriormente en otros apartados, podría ser invocado implicita o explicitamente según el lenguaje. En general, el modo de hacerlo es a través de ***new*** para un objeto X, donde X es un tipo de referencia basado en una clase C.

En java es necesaria que la creación sea explicita como vemos Listing 17.

Listing 17: Instanciacion en Java

```
public class Ejemplo{

        private String ejemplo;

        public Ejemplo(){ ejemplo = null;}
        public Ejemplo(String e){ ejemplo = e;}

        public setEjemplo(String e){ ejemplo = e;}

    public static void main (Strings []args){
        Ejemplo Ej; /* Se declara el objeto */
        Ej = new Ejemplo(); /* Se instancia */

        Ej.setEjemplo("Hola"); /* Se lo utiliza */
    }

}
```

Como vemos, debemos invocar la creación ya que la sola declaración no genera una instancia del objeto.

En C++ se mantiene la filosofía de C. Puedo declarar variables o referencias a variables y las clases son una variable en el lenguaje, un tipo de dato.

En Listing 18 vemos las variables, declaradas localmente en el main. C++ invoca implicitamente al constructor por defecto para crear al objeto.

Listing 18: Declaracion de objetos en C++

```
class claseC {
    /* atributos privados */

    claseC() { };

    /* Metodos publicos */
}

int main (){
    claseC C; /* se invoco al constructor */

    C.setVar("Hola"); /* se utiliza al objeto */
}
```

En caso que se omita la declaración de un constructor, C++ impone uno por defecto pero no inicializa las variables, o sea, estas tendrán la basura que este en momoria en el momento de la instancia.

El código cambia si en vez de usar variables por valor, utilizamos referencias a los objetos, o sea, punteros. En Listing 19 vemos como cambia el ejemplo anterior:

Listing 19: Declaracion de objetos en C++

```
class claseC {
    /* atributos privados */

    claseC() { };

    /* Metodos publicos */
}

int main (){
    claseC *C; /* se declara el puntero */

    C = new claseC(); /* se instancia el objeto */

    C->setVar("Hola"); /* se utiliza al objeto */
}
```

En este caso es necesario hacer de forma explicita el llamado al constructor.

Hagamos una pausa y veamos algunas observaciones. El código debe estar sujeto a lo que es nuestro diseño, por lo tanto, no sería correcto que no se expliciten constructores y se utilicen los que por

defecto haga el sistema o que por ejemplo, todas las variables sean inicializadas en null cuando esto puede no ser correcto para nuestro modelo de negocio.

Es responsabilidad del diseñador y del programador proveer la interfaz correcta para manipular a los objetos aprovechando la potencia del leguaje que se use.

3.3.2.2 *Constructores en C++*

Veremos algunos ejemplos más sobre la sintaxis de los constructores y la forma de invocarlos.

Listing 20: Uso de constructores por defecto en C++

```
class claseC {
    /* atributos privados */

    /* Metodos publicos */
}

int main (){
    claseC c1; /* se instancia con el constructor por default que
         asigno implicitamente */
    claseC c2(); /* Esto es un error*/
    /* no deben usarse los parentesis cuando el constructor no
        recibe parametros*/
}
```

En Listing 21 se define un constructor que recibe parametros.

Del ejemplo planteado por Listing 21 se desprende una elegante implementación como vemos en Listing 22. C++ hace un tratamiento de los datos primitivos como objetos

3.3.2.3 *Comparación, copia y clonación de objetos*

Intuitivamente diremos que dos objetos son iguales si sus atributos son iguales. Con respecto a la comparación de referencias, debemos tener en claro lo que se pretende: La referencia tecnicamente es la dirección de memoria en que se encuentra un objeto determinado, por lo tanto, al comparar dos referencias, si son iguales, implica que señalan al mismo objeto, pero esto no siempre es lo recomendable o lo requerido, muchas veces se debe tener que observar al objeto por sus valores contenidos para compararlos.

El desasignar la referencia de un objeto es asignar *null* a la misma. Esto no hace que el objeto se borre de la memoria, simplemente se quita la dirección al mismo, dependiendo del lenguaje que se use, se

Listing 21: Uso de constructores con parametros en C++

```
class claseC {
    /* atributos privados */
private:
    int a, b;

    claseC (int, int);

    /* Metodos publicos */
}
claseC::claseC (int x, int z){
        a = x;
        b = z;
}
int main (){
    claseC c1; /* Esto es un error, no existe constructor sin
        parametros */
    claseC c2(3, 6); /* instancia el objeto de tipo claseC */

}
```

Listing 22: Uso de constructores con parametros en C++

```
class claseC {
    /* atributos privados */
private:
    int a, b;

    claseC (int, int);

    /* Metodos publicos */
}
claseC::claseC (int x, int z): a(x), b(z) {

}
int main (){
    claseC c1; /* Esto es un error, no existe constructor sin
        parametros */
    claseC c2(3, 6); /* instancia el objeto de tipo claseC */

}
```

deberá tener que liberar la memoria manualmente o no, en caso que tenga un *garbage collector*.

La clonación es la generación de un objeto idéntico a otro. Hay que tener en claro que a partir de ese acto, cada uno tiene una vida independiente.

Para poder ver si son iguales, no podremos hacer una comparación de las referencias, cada uno tendría una dirección de memoria diferente, el método equals() deberá comparar atributo por atributo o sobrecargar al operador = en caso de C++.

Vale aclarar una diferencia entre dos conceptos: *clonación* y *copia* de dos objetos. En el primero se crea un objeto idéntico al que se poseía. En el segundo, los dos objetos existen y

3.3.2.4 *Persistencia de los objetos y la relacion con sus referencias*

Cada vez que un mecanismo de almacenamiento almacena un objeto, debe guardar con el los objetos dependientes del mismo. Cada vez que un mecanismo de recuperación recupera un objeto almacenado previamente, también deberá recuperar a todas las referencias que este posea guardadas.

4

TECNICAS DE ABSTRACCIÓN

4.1 HERENCIA

Ya hemos visto el concepto en forma teórica, genérica. La idea de la generalización, de objetos que pertenecen a conjuntos, familias que lo incluyen. Esto además de ser una de las ideas fundamentales para el paradigma orientado a objetos, tiene claros impactos en el diseño y la implementación.

Definimos que un objeto hereda de otro cuando es del mismo tipo de ese otro como vemos en Figure 30.

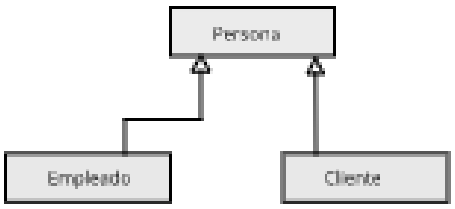


Figure 30: Herencia

En este caso, Empleado *es* una Persona, Cliente *es* una Persona. Cabe aclarar que Cliente *no* es un Empleado.

La herencia implica que todas las características, o sea atributos y métodos del objeto padre, estén en el objeto hijo.

Esto es totalmente equivalente a decir lo siguiente:

Utilizamos la primera visión por una cuestión de economía, de aprovechar realmente el sentido de la herencia. Si hay un cambio en Persona, al realizarlo en la clase padre, automáticamente se ve reflejado en las clases hijas.

Como seguramente se ha dicho en otras oportunidades, la implementación depende de los lenguajes, estas pueden variar. En Listing 23 vemos como declaramos la clase y sus prototipos y en Listing 24 el cuermpo de los métodos en C++.

Analicemos un poco los ejemplos **??** y Listing 24. Tenemos la clase *Empleado*, los atributos propios de la clase y tres métodos, constructor, que recibe dos de esos atributos, un método que modifica sueldo, y la *mostrarEmpleado()*, que transforma en un string los datos que se encuentran en nombre, numEmpleado y sueldo.

También vemos que aparece una palabra reservada, propia del lenguaje en el constructor de la clase:*this*. Esta se utiliza para señalar a la propia clase, fundamentalmente para distinguir cuales son datos propios de la clase y cuales los locales del método:

Listing 23: herencia.h

```
//En el archivo header

#include <iostream>
#include <strng>
using namespace std;

class Empleado {

private:
        string nombre;
        int numEmpleado , sueldo;

        static int contador = 0;

public:

        Empleado(string, int);
        void aumentarSueldo(int);
        string mostrarEmpleado();

}
```

Listing 24: herencia.cpp

```
//En el archivo cpp

#include "herencia.h"

Empleado::Empleado(string nombre, int sueldo) {
    this.nombre = nombre;
    this.sueldo = sueldo;
    numEmpleado = ++contador;
}

void Empleado::aumentarSueldo(int porcentaje) {
    sueldo += (int)(sueldo * aumento / 100);
}

string Empleado::mostrarEmpleado() {
    return "Numero: " + numEmpleado + " Nombre: " + nombre + "
        Sueldo: " + sueldo;
}
```

El método constructor esta recibiendo dos parámetros llamados nombre y sueldo, los cuales coinciden con los atributos de la clase,

entonces es imprescindible que dentro de la función se distinga cual es uno y cual es otro: *this.nombre* es el atributo de la clase y simplemente *nombre* referencia a la variable recibida en el método.

Luego podremos definir una clase *Ejecutivo*, el cual *es* un Empleado, o sea, hereda de el, esto lo hacemos de la siguiente forma Listing 25

Listing 25: Ejecutivo es un Empleado

```
class Ejecutivo: public Empleado {
private:
        int presupuesto;

}
```

Ejecutivo además de tener los atributos de un empleado, tiene un atributo que lo distingue, un atributo propio que es presupuesto.

Obviamente que podemos agregar cualquier método que sea para manipular su atributo propio:

4.2 HERENCIA MÚLTIPLE

4.3 CLASES ABSTRACTAS

Cuando comenzamos a definir una jerarquía de clases, y tenemos una estructura de herencias de las mismas, podemos encontrar que algunos de los objetos definidos, son conceptos, que son genéricos y que estos no serán instanciados, simplemente definen un tipo y poseen atributos y métodos comunes a otros que a su vez, los realizan de modo diferente. Ejemplo: tengo Cliente, Proveedor y Empleado, los tres son Personas con atributos y métodos comunes, entre ellos, altaDePersona(), lo singular es que cada uno requiere de un alta diferente, o sea, el procedimiento de dar de alta a un cliente es diferente al alta de un empleado.

Como la clase Persona está diseñada para que sea heredada por las otras y no para instanciarse y manipularse, se definirá como Abstracta. A su vez, los métodos de esta pueden ser abstractos o no, según se quiera.

Cada lenguaje implemetará estos conceptos de algún modo particular, pero están presentes. Por ejemplo:

- En C++ se dice que es una clase abstracta toda aquella que al menos tenga un método virtual. Esto obligará a las clases que hereden a implementarlo como se ve en Listing 26.
- En Java la clase abstracta se declara simplemente con el modificador ***abstract*** en su declaración. Los métodos abstractos se

declaran también con el mismo modificador como vemos en Listing 27.

Listing 26: Clase abstracta en C++

```
//En el archivo header

class Persona {
private:
     /* declaracion de atributos */
public:
    Persona();
    virtual void altaDePersona();
    /* declaracion de metodos */
};

class Empleado : public Persona {
private:
     /* declaracion de atributos*/
public:
    Empleado();
    void altaDePersona();
    /* declaracion de metodos */
};

//En el archivo de codigo cpp

void Empleado::altaDePersona(){

     /* implementacion del metodo */
}
```

En ambos casos, sea el método virtual o abstract, solo se establece el prototipo (sin el bloque de código encerrado entre). La clase derivada se declara e implementa de forma normal, como cualquier otra, se establece la herencia y estará obligada a implementar los métodos abstractos, sino el compilador lanzará un error.

Por tanto, las clases abstractas tendrán dos utilidades principales:

1. En primer lugar, evitan que los usuarios de la clase puedan crear objetos de la misma, como dice la definición de clase abstracta. De este modo, en nuestro ejemplo anterior, no se podrán crear instancias de la clase *Persona*.

2. En segundo lugar, permiten crear interfaces que luego deben ser implementados por las clases que hereden de la clase abstracta. Es evidente que una clase abstracta, al no poder ser instanciada, no tiene sentido hasta que una serie de clases que heredan de

Listing 27: Clase abstracta en Java

```
abstract class Persona {

    /* declaracion de atributos */
    abstract void altaDePersona();
    /* declaracion de metodos */
}

class Empleado extends Persona {

    /* declaracion de atributos*/

    void altaDePersona(){
          /* implementa el metodo */
    }
    /* implementacion de metodos */
}
```

ella la implementan completamente y le dan un significado a todos sus métodos. A estas clases, que son las que hemos utilizado a lo largo de todo el curso, las podemos nombrar clases concretas para diferenciarlas de las clases abstractas.

Hay que tener en cuenta que la clase abstracta es una clase, un objeto. Por lo tanto, tendrá atributos, métodos (abstractos o no) y constructores.

¿Para qué sirven los constructores de una clase que no se puede instanciar? Para que las clases heredadas puedan inicializar los atributos. Es la única vez que se usarían, dentro de los constructores de los subtipos.

Veamos el ejemplo de Figure 31, C++ posee algunas diferencias pero en escencia todos los conceptos anteriores son válidos. Aquí veremos que se debe utilizar virtual para hacer abstracta al método y por lo tanto a la clase, y sus herencias deberán implementarlas.

En Listing 28 vemos como esta definido el método abstracto.

virtual double getTIPO()"

Un método abstracto debe incluir el modificador *virtual* que nos permitirá acceder a él de modo polimorfo.

En Listing 29 podemos destacar la definición que de los métodos *getPrecio(): double* y *getParteIVA(): double* hemos podido realizar en el nuevo entorno creado:

Vemos que cualquiera de los dos (*getParteIVA(): double* directamente y *getPrecio(): double* indirectamente a través del primero) acceden al

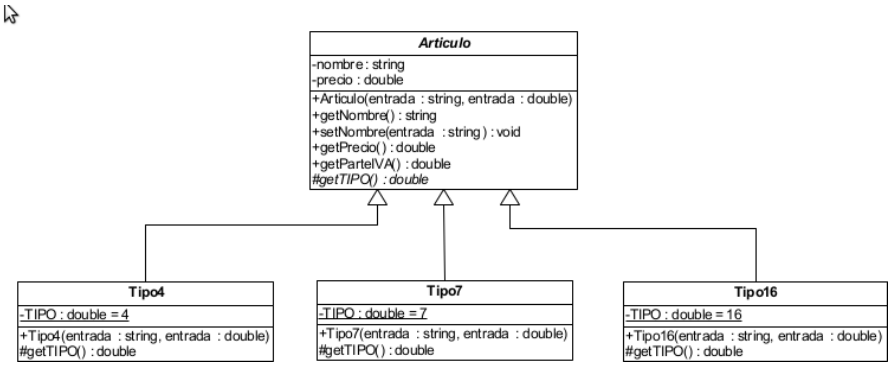


Figure 31: Tipos de articulos

Listing 28: articulo.h

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

#ifndef ARTICULO_H
#define ARTICULO_H 1

class Articulo{
private:
     char string;
     double precio;
     public:
     Articulo(string, double);
     char * getNombre();
     void setNombre(char []);
     double getPrecio();
     double getParteIVA();
     protected:
     virtual double getTIPO();
};
#endif
```

método (abstracto) *getTIPO(): double*, cuya definición no ha sido dada todavía (y lo será en alguna de las clases derivadas).

Aquí es donde el compilador debe desarrollar la tarea de asegurar que no podamos construir objetos de la clase abstracta *Articulo*, ya que para los mismos no habría una definición de *getTIPO(): double*, y comprobar que en las subclases que definamos de *Articulo* y de las cuales queramos construir objetos, el método *getTIPO(): double* sea definido.

Vale mencionar que si pretendemos un main similar a Listing 30

El compilador daría dos errores, por intentar construir un objeto Articulo (ni siquiera a través de un puntero al mismo); pero si po-

Listing 29: articulo.cpp

```
//Fichero Articulo.cpp

#include "Articulo.h"

Articulo::Articulo(char nombre [], double precio){
     strcpy (this->nombre, nombre);
     this->precio = precio;
}

char * Articulo::getNombre(){
     return this->nombre;
}

void Articulo::setNombre(char nuevo_nombre[]){
     strcpy (this->nombre, nuevo_nombre);
}

double Articulo::getPrecio(){
     return this->precio + this->getParteIVA();
}

double Articulo::getParteIVA(){
     return this->precio * this->getTIPO()/100;
}
```

Listing 30: main.cpp

```
// archivo main.cpp

#include <iostream>
#include "Articulo.h"
#include "Tipo4.h"
#include "Tipo7.h"
#include "Tipo16.h"
using namespace std;

int main (){
     Articulo arti ("La historia interminable", 9);
     arti.getParteIVA();
     Articulo * art1;
     art1 = new Articulo ("La historia Interminable", 9);
     system ("PAUSE");
     return 0;
}
```

dríamos usarlo como esta en Listing 31 podemos declarar objetos de tipo abstractos pero siempre son inicializados a través de subtipos de este.

Listing 31: main.cpp

```
// archivo main.cpp

#include <iostream>
#include "Articulo.h"
#include "Tipo4.h"
#include "Tipo7.h"
#include "Tipo16.h"
using namespace std;

int main (){
     Articulo * art1;
     art1 = new Tipo4 ("La historia Interminable", 9);
     Tipo7 * art2;
     art2 = new Tipo7 ("Gafas", 160);
     Articulo * art3;
     art3 = new Tipo16 ("Bicicleta", 550);
     cout << "El precio del primer articulo es " << art1->
         getPrecio() << endl;
     cout << "El precio del segundo articulo es " << art2->
         getPrecio() << endl;
     cout << "El precio del tercer articulo es " << art3->
         getPrecio() << endl;
     system ("PAUSE");
     return 0;

}
```

4.4 INTERFACES

Podríamos definir a la interface como una clase abstracta pura, o sea todos sus métodos son abstractos. Es una colección de métodos abstractos y atributos los cuales por definición todos son públicos, estáticos y finales. En Listing 32 vemos ejemplos.

Este concepto esta implementado en java. En C++ no existe la interface como objeto, pero no sería complicado implementarlo a través de clases abstractas que cumplan con los requisitos que se mencionan antes, y como C++ soporta la herencia múltiple no hay problemas en su uso.

Listing 32: Interfaces

```
/** La sintaxis seria la siguiente **/

interface nombre_interface {
        tipo_retorno nombre_metodo (lista_argumentos) ;
        . . .
}

/** El uso seria el siguiente **/

class nombre_clase extends nombre_clase_base implements interfaz1
    [, interfaz2, ...] {
        /* obligatoriamente se tendran que declarar e implementar
             los metodos de las interfaces */
        /* resto del codigo */
}

/** En un caso real en java **/

interface InstrumentoMusical {
        void tocar();
        void afinar();
        String tipoInstrumento();
}

/** su implementacion seria **/

class InstrumentoViento extends Object implements
    InstrumentoMusical {
        void tocar() { ... };
        voidafinar() { ... };
        String tipoInstrumento() {}
}

class Guitarra extends InstrumentoCuerda {
        String tipoInstrumento() {
                return "Guitarra";
        }
}
```

4.4.1 *Referencias a Interfaces*

Es posible crear referencias a interfaces, pero las interfaces no pueden ser instanciadas. Una referencia a una interface puede ser asignada a cualquier objeto que implemente la interface como ser ve en Listing 33. El concepto es el mismo que se maneja para la *clase abstracta*

Listing 33: Referencias a Interfaces

```
/** Podemos trabajar a la Interfaz de este modo **/

InstrumentoMusical instrumento = new Guitarra();
instrumento.play();
System.out.prinln(instrumento.tipoInstrumento());
InstrumentoMusical i2 = new InstrumentoMusical(); //error.No se
    puede instanciar
```

4.4.2 *Herencia de Interfaces*

Las interfaces pueden extender otras interfaces o sea que pueden heredar de varias. La sintaxis es como se muestra en Listing 34

Listing 34: Referencias a Interfaces

```
interface nombre_interface extends nombre_interface , ...{
        tipo_retorno nombre_metodo(lista_argumentos) ;
        /* ... */
}
```

Part II

LENGUAJE DE PROGRAMACIÓN C++

Haremos un recorrido de las particularidades mas destacadas de C++. Recomendamos leer la publicación de Bjarme Stroustrup sobre lo que es la programación orientada a objetos, el paper esta anexado en el apendice de este documento.

5 SINTAXIS DE C++

Hay un porcentaje muy alto dentro de la sintaxis del C++ que viene heredada del C. De ahi su nombre de C incrementado.

Nos trataremos de concentrar en las particularidades del nuevo lenguaje pero tambien aprovecharemos para reapasar algunos conceptos que ya se vieron al estudiar C.

5.1 LA ESTRUCTURA DE UN PROGRAMA

Veremos con un breve ejemplo las partes componentes básicas.

Listing 35: holaMundo.cpp

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

int main ()
{
        cout << "Hola Mundo!" << endl;
        return 0;

}
```

El la línea 1 vemos la directiva al proprocesador, es igual a las que se usan en C. Comienzan con # esta en particular incluye a las librerias que abarcan los objetos para la entrada y salida estandar. Note que no lleva el *.h*.

La línea 2 indica el uso de una estructura propia de C++ llamada *namespace* que se encuentra en *iostream.h* llamada *std*. Esta, si se desea, podría omitirse pero el código sufriría un pequeño cambio, la llamada a la salida por pantalla de la línea 6 se vería asi: `std :: cout << "HolaMundo" << endl;`.

Al igual que en C, la función principal para la ejecución del programa es la *main*. Su sintaxis es identica al viejo lenguaje:

int main () { ... }

o

int main (int argc, char argv*

){ ... }

En la primera opción, tal vez la más común, se define la función que retorna un entero sin recibir parámetros, en la segunda, los argumentos de entrada.

Entre llaves irá el código, finalizando la función con el retorno del entero según lo indicado en el encabezado.

Los comentarios se pueden realizar por lineas o por bloques al igual que se hacen en C.

La lína 6 hace la llamada referenciando a la standard character output (*std::cout*. El operador << es de inserción o de flujo, enviá a la salida estandar la cadena de caracteres. Toda línea de sentencias finaliza con ;.

5.2 COMPILACIÓN Y EJECUCIÓN DEL PROGRAMA

Una vez escrito el código, se guardará en un archivo con extensión *.cpp* y en la línea de comando, posicionados en el directorio donde el archivo se ubica escribimos:

```
$ g++ Hola.cpp
$ ./a.out
Hola Mundo!
$
```

El resultado de la compilación será un archivo ejecutable llamado *a.out*.

Al usar el compilador de *gnu*, la compilación posee similares opciones que cuando usamos el lenguaje C. Algunas opciones son:

- *-o<nombre>*: para que el ejecutable tenga un nombre determinado.
- *-std=<standard>*: para que compile con un determinado standard. Ej: *std=c++17*.
- *-Wall*: para que muestre los warnings de compilación.

5.3 VARIALBES

C++ es un lenguaje fuertemente tipado, eso implica que las variables deben declararse indicando tipo y nombre antes de usarse. Ejemplo:

En la línea 2, encontramos la definición de un entero llamado x sin inicializar; en la 4 vemos un entero a al que se asigna el número 10 y en la 6 vemos una declaración y asignación de valor equivalente a la anterior aunque se escriba de otra forma.

C++ distingue entre mayúsculas y minúsculas.

Tenemos los siguientes tipos básicos:

char, int, float, double, bool y void

además se pueden agregar a la definición del tipo int:

Listing 36: variables.cpp

```
int x;

int a = 10;

int z(10);
```

Listing 37: enteros.cpp

```
signed int x;

unsigned short int a;

long int z;
```

signed, unsigned, long y short

El uso de *long/short* reserva mayor o menor espacio permitir asignar valores mas o menos grandes.

El *signer/unsigned* definen valores negativos o no, haciendo que el rango numérico varíe:

El *unsigned* posee un rango de 0 a 65535, y el *signed* tendrá desde el -32768 al 32767.

5.3.1 *Estructuras de control*

No ahondaremos en esto, es idéntico al C.

Se posee el **if/else, do/while, for, switch**

5.3.2 *Funciones*

También son iguales a C.

Listing 38: funciones.cpp

```
tipo_de_retorno nombre (tipo arg1, ..., tipo argn){

    if (arg1 > arg2)
        return arg1;
    else
        return arg2;
}
```

Al igual que en C, los tipos de retorno pueden variar y obviamente el *return* debe respertar a ese tipo. Además se debe tener la función antes de usarla, o al inicio declarar el prototipo de la misma.

Listing 39: ej.cpp

```
// Ejemplo completo
// ej.cpp

#include<iostream>
using namespace std;

int maximo (int, int);

int main ()
{
        cout << max(1 , 2) << endl;
}

int max (int a, int b){
    if ( a > b)
        return a;
    else
        return b;
}
```

5.4 OBJETOS

Podemos considerar que un *objeto* es la representación de algo físico o virtual. Ej: un auto, un usuario, un email, etc. Cada objeto tiene una existencia independiente de los otros y puede haber objetos complejos, o sea, compuestos por otros objetos en forma interna.

Posee características y funcionalidad.

5.4.1 *Las clases*

La *clase* es el diseño del objeto, a partir de una clase un objeto puede ser instanciado, o sea creado en memoria.

Podriamos decir que la *clase* es la descripción en código del objeto. Tiene un nombre, atritutos (campos, propiedades y datos) y una interfaz (los métodos o funciones para manipular al objeto).

La clase funciona como un tipo de dato en nuestro programa.

En la línea 20 vemos la declaración de dos objetos de tipo Alumno. En las siguientes vemos dos formas de dar valor a un atributo de dichos objetos.

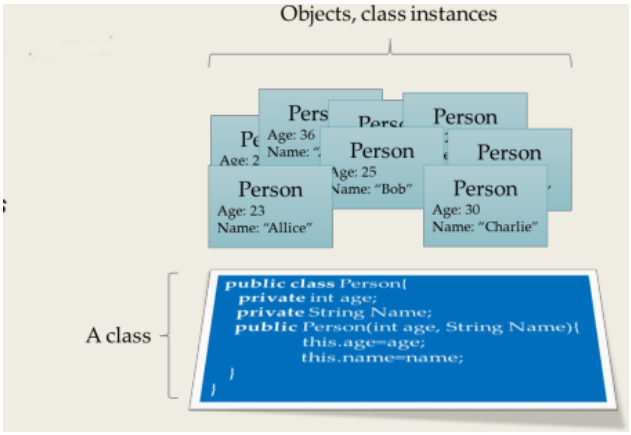


Figure 32: Clases y Objet

Listing 40: clase.cpp

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

class Alumno{
private:
    variables // los atributos
    ...
    metodos // la interfaz
public:
    variables
    ...
    metodos

};

int main ()
{
        Alumno Pablo, Maria;

        Pablo.setNombre("Pablo");
        Maria.edad = 25;

        return 0;
}
```

Más adelante veremos la factibilidad de estos seteos y las herramientas que da el leguaje para manipular atributos y protegerlos de mal usos.

5.5 ENCAPSULAMIENTO

Podemos simplificar al concepto de encapsulamiento al de encerrar al objeto y aislarnos de su funcionamiento, solo sabremos que tiene determiados atributos y una interfaz de métodos para manipular el comportamiento de dicho objeto.

Para esto C++ nos define unas características de permisos de acceso.

5.5.1 *Permisos públicos y privados*

Los atributos y métodos pueden ser públicos o privados.Aquellos que sean públicos podrán ser accedidos desde cualquier punto del código. Siendo este código el que manipule al objeto. En cambio los miembros privados solo pueden ser accedidos desde el interior del objeto al que pertenecen.

El permiso por defecto, cuando no se indica es el de ***privado***

El buen uso y criterio de programación es que los atributos son siempre privados

5.5.2 *Métodos*

Los métodos son miembros de la clase y por lo tanto tienen acceso a los atributos de la misma sin importar el que estos sean públicos o privados.

Obviamente cada método puede definir y poseer sus propias variables. son funciones y por lo tanto se declaran como se venia haciendo hasta el momento.

Los métodos pueden ser públicos y privados, en caso de estos últimos, solo podrán ser invocados por otros métodos de la misma clase.

El conjunto de métodos públicos conforman la *interfaz* del objeto, es por intermedio de ellos que se manipularán los atributos y el comportamiento en general, esto responde al concepto de ***Encapsulamiento***

Si bien c++ lo permite, no deben existir los métodos o funciones globales (Exceptuando el main).

En los ejemplos que hemos visto, las funciones están definidas e implementadas a la vez dentro de la clase (*in line*). Esto solo es recomendable para métodos simples, breves. Lo correcto es declarar los prototipos en las clases e implementarlas en forma externa. Ejemplo:

En la línea 16 vemos el prototipo dentro de la clase y en la 20 vemos la implementación. Hay que tener en cuenta que al estar externa tenemos que indicar a que clase pertenece el método ya que podrían existir métodos con igual nombre pero de diferentes clases, para esto se utiliza el operador de ámbito **::**.

Listing 41: metodos.cpp

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

class Alumno{
private:
    string nombre;

public:

    //Implementacion in line
    void setNombre(string nombre){
        this->nombre = nombre;
    }

    void enrolamiento (Materia);

};

void Alumno::enrolamiento(Materia M){
    // codigo del metodo
}
```

5.5.2.1 *Puntero this*

Es un puntero que posee todo objeto y se referencia a si mismo. O sea, esta autoapuntandose o autoreferenciandose.

En el uso mas general es para identificar o diferencias atributos miembros de la clase con otros locales o externos que poseen el mismo nombre. Ejemplo:

Listing 42: this.cpp

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

class Alumno{
private:
    string nombre;

public:
    vod setNombre(string nombre){
        this->nombre = nombre;
    }
};
```

En la línea 12 vemos que la variable de entrada setea al atributo de la clase, esta está reconocida o diferenciada gracias al uso del puntero this.

5.5.3 *Constructores y Destructores*

Hay dos métodos fundamentales en los objetos, que disparan dos eventos de igual importancia: lso *constructores* y los *destructores*.

5.5.3.1 *Constructores*

Es llamado automáticamente al crear el objeto.
Mediante el constructor se puede:

- Inicializar variables
- Reservar memoria
- Básicamente todo lo que se desee antes de iniciar el comportamiento del objeto.

5.5.3.2 *Destructores*

Es llamodo al momento que se destruye el objeto, esto puede ser por el borrado dinámico de la memoria alocada por el objeto o la terminación local de un objeto (cuando queda fuera del scope al finalizar una función o salir de una estructura de control).
Mediante el desctructor se puede:

- Liberar memoria reservada.
- Cualquier otra acción que se desee antes de que el objeto sea destruido totalmente.

5.5.3.3 *Ejemplos.*

Veremos un conjunto de ejemplos y variantes de constructores y desctructores, como se implementan y como se invocan.

En este caso, consideramos al constructor como uno por defecto, simplemente inicializa en vacio al atributo. En caso que se omita la declaración e implementación del constructor C++ lo asume.

Veremos una variante con la cual hay que ser cuidadoso:

y por último el ejemplo completo y posible, el cual hace uso de la sobrecarga de las funciones, o sea, un mismo método con varios modos de invocación.

Como se ve, se instancian dos objetos diferentes invocando cada uno a un constructor distinto según sea la conveniencia.

Los destructores realizan la función contraria al constructor. En caso de omitir su declaración C++ ejecutará uno por defecto pero en este caso esto puede ser riesgoso.

Listing 43: ejemnplo.cpp

```
// Constructor por defecto

#include<iostream>
using namespace std;

class Alumno{
private:
    string nombre;

public:
    void Alumno(){
        nombre = "";
    }
    void setNombre (string n){
        nombre = n;
    }
};

int main ()
{
    Alumno A; //esta llamando al contructor
              //por defecto
    A.setNombre("Pablo");
}
```

El destructor es fundamental cuando se hace reserva de memoria, ya que C++ es identico a C en este tema... No liberará automaticamente la memoria reservada.

En el ejemplo, vemos en la linea 7 otra forma de inicializar los atributos en el constructor. y en la 16 vemos al código del destructor. Además de liberar la memoria, es una buena práctica el chequear previamente si la memoria está usada antes de liberarla.

Veamos ahora dos posibles usos en el main...

Segundo ejemplo:

5.6 MANEJO DE MEMORIA

Se ha visto, por el lenguaje C, que la memoria está dividida en:

- HEAP donde se almacena la memoria en forma dinámica. Es más lenta que el stack requiere una manipulación dedicada, alocarse, liberarse, etc.
- STACK donde se almacenan las variables locales (automáticamente). Es rápida de cachear y simple en su uso. En propor-

Listing 44: constDefault.cpp

```
// Constructor por defecto

#include<iostream>
using namespace std;

class Alumno{
private:
    string nombre;

public:
    void Alumno(string n){
        nombre = n;
    }
    void setNombre (string n){
        nombre = n;
    }
};

int main ()
{
    Alumno A; //esta llamando al contructor
              //por defecto y no existe!! ERROR!!

    Alumno A("Juan"); // Construccion correcta.

    A.setNombre("Pablo");
}
```

ción es pequeña, no corresponde guardar estructuras de gran tamaño.

- DATA en donde se encuentran las variables globales y estáticas
- TEXT donde se encuentra el código que se ejecuta

5.6.1 *Notas sobre punteros*

Cuando se declara un puntero, debe tener que asignarselo a **nullptr**.

tipo_variable nombre_puntero {nullptr};

nullptr es interpretado como dirección 0. Una variable *int* podria asignarse a *NULL* pero esto podria provocar problemas en la ejecución del programa.

Escribiremos unos ejemplos simples

Listing 45: sobrecargaConst.cpp

```
// Constructor por defecto

#include<iostream>
using namespace std;

class Alumno{
private:
    string nombre;

public:
    void Alumno(){
        nombre = "";
    }
    void Alumno (string n){
        nombre = s;
    }

    void setNombre (string n){
        nombre = n;
    }
};

int main ()
{
    Alumno A; //esta llamando al contructor
              //por defecto

    Alumno B("Juan");

    A.setNombre("Pablo");
}
```

En el código superior vemos en la primera opción como se libera la memoria usada para el entero. En el segundo *delete* se ordena liberar la totalidad del arreglo.

Aquí vemos dos errores comunes, en la función *func* hay un retorno de una dirección que ya está liberada y luego en el *main* se asigna un valor a una variable que ya no tiene dirección de memoria disponible al haberse liberado.

Listing 46: destructor.cpp

```
class Poligono {
    int cantfPuntos;
    Punto* n_Puntos;
public:
    // Constructors...
    Poligono() : cantPuntos(0), n_Puntos(NULL) {}
    Poligono(int cantPoints);

    // Destructor:
    ~Poligono();

    void Print(){/*...*/}
};

Poligono::~Poligono() {
    if (n_Puntos != NULL)
        delete[]n_Puntos;
    cout << "Se ha borrado el Poligono!!!" << endl;
}
```

Listing 47: memFree.cpp

```
int main ()
{

    Poligono A;
    Poligono B(6);

    // Se declaran dos poligonos, al finalizar
    // el main se autoinvoca el destructor.
}
```

Listing 48: dinamicMemFree.cpp

```
int main ()
{

    Poligono *A = new Poligono();
    Poligono *B = new Poligono(6);

    // Se declaran dos punteros a poligonos,
    // y se reserva la memoria para ellos.

    // invoco los contrustores
    delete(A);
    delete(B);
}
```

Listing 49: puntero.cpp

```
int main ()
{

    int* p = new int;
    delete p;

    double *m = new double[25];
    delete []m;
}
```

Listing 50: freePunteroError.cpp

```
char* func(){
    char str[10];
    strcpy(str, "Hola");
    return str;
}

int main ()
{

    int* p = new int;
    delete p;

    *p = 3;
}
```

6 ALGUNOS TIPOS DE DATOS FUNDAMENTALES

6.1 ORDEN DE PRESEDENCIA Y ASOCIATIVIDAD

Un grupo de operadores puede ser asociativo por izquierda, lo que significa que los operadores se ejecutan desde de izquierda a derecha, o pueden ser asociativos por derecha, lo que significa que se ejecutan de derecha a izquierda.

Casi todos los grupos de operadores son asociativos por la izquierda. Los únicos operadores asociativos derechos son los operadores unarios, operadores de asignación y operador condicional. La Tabla 10 *Precedencia y asociatividad de Operadores.* muestra el detalle de todos los operadores en C++

Precedencia	Operador	Asociatividad
1	::	left
2	() [] -> .	left
	posix ++ and –	
3	!	right
	unary + and -	
	prefix ++ and –	
	address-of & indirection *	
	C-style cast (type)	
	sizeof	
	new new[] delete delete[]	
4	.* .->	left
5	* / %	left
6	+ -	left
7	<<>>	left
8	< <= > >=	left
9	== !=	left
10	&	left
11	^	left
12	|	left
13	&&	left
14	||	left
15	?: (conditional operator)	right

Sigue en la página siguiente.

Precedencia	Operador	Asociatividad
	= *= /= %= += -= &= ^= |= <<= >>=	
	throw	
16	, (comma)	left

Table 3: Precedencia y asociatividad de Operadores.

6.2 TIPOS DE DATOS

El concepto de variable es el mismo que venimos manejando y heredamos del C. Cada variable posee un nombre y tiene un tipo que la define a ella y al contenido que tendrá.

Los valores numéricos se dividen en dos categorías, números enteros y números de punto flotante.

6.2.1 *Numeros enteros*

Vemos en el código: 78 declaración e inicialización de números enteros:

Listing 51: numInt.cpp

```
//mediante llaves inicilizadoras

int contador_manzana{15}; //Numero de manzanas
int contador_naranjas{5};
int total_frutas{contador_manzanas + contador_naranjas};

//mediante la notacion funcional

int contador_bananas(12);
int nuevo_total(contador_bananas + total_frutas);

// otra alternativa mas tradicional

int contador_peras = 6;
int suma_total = nuevo_total + contador_peras;
```

La inicialización con llaves es la sintaxis más reciente que se introdujo en C++ 11 específicamente para estandarizar la inicialización. Su principal ventaja es que le permite inicializar casi todo en el de la misma manera, razón por la cual también se la conoce comúnmente como *inicialización uniforme*. Otra ventaja es que la forma del inicial-

izador entre llaves es un poco más segura cuando se trata de limitar las conversiones, como vemos en el código: 92:

Listing 52: cast.cpp

```
int contador_banana(7.5);    //puede compilar sin warning
int contador_cocos = 5.3;    //puede compilar sin warning
int contador_papaya{0.3};    //como minimo lanza un warning, sino
     un error.
```

Si bien son todas las opciones son validas, es preferible usar las llaves y no provocar conversiones forzadas.

6.2.1.1 *Enteros con signo*

La memoria asignada para cada tipo y, por tanto, el rango de valores que puede almacenar, puede variar entre diferentes compiladores. En la tabla 11 vemos

TIPO	BYTES	RANGO
signed char	1	-128 a +127
short short int signet short signet short int	2	-256 a +255
int signed signed int	4	-2.147.483.648 a +2.147.483.647
long long int signed long signed long int	4 o 8	similar al int o al long long
long long long long int signed long long signed long long int	8	-9.223.372.036.854.775.808 a +9.223.372.036.854.775.807

Table 4: Enteros con Signos.

6.2.1.2 *Enteros sin signo*

Hay circunstancias en las que no es necesario almacenar números negativos. Cuando definimos cantidad de objetos, ese número es siempre un entero positivo. Se puede especificar tipos numericos que solo almacenen valores no negativos anteponiendo la siguiente palabra clave: unsigned char o unsigned short o unsigned long long, por ejemplo. Cada tipo unsigned ss un tipo diferente del tipo signed pero ocupa la misma cantidad de memoria.

El tipo char es un tipo de entero diferente tanto con signo como del char unsigned.

6.2.1.3 *Inicialización a cero*

La inicialización puede realizarse como lo muestra el ejemplo 80

Listing 53: zeroinit.cpp

```
int contador{0};    //iguala a cero

int contador2{};   //es equivalente a igualarlo a cero.
```

6.2.1.4 *Cálculos con enteros*

La operatoria con enteros es similar a C. No ahondaremos mucho sobre el tema ya que resulta trivial las operaciones entre números. En la tabla 12 vemos los operadores aritméticos

OPERADOR	OPERACIÓN
+	Suma
-	Resta
*	Multiplicación
/	División
%	Módulo

Table 5: Operadores Aritméticos.

En el código 81 vemos algunos ejemplos:

Listing 54: operaciones.cpp

```
long width {4};
long length {5};
long area { width * length };            // 20
long perimetro {2*width + 2*length};     // 18
```

```
long perimetro2{ (width + length) * 2 };  // 18
```

6.2.1.5 *Asignaciones*

Al igual que en el caso anterior, las operaciones de asignación son similares al como en que usabamos en C. Veamos ejemplos en: 82

Listing 55: asignacion.cpp

```
long perimetro {};
// ...
perimetro = 2 * (width + length);

int a {}, b {}, c {5}, d{4};
a = b = c*c - d*d;

int y {5};
y = y + 1;
```

Unido a esto, tenemos la opcion las operaciones con *op*= como vemos en los ejemplos 83

Listing 56: operAsignacion.cpp

```
int cantidad = cantidad % 2;

int nuevaCantidad %= 2; // es equivalente a la linea anterior

//Algunas otras opciones

x *= y + 1;

//su equivalente seria

x = x * (y + 1);
```

En la tabla 13 vemos los distintos *op*=.

Operacion	Operador	Operacion	Operador
suma	+=	and bit a bit	&=
resta	-=	or bit a bit	|=
multiplicación	*=	or exclusivo bit a bit	^=
división	/=	corrimiento a izquierda	<<=
modulo	%=	corrimiento a derecha	>>=

Table 6: operadores y asignacion.

6.2.1.6 *Incrementando y decrementando*

Vimos el uso del operador += o -=. Tenemos dos operadores más que se tienen que tener en cuenta: ++ y ——

Veamos formas equivalentes en el código 84

Listing 57: incremento.cpp

```
int count {5};
count = count + 1;
count += 1;
++count;
```

el incremento se realiza de 1 en 1. La última notación es mas concisa si lo que se busca es solo el incremento.

En el siguiente caso expresado en el código 85 vemos una operación, hay que tener en cuenta que operadores de incremento y decremento se ejecutan antes que cualquier otro operador aritmético binario en una expresión. Por lo tanto, si count tenia un valor de 5, el recuento se incrementará a 6 y luego este valor se utilizará en la evaluación de la expresión a la derecha de la asignación. Por lo tanto, al total se le asignará el valor 12.

Listing 58: incrementoPresedencia.cpp

```
total = ++count + 6;
```

Si tomamos la forma postfija del operador, incrementa a la variable despues de que su valor se usa en el contexto. En el ejemplo 86 vemos opciones equivalentes para entender esto:

Listing 59: incrementoPostfijo.cpp

```
total = count++ + 6;

// es equivalente a

total = count + 6;
++count;
```

Si bien el resultado no varía, hay que tener cierto cuidado con la escritura, $a++ + b$ es equivalente a $a+++b$, aunque es factible que en el último caso, el compilador realice $a + ++b$, sería mas claro $total = 6 + count++$ o alternativamente usando parentesis: $total = (count++) + 6$;

La observación puntual que se debe hacer, además del uso de parentesis, es la presedencia de signos y operadores y que implica el post y pre fijo del incremento y decremento. Veamos los siguientes ejemplos, tendremos en consideración que el valor inicial en casa cálculo de la variable *cuenta* es 5

total = - -cuenta + 6

El resultado que obtenemos es 10. Ahora, si consideramos la siguiente expresión:

total = 6 + cuenta - -

total tendrá un valor de 11.

¿Qué ocurre? Veamos paso a paso como se evaluan la expresión y sus componentes internas:

- Valor inicial: cuenta = 5, total = 0
- Evalua la ecuación total = 6 + cuenta - -
- Reemplaza el valor de cuenta y suma al 6 para setear total
- Decrementa cuenta

El decremento es la ultima operación que se ejecuta, posterior a todas las otras, si viera el valor de *cuenta* al final, sería 4. En el código 87 y su salida en pantalla:

Listing 60: decrementoPostfijo.cpp

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


int main(){
   int cuenta = 5;

   int total = 6 + cuenta--;

   cout<< "6 + cuenta —- = " << total << " y cuenta = " << cuenta
        <<endl;

   return 0;
}
```

```
$ g++ -std=c++17 -ooperadores operatoria.cpp
$ ./operadores
6 + cuenta-- = 11 y cuenta = 4
```

Algunos de estas declaraciones no están definidas en el estándard, esto no significa que no compile o se resuelve, solo que, el resultado puede no ser el esperado. Por ejemplo:

*total = ++count * 3 + count++ * 5*

Al ejecutarlo, el resultado es 48. Pero como hemos dicho, cuando las variables son usadas mas de una vez con operadores incrementales pueden dar resultados inesperados. Otro caso es el siguiente:

k = k++ + 1;

Aquí k es incrementada en el mismo momento que se asigna lo que hace que su valor se modifique dos veces, al ejecutarla, con un valor

inicial de 5, el resultado fue 6. C++ 17, estableció una regla para casos asi, en el que todos los efectos secundarios del lado derecho de una tarea (y esto incluye asignaciones compuestas, incrementos y decrementos) están completamente comprometidos antes de evaluar el lado izquierdo y la asignación real. Sin embargo, no establece mayor presición sobre lo bien o mal definida la expresión.

Es importante y aconsejable el modificar una variable solo una vez como resultado de evaluar una sola expresión y acceda al valor anterior de la variable sólo para determinar su nuevo valor, es decir, no intente leer una variable nuevamente después de haber sido modificado en la misma expresión.

6.2.2 *Números flotantes*

Tenemos tres posibles tipos de datos de punto flotante, como vemos en el detalle 14

Tipo	Descripción	Precisión
float	Valores de precision simple	7
double	Valores de doble precision	15
long double	Valores de precision doble extendida	18-19

Table 7: Tipos de punto flotante.

Es importante recordar que no se usa el modificador *signed* o *unsigned*, los tipos de punto flotante son siempre *signed*.

El manejo de estos tipos no difiere de lo que hemos visto, vemos unos ejemplos en 88

Listing 61: initFloating.cpp

```
const double pi {3.141592653589793};
double a {0.2};
double z {9};
double volume {};
volume = pi*z*z*a;
```

Hay ciertas limitaciones al trabajar con tipos de punto flotante. Los errores que pueden surgir son:

- Muchos valores decimales no se conviernten exactamente a valores binarios de punto flotante, aunque sean errores pequeños, pueden terminar amplificados en los cálculos que se hagan.
- Al tomar la diferencia de valores casi idénticos, se perderá la precisión

- Trabajar con valores que difieren en varios órdenes de magnitud puede generar errores.

Luego, debemos añadir resultados inválidos o no calculables. El estándar de punto flotante IEEE define valores especiales que tienen una mantisa binaria de todos ceros y un exponente de todos los unos para representar +infinito o -infinito, según el signo. cuando divides un valor positivo distinto de cero por cero, el resultado será +infinito, y dividir un valor negativo por cero resulta en -infinito.

Otro valor especial definido en el estándar es el *not-a-number*, que abreviaremos como *NaN*, serán los resultados del cero dividido cero, entre otras operaciones por ejemplo. En el código 89 vemos algunas de estas operaciones.

Listing 62: infinity.cpp

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

int main(){

double a{ 1.5 }, b{}, c{};
double result { a / b };
std::cout << a << "/" << b << " = " << result << std::endl;
std::cout << result << " + " << a << " = " << result + a << std::
    endl;
result = b / c;
std::cout << b << "/" << c << " = " << result << std::endl;


};
```

La salida en consola es:

```
$
$ ./operadores
1.5/0 = inf
inf + 1.5 = inf
0/0 = -nan
```

6.2.3 *Formateando la salida*

Sirven para cambiar el formato de lo que escribimos por consola. Presentamos algunos de los mas comunes en la tabla 15

Tipo	Descripción
std::fixed	Output floating-point data in fixed-point notation.
std::scientific	Output all subsequent floating-point data in scientific notation, which always

Sigue en la página siguiente.

Tipo	Descripción
	includes an exponent and one digit before the decimal point.
std::defaultfloat	Revert to the default floating-point data presentation.
std::dec	All subsequent integer output is decimal.
std::hex	All subsequent integer output is hexadecimal.
std::oct	All subsequent integer output is octal.
std::showbase	Outputs the base prefix for hexadecimal and octal integer values. Inserting std::noshowbase in a stream will switch this off.
std::left	Output is left-justified in the field.
std::right	Output is right-justified in the field. This is the default.

Table 8: Formateando la salida.

Las mencionadas, solo se invocan, estan accesibles mediante la libreria *iostream* y no reciben parametros. *iomanip* nos brinda algunas opciones mas como podemos ver en la tabla 16

Tipo	Descripción
std::setprecision(n)	Establece la precisión o los n dígitos decimales
std::setw(n)	Establece el ancho del campo de salida en n caracteres, pero solo para los siguientes datos de salida. La proxima salida vuelve al ancho estandar
std::setfill(ch)	Cuando el ancho del campo tiene más caracteres que el valor de salida, el exceso. Los caracteres en el campo serán el carácter de relleno predeterminado, que es un espacio. Este establece el carácter de relleno en ch para todas las salidas posteriores.

Table 9: Formateando la salida.

6.2.3.1 *Mezclando y convirtiendo tipos*

Tecnicamente, toda operación aritmética requiere que ambos operandos sean del mismo tipo. Aunque esto no siempre es el caso, ya que el compilador puedo convertir un operando al tipo del otro, a esto se

lo llama *conversión implicita*. Esto significa que el compilador tomará a la variable del tipo mas restringido para convertirla en la de mayor rango. En el ejemplo 90 podemos verlo:

Listing 63: convImplicit.cpp

```
unsigned int numero{};
const unsigned int factor{12};

double prom /= factor;

double area_pond{numer * prom}; //numero es convertido a double
```

Como se dijo, elige los operandos de rango mas limitado para convertirlo a los mas grandes, el ranking de tipos seria el siguiente, de mayor a menor:

1. long double
2. double
3. float
4. unsigned long long
5. long long
6. unsigned long
7. long
8. unsigned int
9. int

Se tiene que tener en cuenta que el resultado de restas de enteros unsigned pueden envolver numeros positivos muy grandes que suelen llamarse *underflow*. El fenómeno inverso se llama *overflow*, sumar un unsigned char cuyo valor es 253 y 5, no nos daría 258, sino 2, o 258 modulo 256.

En algunos casos, principalmente en asignaciones directas o inicializaciones, las conversiones pueden requerirse en forma explicita como vemos en 91

Listing 64: convExplicit.cpp

```
int y{};
double z{5.0};

y = z; // requiere del cast explicito
```

Para esta conversión, usaremos *static_cast<type_to_convert_to>(expresion)*; como vemos en 92

Listing 65: cast.cpp

```
double valor1{10.9};
double valor2{15.6};

int valor3{static_cast<int>(valor1) + static_cast<int>(valor2)};
```

Como es sabido, C++ no cambia la filosofía de este tipo de conversión, utilizará el *truncado* para inicializar al intero, que terminará quedando con un valor de 25 en caso del ejemplo citado.

C++ también admite el cast en la antigua forma de C, por lo general suele ser el modo mas usado, pero no porque la conversión explicita sea nueva, sino simplemente por uso y costumbre. Lo recomendado es realizar la conversión con la fución de C++.

6.2.3.2 *Buscando limites*

Encontrar los limites altos y bajos de un tipo puede resultar importante. En el código ejmplo 93 vemos el modo de obtenerlos

Listing 66: limites.cpp

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

int main()
{
cout << "El rango del tipo short es de "
            << numeric_limits<short>::min() << " a "
            << numeric_limits<short>::max()  << endl;
cout << "El rango del tipo int es de "
            << numeric_limits<int>::min() << " a "
            << numeric_limits<int>::max() << endl;
cout << "El rango del tipo long es de "
            << numeric_limits<long>::min()<< " a "
            << numeric_limits<long>::max() << endl;
cout << "El rango del tipo float is desde "
            << numeric_limits<float>::min() << " a "
            << numeric_limits<float>::max() << endl;
cout << "El rango positivo del tipo double es desde "
            << numeric_limits<double>::min() << " a "
            << numeric_limits<double>::max() << endl;
cout << "El rango positivo del tipo long double es desde "
            << numeric_limits<long double>::min() << " a "
            << numeric_limits<long double>::max() << endl;
}
```

Al ejecutarlo:

```
$g++ -std=c++17 -olimites limites.cpp
$ ./limites
El rango del tipo short es de -32768 a 32767
El rango del tipo int es de -2147483648 a 2147483647
El rango del tipo long es de -9223372036854775808 a 9223372036854775807
El rango del tipo float is desde 1.17549e-38 a 3.40282e+38
El rango positivo del tipo double es desde 2.22507e-308 a 1.79769e+308
El rango positivo del tipo long double es desde 3.3621e-4932 a 1.18973e+4932
```

6.3 USANDO CARACTERES

Las variables de tipo *char* son usadas para guardar un simple caracter y por lo tanto ocupan 1 byte. C++ no especifica la codificación de dicho caracter, en principio lo delega al compilador y usualmente es ASCII.

Los ejemplos 94 vemos como definir una variable de tipo *char*.

Listing 67: letras.cpp

```
char letra; //sin inicializar - el valor tiene basura
char si{'Y'}; no{'N'}; //inicializado con literales
char l{33}; //inicializado con '!'
```

El caso mas distinguido en el ejemplo es el último, vemos que el tipo caracter es numerico, por lo tanto podemos realizar estas operaciones como en el ejemplo 95

Listing 68: letrasOper.cpp

```
char l{'A'};
char letra{l + 5}; //la letra es 'F'
++l; //l ahora es 'B'
l += 3;  //l ahora es 'E'
```

La salida en pantalla no resulta una dificultad, vemos una posibilidad en el ejemplo 96

Listing 69: Salida en pantalla de char

```
std::cout << "l es '" << ch
          << "' y su codigo numerico " << std::hex << std::
              showbase
          << static_cast<int>(ch) << std::endl;
```

Esto nos mostraría lo siguiente:

```
l es 'E' y su codigo numerico 0x45
```

6.4 EL TIPO *auto*

Se usa la palabra reservada *auto* para indicar al compilador que deduzca al tipo de dato. en el ejemplo

Listing 70: Declaracion de auto

```
auto m {10};       // m es de tipo int
auto n {200UL};    // n tiene un unsigned long
auto pi {3.14159}; // pi tiene un tipo double

auto list = {1, 2, 3};   // list tiene un tipo std::
    initializer_list<int>
```

Vamos a ver algunas diferencias en el uso del auto, su declaracion, inicialización y asignación ya que la deducción de la que hablamos cambio con el C++17. Hay que tener cuidado cuando se usa auto. En el código

Listing 71: Diferencias con estandares en uso de auto

```
/* C++11 and C++14 */
auto i {10};              // i has type std::initializer_list<int
    > !!!
auto pi = {3.14159};      // pi has type std::initializer_list<
    double>
auto list1{1, 2, 3};      // list1 has type std::initializer_list
    <int>
auto list2 = {4, 5, 6};   // list2 has type std::initializer_list
    <int>

/* C++17 and later */
auto i {10};              // i has type int
auto pi = {3.14159};      // pi has type std::initializer_list<
    double>
auto list1{1, 2, 3};      // error: does not compile!
auto list2 = {4, 5, 6};   // list2 has type std::initializer_list
    <int>
```

6.5 ENUMERACIONES

A veces se necesitan variables que tengan un conjunto limitado de valores que puedan ser referenciados por su nombre, ejemplos, dias de la semana, meses, etc. Esta posibilidad la dan los *enum*, un tipo de datos de enumeración. En el código 99 vemos como se puede declarar.

Listing 72: enum

```
enum class Dia {Lunes, Martes, Miercoles, Jueves, Viernes, Sabado
    , Domingo};
```

Esto define un tipo de dato enumerador llamado Día, y las variables de este tipo solo pueden tener valores de el conjunto que aparece entre las llaves, de lunes a domingo. Si se intenta establecer una variable de tipo Día en un valor que no es uno de estos valores, el código no se compilará. Los nombres simbólicos entre llaves son llamados enumeradores.

Como vemos, fuera de la pequeña diferencia de sintaxis, hasta aquí funciona igual que en C.

A cada enumerador se le asignará un entero, comenzando por el 0, de esa forma: Lunes sería igual a 0 y viernes igual a 6. En 100 vemos una forma de uso.

Listing 73: Asignar a variable

```
Dia hoy {Dia::Martes};
std::cout << "Hoy es " << static_cast<int>(hoy) << std::endl;
```

La salida en pantalla sería: "Hoy es 1".

Con respecto a los valores,en 101 podemos ver diferentes formas de romper con la asignación automatica que mostramos antes.

Listing 74: Asignar valores del enum

```
/* Aqui Lunes tiene valor 1 y Domingo tendra valor 7 */
enum class Dia {Lunes = 1, Martes, Miercoles, Jueves, Viernes,
    Sabado, Domingo};

/* Aqui Lunes tiene valor 3, viernes llegaria al 7 y Sabado 1 y
    Domingo tendra valor 2 */
enum class Dia {Lunes = 3, Martes, Miercoles, Jueves, Viernes,
    Sabado = 1, Domingo};

/*No es necesario que las claves sean unicas*/
enum class Dia {Lunes = 1, Lun = 1,  Martes,  Miercoles, Jueves,
    Viernes, Sabado, Domingo};

/*Podemos definir enumeradores en termino de un enumerador previo
    */
enum class Dia { Lunes,                    Lun = Lunes,
                 Martes  = Lunes + 2,      Mar = Martes,
                 Miercoles = Martes + 2,   Mier = Miercoles,
                 Jueves = Miercoles  + 2,  Jue = Jueves,
                 Viernes   = Jueves + 2,   Vie = Viernes,
                 Sabado = Viernes  + 2,    Sab = Sabado,
                 Domingo   = Sabado + 2,   Dom = Domingo
               };
```

Como opción se podría realizar una declaracion como se muestran en 102

Listing 75: Signos de puntuación

```
enum class Punctuacion : char {Coma  = ',', Exclamacion = '!',
    Pregunta = '?'};

std::cout << "Hola" << static_cast<char>(Punctuacion::Exclamacion
    )
    << " Todo bien" << static_cast<char>(Punctuacion::Pregunta)
        << std::endl;
```

Vemos que se debe hacer un cast a char, esto nos indica que cuando se devinieron los enumeradores se uso el valor ASCII del caracter.

6.6 ALIAS PARA TIPOS DE DATOS

Muchas veces, por conveniencia en la descripción o para simplificar la declaración, es mas práctico definir alias a determinados tipos de datos.

C++ tiene una sintaxis para esto mismo como vemos en 103

Listing 76: Alias para tipos de datos

```
using BigOnes = unsigned long long;    // Define a BigOnes como
    alias

BigOnes mynum {};                      // Define & inicializa un
    tipo unsigned long long
```

Vemos que la forma es intuitiva.

Igualmente C++ sigo permitiendo la sintaxis proveniente de C como vemos en

Listing 77: Vieja sintaxis para el alias

```
typedef unsigned long long BigOnes;    // Define BigOnes como
    alias
```

7 VARIABLES

Todas las variables tienen un tiempo de vida finito. Existen desde el momento que son definidas y en algún momento serán destruidas, como mucho, en general, al final de programa. Similar como es en C. Esta duración puede estar determinada por el tipo de almacenamiento de la misma:

- Todas las variables definidas en un bloque (y que no sean estaticas) tiene una duración. Existen desde el punto en que se define hasta que el bloque finaliza (generalmente con la llave que se cierra). Se suelen llamar variables automáticas.
- Las variables definidas como estáticas (static) tienen vida desde el momento en que se declaran hasta que el programa finaliza.
- Las variables alocadas dinámicamente tienen vida desde el momento en que se declaran y crean hasta que se destruyen (C++ no lo hace automaticamente).

7.1 ORDEN DE PRESEDENCIA Y ASOCIATIVIDAD

Un grupo de operadores puede ser asociativo por izquierda, lo que significa que los operadores se ejecutan desde de izquierda a derecha, o pueden ser asociativos por derecha, lo que significa que se ejecutan de derecha a izquierda.

Casi todos los grupos de operadores son asociativos por la izquierda. Los únicos operadores asociativos derechos son los operadores unarios, operadores de asignación y operador condicional. La Tabla 10 *Precedencia y asociatividad de Operadores.* muestra el detalle de todos los operadores en C++

Precedencia	Operador	Asociatividad
1	::	left
2	() [] -> .	left
	posix ++ and –	
3	!	right
	unary + and -	
	prefix ++ and –	
	address-of & indirection *	
	C-style cast (type)	

Sigue en la página siguiente.

Precedencia	Operador	Asociatividad
	sizeof	
	new new[] delete delete[]	
4	.* .->	left
5	* / %	left
6	+ -	left
7	<<>>	left
8	< <= > >=	left
9	== !=	left
10	&	left
11	^	left
12	|	left
13	&&	left
14	||	left
15	?: (conditional operator)	right
	= *= /= %= += -= &= ^= |= <<= >>=	
	throw	
16	, (comma)	left

Table 10: Precedencia y asociatividad de Operadores.

7.2 TIPOS DE DATOS

El concepto de variable es el mismo que venimos manejando y heredamos del C. Cada variable posee un nombre y tiene un tipo que la define a ella y al contenido que tendrá.

Los valores numéricos se dividen en dos categorías, números enteros y números de punto flotante.

7.2.1 *Numeros enteros*

Vemos en el código: 78 declaración e inicialización de números enteros:

La inicialización con llaves es la sintaxis más reciente que se introdujo en C++ 11 específicamente para estandarizar la inicialización. Su principal ventaja es que le permite inicializar casi todo en el de la misma manera, razón por la cual también se la conoce comúnmente como *inicialización uniforme*. Otra ventaja es que la forma del inicializador entre llaves es un poco más segura cuando se trata de limitar las conversiones, como vemos en el código: 92:

Listing 78: numInt.cpp

```
//mediante llaves inicilizadoras

int contador_manzana{15}; //Numero de manzanas
int contador_naranjas{5};
int total_frutas{contador_manzanas + contador_naranjas};

//mediante la notacion funcional

int contador_bananas(12);
int nuevo_total(contador_bananas + total_frutas);

// otra alternativa mas tradicional

int contador_peras = 6;
int suma_total = nuevo_total + contador_peras;
```

Listing 79: cast.cpp

```
int contador_banana(7.5);    //puede compilar sin warning
int contador_cocos = 5.3;    //puede compilar sin warning
int contador_papaya{0.3};    //como minimo lanza un warning, sino
    un error.
```

Si bien son todas las opciones son validas, es preferible usar las llaves y no provocar conversiones forzadas.

7.2.1.1 *Enteros con signo*

La memoria asignada para cada tipo y, por tanto, el rango de valores que puede almacenar, puede variar entre diferentes compiladores. En la tabla 11 vemos

7.2.1.2 *Enteros sin signo*

Hay circunstancias en las que no es necesario almacenar números negativos. Cuando definimos cantidad de objetos, ese número es siempre un entero positivo. Se puede especificar tipos numericos que solo almacenen valores no negativos anteponiendo la siguiente palabra clave: unsigned char o unsigned short o unsigned long long, por ejemplo. Cada tipo unsigned ss un tipo diferente del tipo signed pero ocupa la misma cantidad de memoria.

El tipo char es un tipo de entero diferente tanto con signo como del char unsigned.

TIPO	BYTES	RANGO
signed char	1	-128 a +127
short short int signet short signet short int	2	-256 a +255
int signed signed int	4	-2.147.483.648 a +2.147.483.647
long long int signed long signed long int	4 o 8	similar al int o al long long
long long long long int signed long long signed long long int	8	-9.223.372.036.854.775.808 a +9.223.372.036.854.775.807

Table 11: Enteros con Signos.

7.2.1.3 *Inicialización a cero*

La inicialización puede realizarse como lo muestra el ejemplo 80

Listing 80: zeroinit.cpp

```
int contador{0};    //iguala a cero

int contador2{};   //es equivalente a igualarlo a cero.
```

7.2.1.4 *Cálculos con enteros*

La operatoria con enteros es similar a C. No ahondaremos mucho sobre el tema ya que resulta trivial las operaciones entre números. En la tabla 12 vemos los operadores aritméticos

En el código 81 vemos algunos ejemplos:

OPERADOR	OPERACIÓN
+	Suma
-	Resta
*	Multiplicación
/	División
%	Módulo

Table 12: Operadores Aritméticos.

Listing 81: operaciones.cpp

```
long width {4};
long length {5};
long area { width * length };            // 20
long perimetro {2*width + 2*length};     // 18

long perimetro2{ (width + length) * 2 };  // 18
```

7.2.1.5 *Asignaciones*

Al igual que en el caso anterior, las operaciones de asignación son similares al como en que usabamos en C. Veamos ejemplos en: 82

Listing 82: asignacion.cpp

```
long perimetro {};
// ...
perimetro = 2 * (width + length);

int a {}, b {}, c {5}, d{4};
a = b = c*c - d*d;

int y {5};
y = y + 1;
```

Unido a esto, tenemos la opcion las operaciones con *op*= como vemos en los ejemplos 83

Listing 83: operAsignacion.cpp

```
int cantidad = cantidad % 2;

int nuevaCantidad %= 2; // es equivalente a la linea anterior

//Algunas otras opciones
```

```
x *= y + 1;

//su equivalente seria

x = x * (y + 1);
```

En la tabla 13 vemos los distintos *op*=.

Operacion	Operador	Operacion	Operador
suma	+=	and bit a bit	&=
resta	-=	or bit a bit	|=
multiplicación	*=	or exclusivo bit a bit	^=
división	/=	corrimiento a izquierda	<<=
modulo	%=	corrimiento a derecha	>>=

Table 13: operadores y asignacion.

7.2.1.6 *Incrementando y decrementando*

Vimos el uso del operador += o -=. Tenemos dos operadores más que se tienen que tener en cuenta: ++ y −−

Veamos formas equivalentes en el código 84

Listing 84: incremento.cpp

```
int count {5};
count = count + 1;
count += 1;
++count;
```

el incremento se realiza de 1 en 1. La última notación es mas concisa si lo que se busca es solo el incremento.

En el siguiente caso expresado en el código 85 vemos una operación, hay que tener en cuenta que operadores de incremento y decremento se ejecutan antes que cualquier otro operador aritmético binario en una expresión. Por lo tanto, si count tenia un valor de 5, el recuento se incrementará a 6 y luego este valor se utilizará en la evaluación de la expresión a la derecha de la asignación. Por lo tanto, al total se le asignará el valor 12.

Listing 85: incrementoPresedencia.cpp

```
total = ++count + 6;
```

Si tomamos la forma postfija del operador, incrementa a la variable despues de que su valor se usa en el contexto. En el ejemplo 86 vemos opciones equivalentes para entender esto:

Listing 86: incrementoPostfijo.cpp

```
total = count++ + 6;

// es equivalente a

total = count + 6;
++count;
```

Si bien el resultado no varía, hay que tener cierto cuidado con la escritura, *a++ + b* es equivalente a *a+++b*, aunque es factible que en el último caso, el compilador realice *a + ++b*, sería mas claro *total = 6 + count++* o alternativamente usando parentesis: *total = (count++) + 6;*

La observación puntual que se debe hacer, además del uso de parentesis, es la presedencia de signos y operadores y que implica el post y pre fijo del incremento y decremento. Veamos los siguientes ejemplos, tendremos en consideración que el valor inicial en casa cálculo de la variable *cuenta* es 5

total = - -cuenta + 6

El resultado que obtenemos es 10. Ahora, si consideramos la siguiente expresión:

total = 6 + cuenta - -

total tendrá un valor de 11.

¿Qué ocurre? Veamos paso a paso como se evaluan la expresión y sus componentes internas:

- Valor inicial: cuenta = 5, total = 0
- Evalua la ecuación total = 6 + cuenta - -
- Reemplaza el valor de cuenta y suma al 6 para setear total
- Decrementa cuenta

El decremento es la ultima operación que se ejecuta, posterior a todas las otras, si viera el valor de *cuenta* al final, sería 4. En el código 87 y su salida en pantalla:

Listing 87: decrementoPostfijo.cpp

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


int main(){
```

```
    int cuenta = 5;

    int total = 6 + cuenta--;

    cout<< "6 + cuenta —— = " << total << " y cuenta = " << cuenta
         <<endl;

    return 0;
}
```

```
$ g++ -std=c++17 -ooperadores operatoria.cpp
$ ./operadores
6 + cuenta-- = 11 y cuenta = 4
```

Algunos de estas declaraciones no están definidas en el estándard, esto no significa que no compile o se resuelve, solo que, el resultado puede no ser el esperado. Por ejemplo:

*total = ++count * 3 + count++ * 5*

Al ejecutarlo, el resultado es 48. Pero como hemos dicho, cuando las variables son usadas mas de una vez con operadores incrementales pueden dar resultados inesperados. Otro caso es el siguiente:

k = k++ + 1;

Aquí k es incrementada en el mismo momento que se asigna lo que hace que su valor se modifique dos veces, al ejecutarla, con un valor inicial de 5, el resultado fue 6. C++ 17, estableció una regla para casos asi, en el que todos los efectos secundarios del lado derecho de una tarea (y esto incluye asignaciones compuestas, incrementos y decrementos) están completamente comprometidos antes de evaluar el lado izquierdo y la asignación real. Sin embargo, no establece mayor presición sobre lo bien o mal definida la expresión.

Es importante y aconsejable el modificar una variable solo una vez como resultado de evaluar una sola expresión y acceda al valor anterior de la variable sólo para determinar su nuevo valor, es decir, no intente leer una variable nuevamente después de haber sido modificado en la misma expresión.

7.2.2 *Números flotantes*

Tenemos tres posibles tipos de datos de punto flotante, como vemos en el detalle 14

Tipo	Descripción	Precisión
float	Valores de precision simple	7
double	Valores de doble precision	15
long double	Valores de precision doble extendida	18-19

Sigue en la página siguiente.

Tipo	Descripción	Precisión

Table 14: Tipos de punto flotante.

Es importante recordar que no se usa el modificador *signed* o *unsigned*, los tipos de punto flotante son siempre *signed*.

El manejo de estos tipos no difiere de lo que hemos visto, vemos unos ejemplos en 88

Listing 88: initFloating.cpp

```
const double pi {3.141592653589793};
double a {0.2};
double z {9};
double volume {};
volume = pi*z*z*a;
```

Hay ciertas limitaciones al trabajar con tipos de punto flotante. Los errores que pueden surgir son:

- Muchos valores decimales no se conviernten exactamente a valores binarios de punto flotante, aunque sean errores pequeños, pueden terminar amplificados en los cálculos que se hagan.
- Al tomar la diferencia de valores casi idénticos, se perderá la precisión
- Trabajar con valores que difieren en varios órdenes de magnitud puede generar errores.

Luego, debemos añadir resultados inválidos o no calculables. El estándar de punto flotante IEEE define valores especiales que tienen una mantisa binaria de todos ceros y un exponente de todos los unos para representar +infinito o -infinito, según el signo. cuando divides un valor positivo distinto de cero por cero, el resultado será +infinito, y dividir un valor negativo por cero resulta en -infinito.

Otro valor especial definido en el estándar es el *not-a-number*, que abreviaremos como *NaN*, serán los resultados del cero dividido cero, entre otras operaciones por ejemplo. En el código 89 vemos algunas de estas operaciones.

Listing 89: infinity.cpp

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

int main(){

double a{ 1.5 }, b{}, c{};
double result { a / b };
```

```
std::cout << a << "/" << b << " = " << result << std::endl;
std::cout << result << " + " << a << " = " << result + a << std::
    endl;
result = b / c;
std::cout << b << "/" << c << " = " << result << std::endl;


};
```

La salida en consola es:

```
$
$ ./operadores
1.5/0 = inf
inf + 1.5 = inf
0/0 = -nan
```

7.2.3 *Formateando la salida*

Sirven para cambiar el formato de lo que escribimos por consola. Presentamos algunos de los mas comunes en la tabla 15

Tipo	Descripción
std::fixed	Output floating-point data in fixed-point notation.
std::scientific	Output all subsequent floating-point data in scientific notation, which always includes an exponent and one digit before the decimal point.
std::defaultfloat	Revert to the default floating-point data presentation.
std::dec	All subsequent integer output is decimal.
std::hex	All subsequent integer output is hexadecimal.
std::oct	All subsequent integer output is octal.
std::showbase	Outputs the base prefix for hexadecimal and octal integer values. Inserting std::noshowbase in a stream will switch this off.
std::left	Output is left-justified in the field.
std::right	Output is right-justified in the field. This is the default.

Table 15: Formateando la salida.

Las mencionadas, solo se invocan, estan accesibles mediante la libreria *iostream* y no reciben parametros. *iomanip* nos brinda algunas opciones mas como podemos ver en la tabla 16

Tipo	Descripción
std::setprecision(n)	Establece la precisión o los n dígitos decimales
std::setw(n)	Establece el ancho del campo de salida en n caracteres, pero solo para los siguientes datos de salida. La proxima salida vuelve al ancho estandar
std::setfill(ch)	Cuando el ancho del campo tiene más caracteres que el valor de salida, el exceso. Los caracteres en el campo serán el carácter de relleno predeterminado, que es un espacio. Este establece el carácter de relleno en ch para todas las salidas posteriores.

Table 16: Formateando la salida.

7.2.3.1 *Mezclando y convirtiendo tipos*

Tecnicamente, toda operación aritmética requiere que ambos operandos sean del mismo tipo. Aunque esto no siempre es el caso, ya que el compilador puedo convertir un operando al tipo del otro, a esto se lo llama *conversión implicita*. Esto significa que el compilador tomará a la variable del tipo mas restringido para convertirla en la de mayor rango. En el ejemplo 90 podemos verlo:

Listing 90: convImplicit.cpp

```
unsigned int numero{};
const unsigned int factor{12};

double prom /= factor;

double area_pond{numer * prom}; //numero es convertido a double
```

Como se dijo, elige los operandos de rango mas limitado para convertirlo a los mas grandes, el ranking de tipos seria el siguiente, de mayor a menor:

1. long double
2. double
3. float
4. unsigned long long

5. long long
6. unsigned long
7. long
8. unsigned int
9. int

Se tiene que tener en cuenta que el resultado de restas de enteros unsigned pueden envolver numeros positivos muy grandes que suelen llamarse *underflow*. El fenómeno inverso se llama *overflow*, sumar un unsigned char cuyo valor es 253 y 5, no nos daría 258, sino 2, o 258 modulo 256.

En algunos casos, principalmente en asignaciones directas o inicializaciones, las conversiones pueden requerirse en forma explicita como vemos en 91

Listing 91: convExplicit.cpp

```
int y{};
double z{5.0};

y = z; // requiere del cast explicito
```

Para esta conversión, usaremos *static_cast<type_to_convert_to>(expresion)*; como vemos en 92

Listing 92: cast.cpp

```
double valor1{10.9};
double valor2{15.6};

int valor3{static_cast<int>(valor1) + static_cast<int>(valor2)};
```

Como es sabido, C++ no cambia la filosofía de este tipo de conversión, utilizará el *truncado* para inicializar al intero, que terminará quedando con un valor de 25 en caso del ejemplo citado.

C++ también admite el cast en la antigua forma de C, por lo general suele ser el modo mas usado, pero no porque la conversión explicita sea nueva, sino simplemente por uso y costumbre. Lo recomendado es realizar la conversión con la fución de C++.

7.2.3.2 *Buscando limites*

Encontrar los limites altos y bajos de un tipo puede resultar importante. En el código ejmplo 93 vemos el modo de obtenerlos

Listing 93: limites.cpp

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

int main()
{
cout << "El rango del tipo short es de "
            << numeric_limits<short>::min() << " a "
            << numeric_limits<short>::max()  << endl;
cout << "El rango del tipo int es de "
            << numeric_limits<int>::min() << " a "
            << numeric_limits<int>::max() << endl;
cout << "El rango del tipo long es de "
            << numeric_limits<long>::min()<< " a "
            << numeric_limits<long>::max() << endl;
cout << "El rango del tipo float is desde "
            << numeric_limits<float>::min() << " a "
            << numeric_limits<float>::max() << endl;
cout << "El rango positivo del tipo double es desde "
            << numeric_limits<double>::min() << " a "
            << numeric_limits<double>::max() << endl;
cout << "El rango positivo del tipo long double es desde "
            << numeric_limits<long double>::min() << " a "
            << numeric_limits<long double>::max() << endl;
}
```

Al ejecutarlo:

```
$g++ -std=c++17 -olimites limites.cpp
$ ./limites
El rango del tipo short es de -32768 a 32767
El rango del tipo int es de -2147483648 a 2147483647
El rango del tipo long es de -9223372036854775808 a 9223372036854775807
El rango del tipo float is desde 1.17549e-38 a 3.40282e+38
El rango positivo del tipo double es desde 2.22507e-308 a 1.79769e+308
El rango positivo del tipo long double es desde 3.3621e-4932 a 1.18973e+4932
```

7.3 USANDO CARACTERES

Las variables de tipo *char* son usadas para guardar un simple caracter y por lo tanto ocupan 1 byte. C++ no especifica la codificación de dicho caracter, en principio lo delega al compilador y usualmente es ASCII.

Los ejemplos 94 vemos como definir una variable de tipo *char*.

Listing 94: letras.cpp

```
char letra; //sin inicializar - el valor tiene basura
char si{'Y'}; no{'N'}; //inicializado con literales
char l{33}; //inicializado con '!'
```

El caso mas distinguido en el ejemplo es el último, vemos que el tipo caracter es numerico, por lo tanto podemos realizar estas operaciones como en el ejemplo 95

Listing 95: letrasOper.cpp

```
char l{'A'};
char letra{l + 5}; //la letra es 'F'
++l; //l ahora es 'B'
l += 3;  //l ahora es 'E'
```

La salida en pantalla no resulta una dificultad, vemos una posibilidad en el ejemplo 96

Listing 96: Salida en pantalla de char

```
std::cout << "l es '" << ch
          << "' y su codigo numerico " << std::hex << std::
              showbase
          << static_cast<int>(ch) << std::endl;
```

Esto nos mostraría lo siguiente:

```
l es 'E' y su codigo numerico 0x45
```

7.4 EL TIPO *auto*

Se usa la palabra reservada *auto* para indicar al compilador que deduzca al tipo de dato. en el ejemplo

Listing 97: Declaracion de auto

```
auto m {10};       // m es de tipo int
auto n {200UL};    // n tiene un unsigned long
auto pi {3.14159}; // pi tiene un tipo double

auto list = {1, 2, 3};   // list tiene un tipo std::
    initializer_list<int>
```

Vamos a ver algunas diferencias en el uso del auto, su declaracion, inicialización y asignación ya que la deducción de la que hablamos cambio con el C++17. Hay que tener cuidado cuando se usa auto. En el código

Listing 98: Diferencias con estandares en uso de auto

```
/* C++11 and C++14 */
```

```
auto i {10};              // i has type std::initializer_list<int
    > !!!
auto pi = {3.14159};      // pi has type std::initializer_list<
    double>
auto list1{1, 2, 3};      // list1 has type std::initializer_list
    <int>
auto list2 = {4, 5, 6};   // list2 has type std::initializer_list
    <int>

/* C++17 and later */
auto i {10};              // i has type int
auto pi = {3.14159};      // pi has type std::initializer_list<
    double>
auto list1{1, 2, 3};      // error: does not compile!
auto list2 = {4, 5, 6};   // list2 has type std::initializer_list
    <int>
```

7.5 ENUMERACIONES

A veces se necesitan variables que tengan un conjunto limitado de valores que puedan ser referenciados por su nombre, ejemplos, dias de la semana, meses, etc. Esta posibilidad la dan los *enum*, un tipo de datos de enumeración. En el código 99 vemos como se puede declarar.

Listing 99: enum

```
enum class Dia {Lunes, Martes, Miercoles, Jueves, Viernes, Sabado
    , Domingo};
```

Esto define un tipo de dato enumerador llamado Día, y las variables de este tipo solo pueden tener valores de el conjunto que aparece entre las llaves, de lunes a domingo. Si se intenta establecer una variable de tipo Día en un valor que no es uno de estos valores, el código no se compilará. Los nombres simbólicos entre llaves son llamados enumeradores.

Como vemos, fuera de la pequeña diferencia de sintaxis, hasta aquí funciona igual que en C.

A cada enumerador se le asignará un entero, comenzando por el 0, de esa forma: Lunes sería igual a 0 y viernes igual a 6. En 100 vemos una forma de uso.

Listing 100: Asignar a variable

```
Dia hoy {Dia::Martes};
std::cout << "Hoy es " << static_cast<int>(hoy) << std::endl;
```

La salida en pantalla sería: "Hoy es 1".

Con respecto a los valores,en 101 podemos ver diferentes formas de romper con la asignación automatica que mostramos antes.

Listing 101: Asignar valores del enum

```
/* Aqui Lunes tiene valor 1 y Domingo tendra valor 7 */
enum class Dia {Lunes = 1, Martes, Miercoles, Jueves, Viernes,
    Sabado, Domingo};

/* Aqui Lunes tiene valor 3, viernes llegaria al 7 y Sabado 1 y
    Domingo tendra valor 2 */
enum class Dia {Lunes = 3, Martes, Miercoles, Jueves, Viernes,
    Sabado = 1, Domingo};

/*No es necesario que las claves sean unicas*/
enum class Dia {Lunes = 1, Lun = 1,  Martes,  Miercoles, Jueves,
    Viernes, Sabado, Domingo};

/*Podemos definir enumeradores en termino de un enumerador previo
    */
enum class Dia { Lunes,                    Lun = Lunes,
                 Martes  = Lunes + 2,      Mar = Martes,
                 Miercoles = Martes + 2,   Mier = Miercoles,
                 Jueves = Miercoles  + 2,  Jue = Jueves,
                 Viernes   = Jueves + 2,   Vie = Viernes,
                 Sabado = Viernes  + 2,    Sab = Sabado,
                 Domingo   = Sabado + 2,   Dom = Domingo
               };
```

Como opción se podría realizar una declaracion como se muestran en 102

Listing 102: Signos de puntuación

```
enum class Punctuacion : char {Coma  = ',', Exclamacion = '!',
    Pregunta = '?'};

std::cout << "Hola" << static_cast<char>(Punctuacion::Exclamacion
    )
    << " Todo bien" << static_cast<char>(Punctuacion::Pregunta)
        << std::endl;
```

Vemos que se debe hacer un cast a char, esto nos indica que cuando se devinieron los enumeradores se uso el valor ASCII del caracter.

7.6 ALIAS PARA TIPOS DE DATOS

Muchas veces, por conveniencia en la descripción o para simplificar la declaración, es mas práctico definir alias a determinados tipos de datos.

C++ tiene una sintaxis para esto mismo como vemos en 103

Listing 103: Alias para tipos de datos

```
using BigOnes = unsigned long long;    // Define a BigOnes como
    alias

BigOnes mynum {};                      // Define & inicializa un
    tipo unsigned long long
```

Vemos que la forma es intuitiva.

Igualmente C++ sigo permitiendo la sintaxis proveniente de C como vemos en el ejemplo 104

Listing 104: Vieja sintaxis para el alias

```
typedef unsigned long long BigOnes;    // Define BigOnes como
    alias
```

Esto visto de este modo puede parecer innecesario, pero si lo aprovechamos para tipos complejos nos puede resultar de mucha utilidad, más adelante veremos los contenedores, ahora simplemente nos servirá de ejemplo el código 105

Listing 105: Alias de dato complejo

```
using PhoneBook = std::map<std::shared_ptr<Contact>, std::string
    >;
```

8

CADENAS DE CARACTERES

Sabemos que una cadena es una secuencia de caracteres, en este apartado veremos la clase **string** de la biblioteca estándar, la cual proporciona para poder manipularlos: asignar, concatenar, busqueda de subcadenas, etc.

C++ heredó de C la noción de la cadena como un array de caracteres terminados en cero.

8.1 STRING

En primera instancia veremos el manejo básico del objeto **string** luego, profundizaremos en las clases y su naturaleza.

8.1.0.1 *Construcción*

Listing 106: declaracion.cpp

```
void f(char* p, vector<char> v)
{
        string s0;          // la string esta vacia
        string s00 = "";    // la string esta vacia

        string s1 = 'a';    // error!!!!  no se puede convertir
            un caracter en string
        string s2 = 7;      // error!!!! no hay conversion de int
             a string
        string s3(7);       // error!!! no hay constructor que
            tome un int

        string s4(7, 'a');  // crea un string con 7 copias de a,
            o sea, "aaaaaaa"
        string s5("Frodo"); // copia de "Frodo"
        string s6 = s5;     // copia de s5

        string s7(s5, 3, 2);//s5[3] y s5[4] o sea copia "do"
        string s8(p+7, 3);  //p[7], p[8] y p[9]
        string s9(p, 7, 3); //string(string(p), 7, 3) no es
            recomendable
        string s10(v.begin(), v.end()); //copiar todos los
            caracteres de v

}
```

Al igual que C, los caracteres se cuentan a partir de 0, y la cadena tendrá entonces de 0 a **lengh()** −1.

lengh() es sinónima a **size()** dan el largo de la cadena sin contar el terminador.

En general, la manipulación mas común como crear, copiar, comparar, no generan mayor problemas o errores, pero en el trabajo con ***substrings*** o de accesos determinados ***at()*** es factible que surjan errores, *C* comprueba y lanza ***out_of_range*** si intentamos acceder mas alla de la cadena. Ejemplos

Listing 107: cadena-init.h

```
void f()
{
    string s = "Snobol4";
    string s2(s, 100, 2);    //posicion del caracter mas alla del
         final
                             //de la cadena: lanza error
                                 out_of_range()
    string s3(s, 2, 100);    //Recuento de caracteres demasiado
        grande:
                             //equivalente a s3(s, 2, s.size()-2)
    string s4(s, 2, string::npos); //los caracteres a partir de s
        [2]

}
```

Como se ve, se no hay que indicar posiciones demasiado grandes, pero si es posible los recuentos grandes. el ejemplo es válido para el uso de números negativos, ejemplo:

Listing 108: cadenas-rangos.h

```
void g(string s)
{
    string s5(s, -2, 3);    //posicion grande!! lanza
        out_of_range();

    string s6(s, 3, -2);    //recuento de caracteres grande! bien
        !

}
```

El constructor de **string** cuando recibe el valor de posición o recuento, este es de tipo *size_type* el cual es de tipo **unsigned** y la conversión de esto es que sera un positivo muy grande.

Listing 109: cadena-ejemplo.h

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

int main(){

    string::size_type st1 = 5;
    string::size_type st2 = -1;

    cout << " st1 tiene un valor de: " << st1 << " y st2 de: " <<
         st2 << endl;
}
```

```
$ g++ -o prueba_size size\_type.cpp
$ ./prueba\_size
st1 tiene un valor de: 5 y st2 de: 18446744073709551615
$
$
```

8.1.1 *Asignaciones*

La asignaciones para cadenas funcionan de la siguiente forma:

Listing 110: cadena-asig.h

```
void g()
{
    string s1 = "Knould";
    string s2 = "Tod";

    s1 = s2;        //dos copias de "Tod"
    s2[1] = 'u';    //s2 es "Tud"

}
```

También está soportada la asignación de un simple caracter, pero no en el constructor. Ejemplo:

Si bien el segundo ejemplo no parece muy útil, no es tan extraño anexar un caracter: $s+='a'$.

8.1.2 *Comparaciones*

Las cadenas pueden compararse con cadenas de su mismo tipo y con arrays de caracteres.

Listing 111: cadena-asig-char.h

```
void f()
{
    string s = 'a';    //Error!!!!
    s = 'a';           //Esto es correcto
    s = "a";           //Esto es correcto.

}
```

La función mas común es **compare()**, la cual correspondería usarse de la siguiente forma **s.compare(s2)** retornando o en caso que sean iguales, un número negativo si *s* está lexicográficamente antes que *s2* y un número positivo si es el caso contrario.

Como vemos, compare() también toma como argumento size_type **pos** y size_type **n**, este último es la cantidad de posiciones que se compararán, y el primer argumento desde que posición comenzará. Como vemos en el ejemplo anterior, se compara 3 caracteres de s con la totalidad de s2.

```
$ ./cadenas_comparacion
usamos compare...
son distintas
usamos == ...
son distintas
usamos compare para comparar los 3 primeros
 caracteres con s2
son diferentes
usamos compare para comparar los 3 primeros
 caracteres con s3
son iguales
$
```

Por una cuestión de brevedad, omitiremos los ejemplos de uso con arryas o char*, pero son de uso similar, solo hay que tener en cuenta que se use correctamente el terminador de las cadenas.

Hay que tener en cuenta que la función en cuestión distingue las mayúsculas de las minúsculas. Para evitar esto tenemos un par de opciones, armar un método propio utilizando **toupper()** o utilizar lexicographical_compare()

lexicographical_compare() tiene dos prototipos, el primero tomando cuatro parámetros, inicio y fin de los dos strings a comparar, y el segundo con cinco parámetros: los cuatro anteriores y una función binaria que acepta dos argumentos de tipo iteradores y retorna verdadero o falso, la comparación es realizada usando al operador $<$ .Ejemplo de uso:

```
$  g++ lexicographical.cpp -o lexicographical
$
$ ./lexicographical
Comparando s y s2 lexicographicamente (s < s2):
```

Listing 112: cadena-comp.h

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

using namespace std;

int main ()
{
    string s = "abcde", s2 = "abcdf", s3 = "abc";

    cout << "usamos compare..." << endl;
    if (!s.compare(s2)) // como retorna 0 si son iguales, usamos
        el !
        cout << "son iguales" << endl;
    else
        cout << "son distintas" << endl;

    cout << "usamos == ..." << endl;
    if (s == s2)
        cout << "son iguales" << endl;
    else
        cout << "son distintas" << endl;
    cout << "usamos compare para comparar los 3 primeros\n"
              << "caracteres con s2" << endl;
    if (s.compare(0,3,s2) == 0)
        cout << "son iguales" << endl;
    else
        cout << "son diferentes\n";
    cout << "usamos compare para comparar los 3 primeros\n"
              << "caracteres con s3" << endl;
    if (s.compare(0,3,s3) == 0)
        cout << "son iguales" << endl;
    else
        cout << "son diferentes\n";


    return 0;
}
```

```
usando la comparacion por default (operador <): true
Usando la funcion mi_comparador: false
$
```

Listing 113: cadena-cmp-case-sensit.h

```
int cmp_nocase(string s, string s2)
{
    string::const_iterator p = s.begin();
    string::const_iterator p2 = s2.begin();

    while(p != s.end() && p2 != s2.end()){
        if (toupper(*p) != toupper(*p2) )
                return (toupper(*p) < toupper(*p2)) ? -1 : 1;

        ++p;
        ++p2;
    }

    return (s2.size() == s.size()) ? 0 : (s.size() < s2.size()) ?
         -1 : 1;
}
```

Listing 114: cadena_lexic.h

```
// lexicographical_compare

#include <iostream>     // std::cout
#include <algorithm>    // std::lexicographical\_compare
#include <cctype>       // std::tolower

using namespace std;


bool mi_comparador (char c1, char c2)
{
    return tolower(c1) < tolower(c2);
}

int main () {
    string s = "Apple";
    string s2 = "apartment";
    string::iterator i = s.begin();
    string::iterator e = s.end();
    string::iterator i2 = s2.begin();
    string::iterator e2 = s2.end();

    cout << boolalpha; // setea la salida en pantalla para
        mostrar true o false;

    cout << "Comparando s y s2 lexicographicamente (s < s2):\n";

    cout << "usando la comparacion por default (operador <): ";
    cout << lexicographical_compare(i,e,i2,e2) << endl;


    cout << "Usando la funcion mi_comparador: ";
    cout << std::lexicographical_compare(i,i+5,i2,i2+9,
        mi_comparador) << endl;

    return 0;
}
```

9 LA CLASE VECTOR

Dentro de la librería estándar de C++ ya encontramos varias estructuras de datos básicas las cuales nos resultarán de mucho interés y utilidad.

A diferencia de C, que nos veíamos obligados a generar nuestas propias estructuras abstractas, las de C++ al estar implementadas con *templates*, tienen un grado de polimorfismo que suman mucha potencia de uso.

vector nos permitirá disponer de un arreglo dinámico, sin tener que preocuparnos por el tamaño, teniendo asi una ventaja del tradicional *array* heredado de C.

9.1 VECTOR.H

Será imprescindible el incluir a *vector.h* en la cabecera de nuestro código.

9.1.1 *Construcción*

```
$ c++ -o initVector initVector.cpp
$
$ ./initVector
6
3.1416
3.1416
3.1416
3.1416
3.1416
3.1416

$
```

En las líneas 13 y 14 que están comentadas, vemos que el comentario dice error, efectivamente si quisieramos compilar con dichas lineas activas veriamos el siguiente mensaje en consola:

```
$ c++ -o initVector initVector.cpp
initVector.cpp: In function 'int main()':
initVector.cpp:12:14: error: no match for 'operator=' (operand types are 'std::
     vector<double>' and 'int')
     Vector_1 = 10;
              ^
initVector.cpp:12:14: note: candidate is:
In file included from /usr/include/c++/4.9/vector:69:0,
                 from initVector.cpp:2:
/usr/include/c++/4.9/bits/vector.tcc:167:5: note: std::vector<_Tp, _Alloc>& std::
     vector<_Tp, _Alloc>::operator=(const std::vector<_Tp, _Alloc>&) [with _Tp =
     double; _Alloc = std::allocator<double>]
     vector<_Tp, _Alloc>::
     ^
```

Listing 115: initVector.cpp

```
//initVector.cpp
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;

int main()
{
    //vector sin inicializar ni indicar size
    vector<double> Vector_1;
//    Vector_1 = 10;    // Error!!!
//    Vector[0]= 1;     // Error!!!

    vector<int> Vector_3(10);

    //vector con size 5 y componentes iniclizadas
    vector<double> Vector_2(6,3.1416);

    // Vemos el size del vector con .size()
    cout << Vector_2.size() << endl;

    //mostrar las componentes con un ciclo

    for(int i=0; i<Vector_2.size() ;i++)
    { //con el metodo .size() se obtiene el size del vector
        cout << Vector_2[i] << endl;
    }

    cout<<endl;
}
```

```
/usr/include/c++/4.9/bits/vector.tcc:167:5: note:   no known conversion for
     argument 1 from 'int' to 'const std::vector<double>&'
initVector.cpp:13:5: error: 'Vector' was not declared in this scope
     Vector[0]= 1;
     ^


$
```

La forma de inicializar al vector es como se muestra en las lineas siguientes a los comentarios de error.

9.1.2 *Copia de un vector*

Estas estructuras, a pesar de ser complejas, poseen sobrecargados los operadores, eso implica que están redefinidos para que sean usados de modo transparentes, directos.

```
$ c++ -o copiaVector copiaVector.cpp
```

Listing 116: copiaVector.cpp

```
//copiaVector.cpp
#include <iostream>
#include <vector>

using namespace std;

int main()
{
        //
    // copia de vectores...
    // declaro e inicializo un vector con valores en 1
    vector<int> vInt_1(20, 1);

    // No se inicializa
    vector<int> vInt_2;

    cout<< "vInt_2 antes: " << vInt_2.size() <<endl;

    // Uso el igual para asignar, para hacer la copia
    vInt_2 = vInt_1;

    cout<< "vInt_2 despues: " << vInt_2.size() <<endl;
    for(int i=0; i < vInt_2.size() ;i++)
    {
        cout << "vInt_2 = " << vInt_2[i] << endl;
    }
}
```

```
$
$./copiaVector
vInt_2 antes: 0
vInt_2 despues: 20
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
```

```
$
```

9.1.3 *Insertando y eliminando datos*

Los ejemplos que se mostrarán no son exhaustivos, pero darán una base de como se manipulan los datos en el vector.

Como se ve, en los comentarios al codigo se hace referencia o se explica como proceder o las consecuencias de lo que se realiza. Al ejecutar el código vemos:

```
$ c++ -o push_popVector push_popVector.cpp
$
$./push_popVector
Size de vInt_2 = 10
Nuevo size de vInt_2 = 11
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 5
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
Size de vInt_2 = 10
El nuevo size es: 5
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
vInt_2 = 1
El size de vInt_3 es 10
--------------
vInt_3 = 1
vInt_3 = 2
vInt_3 = 3
vInt_3 = 4
vInt_3 = 5
vInt_3 = 6
vInt_3 = 7
vInt_3 = 8
vInt_3 = 9
vInt_3 = 10
--------------
vInt_3 = 1
vInt_3 = 2
vInt_3 = 3
vInt_3 = 7
vInt_3 = 8
vInt_3 = 9
```

```
vInt_3 = 10
El size despues del borrado de vInt_3 es 7
vInt_3 = 1
vInt_3 = 2
vInt_3 = 3
vInt_3 = 7
vInt_3 = 8
vInt_3 = 9
vInt_3 = 10

vInt_3 = 1
vInt_3 = 2
vInt_3 = 3
vInt_3 = 3
vInt_3 = 7
vInt_3 = 8
vInt_3 = 9
vInt_3 = 10

vInt_3 = 1
vInt_3 = 2
vInt_3 = 3
vInt_3 = 3
vInt_3 = -1
vInt_3 = -1
vInt_3 = -1
vInt_3 = 7
vInt_3 = 8
vInt_3 = 9
vInt_3 = 10

$
```

9.1.4 *Función sort*

Para el uso de ciertas funciones, debemos incluir la lib *algorithm*

```
$ c++ -o sortVector sortVector.cpp
$
$./sortVector
El tamano de vInt_3 es 10
--------------
vInt_3 = 11
vInt_3 = 10
vInt_3 = 9
vInt_3 = 8
vInt_3 = 7
vInt_3 = 6
vInt_3 = 5
vInt_3 = 4
vInt_3 = 3
vInt_3 = 2

vInt_3 = 2
vInt_3 = 3
vInt_3 = 4
vInt_3 = 5
vInt_3 = 6
vInt_3 = 7
vInt_3 = 8
vInt_3 = 9
vInt_3 = 10
vInt_3 = 11
Pablo Juan alicia
```

```
Juan Pablo alicia
$
```

9.1.5 *Iteradores*

Los iteradores son punteros. Están asociados a los contenedores y nos ayudarán en el momento de sus recorridos principalmente.

```
$ g++ -o itVector itVector.cpp
./itVector
El tamano de vInt_3 es 10
--------------
vInt_3 = 11
vInt_3 = 10
vInt_3 = 9
vInt_3 = 8
vInt_3 = 7
vInt_3 = 6
vInt_3 = 5
vInt_3 = 4
vInt_3 = 3
vInt_3 = 2
vInt_3[0] = 11

Ordenado a traves de iteradores
vInt_3 = 2
vInt_3 = 3
vInt_3 = 4
vInt_3 = 5
vInt_3 = 6
vInt_3 = 7
vInt_3 = 8
vInt_3 = 9
vInt_3 = 10
vInt_3 = 11

$
```

Listing 117: push_popVector.cpp

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> vInt_2(10,1);
    //push_back() es el que permite ingresar un nuevo elemento
    //al vector
    //
    cout << "Size de vInt_2 = " << vInt_2.size() << endl;

    vInt_2.push_back(5);

    cout << "Nuevo size de vInt_2 = " << vInt_2.size() << endl;

    for(int i=0; i < vInt_2.size() ;i++)
    {
        cout << "vInt_2 = " << vInt_2[i] << endl;
    }

    //pop_back() eliminamos el ultimo elemento
    //Esta funcion no lo retorna
    //
    vInt_2.pop_back();

    for(int i=0; i < vInt_2.size() ;i++)
    {
        cout << "vInt_2 = " << vInt_2[i] << endl;
    }

    cout << "Size de vInt_2 = " << vInt_2.size() << endl;
    vInt_2.resize(5);
    //ahora su tamano es
    cout<<"El nuevo size es: " << vInt_2.size()<<endl;

    //Esto ha provocado una perdida de informacion
    //
    for(int i=0; i < vInt_2.size() ;i++)
    {
        cout << "vInt_2 = " << vInt_2[i] << endl;
    }

    // Borrado de elementos
    //
    vector<int> vInt_3(10);

    for (int i = 0; i < 10; i++)
        vInt_3[i] = i+1;

    cout << "El size de vInt_3 es " << vInt_3.size() << endl;
    cout << "----------------" << endl;
    for(int i=0; i < vInt_3.size() ;i++)
    {
        cout << "vInt_3 = " << vInt_3[i] << endl;
    }
```

Listing 118: sortVector.cpp

```
\\sortVector.cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{

    vector<int> vInt_3(10);

    for (int i = 0, j = 10; i < 10; i++, j--)
        vInt_3[i] = j+1;

    cout << "El tamano de vInt_3 es " << vInt_3.size() << endl;
    cout << "----------------" << endl;
    for(int i=0; i < vInt_3.size() ;i++)
    {
        cout << "vInt_3 = " << vInt_3[i] << endl;
    }

    // Con los contenedores podemos usar una funcion
   // de la lib algorithm muy util. Antes hay que
   // incluir la lib #include<algortihm>
   //
   sort(vInt_3.begin(), vInt_3.end());

   cout << endl;
   for(int i=0; i < vInt_3.size() ;i++)
   {
       cout << "vInt_3 = " << vInt_3[i] << endl;
   }
   vector<string> Nombres;
   Nombres.push_back("Pablo");
   Nombres.push_back("Juan");
   Nombres.push_back("alicia");
   for(int i=0; i < Nombres.size() ;i++)
   {
       cout << Nombres[i] << " ";
   }
   cout << endl;

   sort(Nombres.begin(), Nombres.end());

   for(int i=0; i < Nombres.size() ;i++)
   {
       cout << Nombres[i] << " ";
   }
   cout << endl;
}
```

Listing 119: itVector.cpp

```
//itVector.cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
        vector<int> vInt_3(10);

    for (int i = 0, j = 10; i < 10; i++, j--)
        vInt_3[i] = j+1;

    cout << "El tamano de vInt_3 es " << vInt_3.size() << endl;
    cout << "----------------" << endl;
    for(int i=0; i < vInt_3.size() ;i++)
    {
        cout << "vInt_3 = " << vInt_3[i] << endl;
    }

   // Iteradores
   //
   // Son punteros
   vector<int>::iterator I;

   I = vInt_3.begin();

   cout << "vInt_3[0] = " << *I << endl;

    // como se ve, begin() retorna un iterador
    // entonces podemos hacer lo siguiente:
    //
    vector<int>::iterator I1, I2;
    I1 = vInt_3.begin();
    I2 = vInt_3.end();

    cout<< endl << "Ordenado a traves de iteradores" << endl;

    sort(I1, I2);

    for( ; I1 != I2 ;I1++)
    {
        cout << "vInt_3 = " << *I1 << endl;
    }

    return 0;
 }
```

10 ENTRADA Y SALIDA EN ARCHIVOS

10.1 C++ STREAMS

Antes de ver propiamente el manejo de entrada y salida por archivos, veremos conceptos basicos de los streams, ya que son estos los usados para esa función.

A simple vista *std::ostringstream* es una clase cuyos objetos parecen flujos de salidas o sea que se utiliza el operador « para escribir en ellos), pero en realidad ostringstream almacena resultados de la escritura y los proporciona en forma de flujo o stream.

Vemos un ejemplo breve en el código 120: *ej_ostringstream.cpp*

Listing 120: ej_ostringstream.cpp

```
#include <sstream>
#include <string>
using namespace std;

int main()
{
    ostringstream ss; //aqui se crea el objeto

    //Se lo manipula como un flujo normal
    ss << "the answer to everything is " << 42;

    //Se convierte en string
    const string result = ss.str();
}
```

Esto es principalmente útil cuando tenemos una clase para la cual se ha definido la serialización de flujo y para la cual queremos un forma de cadena. Por ejemplo, supongamos que tenemos alguna clase como en el ejemplo **??**: *ejemplo_foo.cpp*

Listing 121: ejemplo_foo.cpp

```
class foo
{
    // La definicion del objeto.
};

ostream &operator<<(ostream &os, const foo &f);

// Luego, para tener el string reprensando al objeto
```

```
// Declaramos al objeto
foo f;
// Podemos usar lo siguiente
ostringstream ss;
ss << f;
// Lo convertimos en un string regular
const string result = ss.str();
```

10.1.1 *Mostrando collecciones con iostream*

10.1.1.1 *Ejemplo básico*

std::ostream_iterator permite imprimir el contenido de un contenedor en un flujo de salida sin bucles explícitos.

El segundo argumento del constructor *std::ostream_iterator* establece el delimitador. Por ejemplo, el siguiente código:

Listing 122: iteracion.cpp

```
std::vector<int> v = {1,2,3,4};
std::copy(v.begin(), v.end(), std::ostream_iterator<int>(std::
    cout, " ! "));
```

La salida en pantalla seria:

```
1 ! 2 ! 3 ! 4 !
```

10.1.1.2 *Cast implícito*

11

STANDARD TEMPLATE LIBRARY

En 1980-1983 Bjarne Stroustrup (al igual que los creadores de C, trabajaba para laboratorios Bell de AT&T) crea una modificación de C, que el llamó C con clases. Tiempo después se propone el nombre de C++ (C incrementado) tomó ideas de SIMULA 67 (clases y funciones virtuales), de ADA (tipos genéricos y expresiones, de ALGOL 68 el poder declarar variables en cualquier punto del programa e ideas propias como la implementación de herencia multiple logrando combinar la potencia de la programación imperativa de C con la orientación a objetos.

En 1985 Stroustrup edita la primera edición de C++ programming languaje, el cual pasa a ser un standard de referencia (recién en 1990 se crearía el comite ANSI de C++).

Un hecho importante se dá en el año 1993 aproximadamente cuando Alexander Stepanov junto con Andrew Koening diseñan y desarrollan la ***Standard Template Library*** proporcionando a C++ de clases con contenedroes y algoritmos genéricos brindando al lenguaje una potencia única.

En 1998 el consorcio formaliza el ANSI C++ y la STL queda incorporada al lenguaje de manera formal.

11.1 LA ESTRUCTURA DE LA LIBRERÍA

La STL tiene cuatro componentes o conjuntos fundamentales de elementos que a sus vez tienen importantes componentes:

- Contenedores: son colecciones de objetos y tipos primitivos
  - Contenedores secuenciales:
    - Vector
    - List
    - Deque
    - arrays
    - forward_list
  - Contenedores adaptables
    - queue
    - priority_queue
    - stack
  - Contenedores asociativos ordenados

        - Set
        - multiset
        - map
        - multimap
    - Contenedores asociativos no ordenados
        - Unsorted_set
        - unsorted_multiset
        - unsorted_map
        - unsorted_multimap

- Algoritmos: para usar con los datos de las estructuras anteriores (ordenamiento, búsquedas, máximos, tamaños, etc.)
    - Sorting
    - Searching
    - Algoritmos varios (borrados, busquedas especiales, etc)
    - Algoritmos útiles para arreglos
    - Operaciones de particionamiento
    - Clase valarray (provee operaciones para arreglos)
- Funciones
    - functors
- Iteradores: punteros para recorrer o generar índices (avances, retrocesos, índices, etc.)
- pair

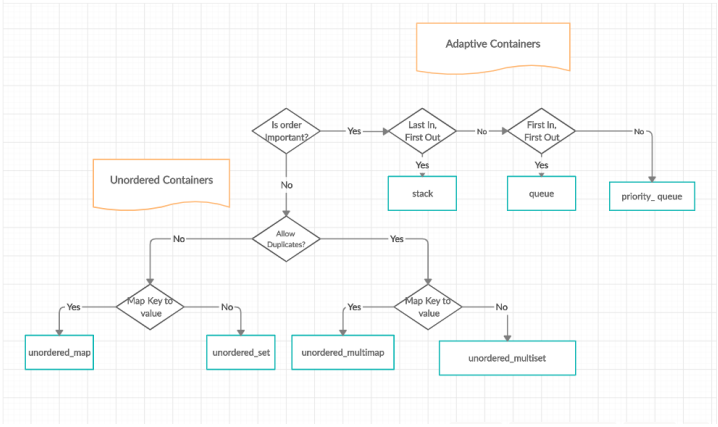


Figure 33: Mapa de contenedores adaptados

En las secciones siguientes haremos una descripción de los contenedores, por lo menos de los que considero mas importantes

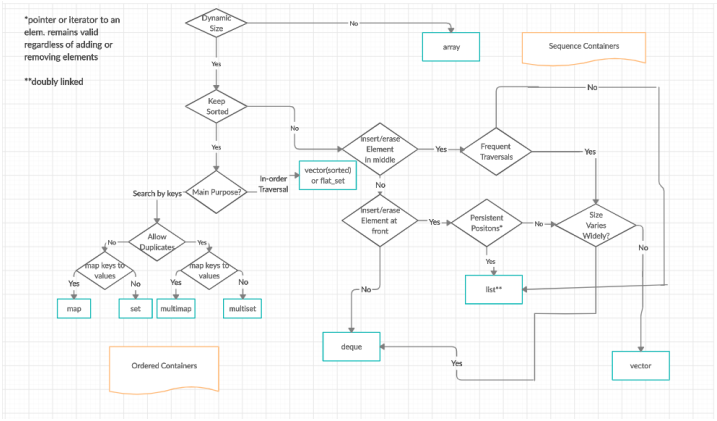


Figure 34: mapas de contenedores secuenciales

11.2 ITERADORES

Hemos visto un uso mínimo de ellos principalmente para el recorrido de las estructuras. Hay varios tipos definidos:

- ***Input Iterator***: para operaciones secuenciales de entrada.
- ***Output Iterator***: para operaciones secuenciales de salida.
- ***Iteradores de avance***: Ejemplo, lista enlazada.
- ***Iteradores bidireccionales***: navegan los datos uno a uno con capacidad de volver hacia atras. Ejemplo, lista doblemente enlazada.
- ***Iterador de acceso aleatorio***: accede a un elemento específico.

Al menos todos los contenedores soportan el iterador de avance. Cada contenedor soporta las operaciones de *begin()* y *end()*.

- begin() retorna el iterador de avance al primer elemento
- end() retorna un iterator especial, es el punto final del contendor, no se puede desreferenciar, es similar al uno del puntero NULL para distinguir al último elemento.

Cada contenedor tendrá definido sus iteradores, algunos comunes y otros particulares dependiendo la naturaleza de la estructura.

11.3 ALGORITMOS DE LA STL

<algorithm> es una colección de algoritmos para ser usados en rangos de datos. Recibirá dichos rangos como argumentos, está implementada en forma genérica por lo tanto admitirá a los contenedores de la STL.

El rango estará definido por iteradores indicando principio y fin. Esto implica que si paso los iteradores begin() y end() el último elemento está incluido en el rango, por lo tanto debo examinar hasta el elemento anterior del end (que ya sería el nulo).

Un ejemplo simple seria el siguiente 123: *sortVector.cpp*

Listing 123: sortVector.cpp

```
#include<iostream>
#include<algorithm>
#include<vector>
using namespace std;

int main (){

    vector<int>numeros;

    numeros.push_back(5);
    numeros.push_back(3);
    numeros.push_back(8);

    sort(numeros.begin(), numeros.end());

    for(auto x: numeros)
         cout<< x << endl;

    return 0;
}
```

```
$ g++ -oej_sort -std=c++11 -Wall ej_sort.cpp
$ ls
ej_sort  ej_sort.cpp
$ ./ej_sort
3
5
8
$
```

Aqui otro 124: *encuentraMap.cpp* algo mas complejo:

Listing 124: encuentraMap.cpp

```
#include<iostream>
#include<algorithm>
#include<map>
using namespace std;

int main (){

    map<int, string>ejemplo;

    ejemplo[0]="Pablo";
    ejemplo[1]="otro  Pablo";
```

```
    cout<< "Elemento\n";
    auto x = ejemplo.find(0);
    cout<< "Clave " << x->first << " Dato: " << x->second << endl
        ;

    return 0;
}
```

```
$ g++ -oej_map -std=c++11 -Wall ej_map.cpp
$ ./ej_map
Elemento
Clave 0 Dato: Pablo
```

11.4 CONTENEDORES SECUENCIALES

11.4.1 *vector*

En el capítulo 9 hemos visto como se declaran y utilizan, aqui repasaremos algunas cuestiones generales:

11.4.1.1 *Métodos que trabajan con Iteradores*

1. begin(): Retorna un puntero iterador al primer elemento del vector
2. end(): Retorna un puntero iterador al ultimo elemento del vector
3. rbegin(): Retorna un puntero iterador de reversa al último elemento. Es para poder moverse desde el último al primer elemento.
4. rend(): Retorna un puntero iterador al elemento que precede al primer elemento considerado como extremo inverso.
5. cbegin(): Retorno un puntero iterador constante al primer elemento del vector.
6. cend(): Retorna un puntero iterador constante al ultimo elemento del vector.
7. crbegin(): El puntero que retorna es identico al rbegin pero constante.
8. crend(): El puntero que retorna es idéntico al rend() pero constante.

Veamos el código 125: *iteradoresVector.cpp*

Listing 125: iteradoresVector.cpp

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> g1;

    for (int i = 1; i <= 5; i++)
        g1.push_back(i);

    cout << "Recorriendo de principio a fin: ";
    for (auto i = g1.begin(); i != g1.end(); ++i)
        cout << *i << " ";

    cout << "\nDe cbegin a cend: ";
    for (auto i = g1.cbegin(); i != g1.cend(); ++i)
        cout << *i << " ";

    cout << "\nDe rbegin a rend: ";
    for (auto ir = g1.rbegin(); ir != g1.rend(); ++ir)
        cout << *ir << " ";

    cout << "\nDe crbegin a crend : ";
    for (auto ir = g1.crbegin(); ir != g1.crend(); ++ir)
        cout << *ir << " ";

    return 0;
}
```

```
$ g++ -Wall -ovector_iterador vector_iteradores.cpp
$ ./vector_iterador
Recorriendo de principio a fin: 1 2 3 4 5
De cbegin a cend: 1 2 3 4 5
De rbegin a rend: 5 4 3 2 1
De crbegin a crend : 5 4 3 2 1
$
$
```

11.4.1.2 *Métodos relacionados con la capacidad*

1. size(): Retorna el número de elementos en el vector
2. max_size(): Retorna el máximo número de elementos que el vector puede tener
3. capacity(): Returna el tamaño del almacenamiento actualmente asignado al vector expresado en número de elementos

4. resize(n): Redimensiona al contenedor para guardar n elementos.
5. empty(); Retorna si el contenedor está vacio.
6. shrink_to_fit(): reduce la capacidad del contenedor para adaptarse a su tamaño y destruye todos los elementos restantes
7. reserve(): solicita al contenedor que reserve al menos la capacidad para contener n elementos.

En 126: *capacidadVector.cpp* vemos ejemplos de esto:

Listing 126: capacidadVector.cpp

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> g1;

    for (int i = 1; i <= 5; i++)
        g1.push_back(i);

    cout << size() : " << g1.size();
    cout << "\ncapacity() : " << g1.capacity();
    cout << "\nmax_size() : " << g1.max_size();

    // Redimensiona al vector a 4
    g1.resize(4);

    // muestra nuevo tamano resize()
    cout << "\nDespues del resize...";
    cout << "\nSize : " << g1.size();
    cout << "\nCapacity : " << g1.capacity();
    cout << "\nMax_Size : " << g1.max_size();

    g1.shrink_to_fit();

    // muestra valores despues del shrink()
    cout << "\nDespues del shrink...";
    cout << "\nSize : " << g1.size();
    cout << "\nCapacity : " << g1.capacity();
    cout << "\nMax_Size : " << g1.max_size();

    // Chequea si esta vacio o no
    if (g1.empty() == false)
        cout << "\nVector is not empty";
    else
```

```
        cout << "\nVector is empty";


    cout << "\nLos elementos son: ";
    for (auto it = g1.begin(); it != g1.end(); it++)
        cout << *it << " ";

    return 0;
}
```

```
$ g++ -ovector_capacity -Wall vector_capacity.cpp
$ ./vector_capacity
Size : 5
Capacity : 8
Max_Size : 2305843009213693951
Despues del resize...
Size : 4
Capacity : 8
Max_Size : 2305843009213693951
Despues del shrink...
Size : 4
Capacity : 4
Max_Size : 2305843009213693951
Vector is not empty
Vector elements are: 1 2 3 4
$
$
```

11.4.1.3 *Acceso a los elementos*

1. operador [x]: Retorna la referencia al elemento en la posición x.
2. at(x); Retorna la referencia al elemento en la posición x.
3. front(): Retorna una referencia al primer elemento del vector
4. back(): Retorna una referencia al último elemento del vector
5. data(): Retorna la dirección del puntero al arreglo en memoria usado internamente por el vector donde estan almacenados los elementos

Vemos en 127: *accesoVector.cpp* su uso:

Listing 127: accesoVector.cpp

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

int main()
{
    vector<int> g1;

    for (int i = 1; i <= 10; i++)
```

```
        g1.push_back(i * 10);

    cout << "\n Operador de referencia [g] : g1[2] = " << g1[2];

    cout << "\nat : g1.at(4) = " << g1.at(4);

    cout << "\nfront() : g1.front() = " << g1.front();

    cout << "\nback() : g1.back() = " << g1.back();

    // puntero al array
    int* pos = g1.data();

    cout << "\nprimer elemento por  " << *pos;
    return 0;
}
```

```
$ g++ -ovector_elementos -Wall vector_elementos.cpp
$./vector_elementos

 Operador de referencia [g] : g1[2] = 30
at : g1.at(4) = 50
front() : g1.front() = 10
back() : g1.back() = 100
primer elemento por  10
$
$
```

11.4.1.4 *Métodos modificadores de los datos del vector*

1. assign(): asigna un nuevo valor a un elemento del vector reemplazando a otro
2. push_back(): agrega un elemento al final del vector
3. pop_back(): se utiliza para quitar o borrar al ultimo elemento.
4. insert(): inserta un nuevo elemento antes del elemento de la posición indicada
5. erase(): borra elementos del contenedor de una posición o rango determinado
6. swap(): intercambia elementos de un vector a otro mientras sean del mismo tipo.
7. clear(): borra todos los elementos de un vector
8. emplace(): extiende al vector insertando un nuevo elemento en una posicion dada
9. emplace_back(): extiende al vector insertando un nuevo elemento al final.

FALTA COMPLETAR

11.5 ESTRUCTURAS ÚTILES

Mostraremos ejemplos de usos sin ahondar en detalles, pero mostrando usos de algunas estructuras que pueden darnos soluciones en el manejo de datos.

11.5.1 *Tuplas*

Ejemplo 128: *tuplas.cpp* muestra el uso:

Listing 128: tuplas.cpp

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

tuple<int, int, int, int> operaciones( int x, int y){
    return make_tuple(x+y, x-y, x*y, x/y);
}

int main (){

    tuple<int, int, int, int> resultado = operaciones (6, 2);

    cout<< "Resultado\n";
    cout<< "Suma: "              << get<0>(resultado) << endl;
    cout<< "resta: "                     << get<1>(resultado) <<
        endl;
    cout<< "Multiplicacion: "  << get<2>(resultado) << endl;
    cout<< "Division: "          << get<3>(resultado) << endl;

    return 0;
}
```

```
$ g++ -oej_tupla -std=c++11 -Wall ej_tupla.cpp
$ ./ej_tupla
Resultado
Suma: 8
resta: 4
Multiplicacion: 12
Division: 3
```

Parecido en 129: *tuplas_tie.cpp* pero diferente...

Listing 129: tuplas_tie.cpp

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

tuple<int, int, int, int> operaciones( int x, int y){
    return make_tuple(x+y, x-y, x*y, x/y);
```

```
}

int main (){

    int sum, resta, mult, div;

    tie(sum, resta, mult, div) = operaciones (6, 2);

    cout<< "Resultado\n";
    cout<< "Suma: "              << sum << endl;
    cout<< "resta: "                    << resta << endl;
    cout<< "Multiplicacion: "  << mult << endl;
    cout<< "Division: "          << div << endl;

    return 0;
}
```

tie debe usarse cuando necesitamos tuplas de referencias. Vemos en ejemplo en 130: *tuplasRef.cpp*

Listing 130: tuplasRef.cpp

```
tuple<int&, int&> min_max(int& x, int& y)
{
    if(x < y)
        return tie(x, y);
    else
        return tie(y, x);
}
```

11.5.2 *pair*

Veremos mas en detalle esta estructura, igualmente, este es un ejemplo de su uso 131: *par.cpp*

Listing 131: par.cpp

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

pair<int, int> sum_resta( int x, int y){
    return {x+y, x-y};
}

int main (){

    pair<int, int> resultado = sum_resta (6, 2);

    cout<< "Resultado\n";
```

```
    cout<< "Suma: "  << resultado.first  << endl;
    cout<< "resta: " << resultado.second << endl;

    return 0;
}
```

Es muy similar a la tupla, pero con la simpleza o limitación a dos elementos. Al *pair* lo provee la librería *utility.h*

Ejemplo de la definición de un grafo: 132

Listing 132: grafo.cpp

```
#include<iostream>
#include<vector>
#include<utility>
using namespace std;

/*
Sea un grafo G = (V, E)
*/
class grafo{

    vector<char> V;
    vector<pair<pair<char, char>, int>> E;
public:
    grafo(){}
        ~grafo(){};

    void insertar_vertice(const char&);
    void insertar_arista(const char&, const char&, const int&);

    friend ostream& operator <<(ostream&, grafo);
};
```

12 ALGUNAS ESTRUCTURAS ÚTILES

12.1 PAIR

Su declaración e inicialización dentro del estándar 11 sería como en el ejemplo 133: *initPar.cpp*:

Listing 133: initPar.cpp

```
std::pair <std::string,double> producto1; //constructor default
std::pair <std::string,double> producto2 ("tomates",2.30);  //
    inicializando con valores
std::pair <std::string,double> producto3 (producto2); //
    constructor de copia

product1 = std::make_pair(std::string("bulbos"),0.99); //usando
    make_pair (move)
```

12.1.1 *Operadores de comparación*

El operador de comparación es binario, o sea tiene operador izquierdo y derecho.

- el operador == compara que ambos elementos sean iguales. Retorna true si izq.first == derecho.first y izq.second == der.second, sino retornará false. Veamos el ejemplo 134: *operPar.cpp*:

Listing 134: operPar.cpp

```
std::pair<int, int> p1 = std::make_pair(1, 2);
std::pair<int, int> p2 = std::make_pair(2, 2);
    if (p1 == p2)
        std::cout << "son iguales";
    else
        std::cout << "no son iguales";
```

- El operador != revisa que algunos de los elementos no sean iguales. Es el caso contrario a lo visto en el item anterior. Retorna true si encuentra alguna diferencia o false si son iguales.

- El operador < chequea que izq.first sea menor a der.first y retorna true, en caso contrario retorna false. En otro caso, chequea al segundo componente retornando true si es menor, sino false.
- el operador <= retorna !(der < izq)
- el operador > retorna (der < izq)
- el operador >= retorna !(izq < der)

En 135: *comparaPar.cpp* vemos un ejemplo:

Listing 135: comparaPar.cpp

```
#include <iostream>
#include <utility>
#include <vector>
#include <algorithm>
#include <string>

int main()
{
    std::vector<std::pair<int, std::string>> v = { {2, "baz"},
        {2, "bar"}, {1, "foo"} };
    std::sort(v.begin(), v.end());
    for(const auto& p: v) {
        std::cout << "(" << p.first << "," << p.second << ") ";
        //output: (1,foo) (2,bar) (2,baz)
    }
}
```

12.1.2 *Ejemplo de accesos a componentes*

El código 136: *accesoAPar.cpp* nos muestra una posible forma:

Listing 136: accesoAPar.cpp

```
#include <iostream>
#include <utility>
int main()
{
    std::pair<int,int> p = std::make_pair(1,2); //se crea el par
    std::cout << p.first << " " << p.second << std::endl; //
        accedo a los elementos

    //Se puede hacer en dos tiempos
    std::pair<int,int> p1;
    p1.first = 3;
```

```
    p1.second = 4;
    std::cout << p1.first << " " << p1.second << std::endl;

    //Si usamos el constructor
    std::pair<int,int> p2 = std::pair<int,int>(5, 6);
    std::cout << p2.first << " " << p2.second << std::endl;

    return 0;

}
```

12.2 MAPS

- para usar std::map o std::multimap se necesita incluir el *header* <map>.
- std::map y std::multimap mantienen sus elementos ordenados según el orden ascendente de las claves. En el caso de std::multimap, no se clasifican los valores de la misma clave.
- La principal diferencia entre map y multimap es que el primero no admite duplicar valores con una misma clave y el multimap si.
- Maps se implementa como árboles de búsqueda binarios. Así que search(), insert(), erase() toma tiempo en promedio logarítmico. Para una operación de tiempo constante, use std::unordered_map.
- size() y empty() son funciones de tiempo promedio es 1, la cantidad de nodos se almacena en caché para evitar caminar a través del árbol cada vez que se llama a estas funciones.

12.2.1 *Accediendo a los datos*

Un Maps toma como entrada un par (key, valor). Podemos ver las siguientes formas de inicializarlo en el código 137: *initMap.cpp*

Listing 137: initMap.cpp

```
std::map < std::string, int > ranking { std::make_pair("
    stackoverflow", 2), std::make_pair("docs-beta", 1) };
```

La inserción dentro del map se haría de la siguiente forma 142: *insertMap.cpp*

Listing 138: insertMap.cpp

```
ranking["stackoverflow"]=2;
ranking["docs-beta"]=1;
```

Teniendo en cuenta el ejemplo, como *stackoverflow* ya se encuentra presente, se actualiza con el valor 2.

Para acceder podemos usar el indice de esta forma 139: *indiceMap.cpp*

Listing 139: indiceMap.cpp

```
std::cout << ranking[ "stackoverflow" ] << std::endl;
```

Esta forma tiene un problema, al usar los operadores [] se está realizando una inserción (aunque sea el update por el mismo valor), si la declaración es *const std::map*, no se podrá usar de este modo. C++11 facilita el acceso como vemos en 141: *indiceMapVariante.cpp*

Listing 140: indiceMapVariante.cpp

```
std::cout << ranking.ar("stackoverflow") << std::endl;
```

y se evita la inserción.

12.2.1.1 *Accediendo con Iteradores*

Para ambos contenedores, tanto maps como multimaps, los elementos pueden ser accedidos de la siguiente forma 141: *indiceMapVariante.cpp*

Listing 141: indiceMapVariante.cpp

```
// Version C++11 o superior
// Example using begin()

std::multimap < int, std::string > mmp { std::make_pair(2, "
    stackoverflow"), std::make_pair(1, "docs-beta"), std::
    make_pair(2, "stackexchange") };

auto it = mmp.begin();

std::cout << it->first << " : " << it->second << std::endl; //
    Output: "1 : docs-beta"
it++;
std::cout << it->first << " : " << it->second << std::endl; //
    Output: "2 : stackoverflow"
it++;
std::cout << it->first << " : " << it->second << std::endl; //
    Output: "2 : stackexchange"

// Example using rbegin()
```

```
std::map < int, std::string > mp { std::make_pair(2, "
    stackoverflow"), std::make_pair(1, "docs-beta"), std::
    make_pair(2, "stackexchange") };

auto it2 = mp.rbegin();
std::cout << it2->first << " : " << it2->second << std::endl; //
    Output: "2 : stackoverflow"
it2++;
std::cout << it2->first << " : " << it2->second << std::endl; //
    Output: "1 : docs-beta"
```

12.2.2 *Insertando elementos*

Algunas cosas han sido mencionadas. Para *std::map* se inserta si la clave no existe, sino realiza un update. Algunos ejemplos:

Listing 142: insertMap.cpp

```
std::map< std::string, size_t > fruits_count;
```

- Usando el método *insert()*, se requiere un pair para usarlo como mostramos en 143: *insertMapVariante.cpp*

  Listing 143: insertMapVariante.cpp

  ```
  fruits_count.insert({"grapes", 20});
  fruits_count.insert(make_pair("orange", 30));
  fruits_count.insert(pair<std::string, size_t>("banana", 40))
      ;
  fruits_count.insert(map<std::string, size_t>::value_type("
      cherry", 50));
  ```

  insert() retorna un pair que consiste en un iterador y un valor booleano:

  - Si la inserción fue exitosa, el iterador apunta al nuevo elemento y el bool es true.
  - Si la clave existe, el insert falla. El iterador apunta al elemento en cuestión y el bool es false.

  Veamos una variante en el código 144: *MapVariante.cpp*:

  Listing 144: MapVariante.cpp

  ```
  auto success = fruits_count.insert({"grapes", 20});
  if (!success.second) { // we already have 'grapes' in the
      map
  ```

```
success.first->second += 20; // access the iterator to
    update the value
}
```

- Esta es la forma ya vista, cuando se accede a un elemento através del índice, inserta o realiza el update en caso de ser necesario, el ejemplo **??**: *indiceMap2.cpp*.

Listing 145: indiceMap2.cpp

```
fruits_count["apple"] = 10;
```

- *insert()* se puede usar para agregar varios elementos a la vez en forma de pares. Retorna *void* como vemos en 146: *multipleInsertMap.cpp*.

Listing 146: multipleInsertMap.cpp

```
fruits_count.insert({{"apricot", 1}, {"jackfruit", 1}, {"
    lime", 1}, {"mango", 7}});
```

insert() puede ser usado tambien con iteradores como vemos en el código 147: *insertMapIterator.cpp*:

Listing 147: insertMapiterator.cpp

```
std::map< std::string, size_t > fruit_list{ {"lemon", 0}, {"
    olive", 0}, {"plum", 0}};
fruits_count.insert(fruit_list.begin(), fruit_list.end());
```

En el código 148: *ejemploMap.cpp* vemos el ejemplo:

Listing 148: ejemploMap.cpp

```
std::map<std::string, size_t> fruits_count;
std::string fruit;
while(std::cin >> fruit){
    // Inserta un elemento 'fruit' como clave y '1' de valor
    // (if the key is already stored in fruits_count, insert does
        nothing)
    auto ret = fruits_count.insert({fruit, 1});
    if(!ret.second){ // 'fruit' is already in the map
        ++ret.first->second; // increment the counter
    }
}
```

Se hizo ya el comentario que std::map implementa como estructura interna árboles, por lo tanto la operación es de orden O(log n)

En el estándar 11, un par puede ser construído usando *make_pair()* y *emplace()* como vemos en 149: *makepair.cpp*.

Listing 149: makepair.cpp

```
std::map< std::string , int > runs;
runs.emplace("Babe Ruth", 714);
runs.insert(make_pair("Barry Bonds", 762));
```

Si sabemos dónde se insertará el nuevo elemento, entonces podemos usar *emplace_hint()* para especificar una sugerencia de iterador. Si el nuevo elemento se puede insertar justo antes de la sugerencia, luego la inserción se puede realizar en tiempo constante. De lo contrario se comporta de la misma manera que *emplace()*. Veamos el ejemplo 150: insertposicion.cpp:

Listing 150: insertposicion.cpp

```
std::map< std::string , int > runs;
auto it = runs.emplace("Barry Bonds", 762); // guardo el iterador
     del elemento guardado
// el siguiente elemento sera ubicado despues de  "Barry Bonds",
    por lo tanto se insertara despues de 'it'
runs.emplace_hint(it, "Babe Ruth", 714);
```

12.2.3 *Busqueda*

Son varias las formas de buscar una clave en un std::map o std::multimap, veremos algunas posibles:

Con *find()* obtenemos al iterador de la primera ocurrencia, o (end()) en caso que no exista. Vemos el código ejemplo en 151: *buscar.cpp* .

Listing 151: buscar.cpp

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9},
    {3, 4}, {6, 7} };
auto it = mmp.find(6);

if(it!=mmp.end())
    std::cout << it->first << ", " << it->second << std::endl; //
        Imprime: 6, 5
else
    std::cout << "El valor no existe!" << std::endl;

it = mmp.find(66);
```

```
if(it!=mmp.end())
    std::cout << it->first << ", " << it->second << std::endl;
else
    std::cout << "El valor no existe!" << std::endl; // Esta es
        la linea que se mostrara.
```

Otra alternativa es usando *count()* previamente, principalmente para el uso de std::multimap, el cual controlaría la existancia y cantidad de veces que se encuentra la clave, en 152: *cantidad.cpp* vemos el uso.

Listing 152: cantidad.cpp

```
std::map< int , int > mp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4},
     {6, 7} };
if(mp.count(3) > 0) // existen 2 elementos con esa clave
   std::cout << "La clave existe!" << std::endl; // esta sera la
       linea ejecutada.
else
    std::cout << "La clave no existe!" << std::endl;
```

Para el caso del std::multimap, podemos querer obtener al conjunto que comparta la misma clave, para esto usaremos *equal_range()* que nos retorna un par de iteradores, lowerbound (inclusivo) y upper bound (exclusivo). Si la clave no existe retorna *end()*. Vemos el ejemplo: 153: *rangoelementos.cpp*:

Listing 153: rangoelementos.cpp

```
auto eqr = mmp.equal_range(6);
auto st = eqr.first, en = eqr.second;
for(auto it = st; it != en; ++it){
    std::cout << it->first << ", " << it->second << std::endl;
}
// prints: 6, 5
//         6, 7
```

12.2.4 *Inicialización*

En los ejemplos anteriores hemos visto como se inicializan los std::map y std::multimap, igualmente agregaremos algunos ejemplos y variantes como el uso de iteradores o arreglos de pares.

12.2.5 *Número de elementos*

Vemos el ejemplo en el siguiente código 155: *numelementos.cpp*.

Listing 154: initmap.cpp

```
std::multimap < int, std::string > mmp { std::make_pair(2, "
    stackoverflow"), std::make_pair(1, "docs-beta"),
std::make_pair(2, "stackexchange") };
// 1 docs-beta
// 2 stackoverflow
// 2 stackexchange

// como map no admite repeticiones, y con prioridad de derecha a
    izq, stackexchange sera sobreescrito
std::map < int, std::string > mp { std::make_pair(2, "
    stackoverflow"), std::make_pair(1, "docs-beta"),
std::make_pair(2, "stackexchange") };
// 1 docs-beta
// 2 stackoverflow

//pueden ser inicializados con iteradores
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9},
    {6, 8}, {3, 4}, {6, 7} };
        // {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 8}, {6, 7}, {8, 9}
auto it = mmp.begin();
std::advance(it,3); //mueve el cursor al primero {6, 5}
std::map< int, int > mp(it, mmp.end()); // {6, 5}, {8, 9}

//inicializacion por un arreglo de pares
std::pair< int, int > arr[10];
arr[0] = {1, 3};
arr[1] = {1, 5};
arr[2] = {2, 5};
arr[3] = {0, 1};
std::map< int, int > mp(arr,arr+4); //{0 , 1}, {1, 3}, {2, 5}

//inicializacion por un vector de pares
std::vector< std::pair<int, int> > v{ {1, 5}, {5, 1}, {3, 6}, {3,
     2} };
std::multimap< int, int > mp(v.begin(), v.end());
    // {1, 5}, {3, 6}, {3, 2}, {5, 1}
```

12.2.6 *Tipos de Maps*

12.2.6.1 *Regular Maps*

Como se ha visto, es un contenedor asociativo, clave-valor.

Se puede proporcionar un orden particular si no se desea respetar el predeterminado:

Si el comparador StrLess devuelve falso para dos claves, se consideran iguales incluso si su contenido real difieren.

Listing 155: numelementos.cpp

```
std::map<std::string , int> rank {{"facebook.com", 1} ,{"google.
    com", 2}, {"youtube.com", 3}};
if(!rank.empty())
    std::cout << "Number of elements in the rank map: " << rank.
        size() << std::endl;
else
    std::cout << "The rank map is empty" << std::endl;
```

Listing 156: nuevoorden.cpp

```
#include <string>
#include <map>
#include <cstring>

struct StrLess {
    bool operator()(const std::string& a, const std::string& b) {
        return strncmp(a.c_str(), b.c_str(), 8)<0;
            //compare only up to 8 first characters
    }
}
std::map<std::string, size_t, StrLess> fruits_count2;
```

12.2.6.2 *Multi-Map*

Se ha dicho ya, la diferencia fundamental es que admite repetición de claves, en el resto, el uso es similar.

12.2.6.3 *Hash-Map (Map desordenado)*

Un mapa hash almacena pares clave-valor de forma similar a un mapa normal. No ordena los elementos con respecto a la clave, aunque en su lugar, se utiliza un valor hash para la clave para acceder rápidamente a los pares clave-valor necesarios.

Listing 157: hashmap.cpp

```
#include <string>
#include <unordered_map>
std::unordered_map<std::string, size_t> fruits_count;
```

Los mapas desordenados suelen ser más rápidos, pero los elementos no se almacenan en ningún orden predecible. Por ejemplo, iterando

sobre todos los elementos en un mapa_desordenado da los elementos en un orden aparentemente aleatorio.

12.2.7 *Borrando elementos*

Algunos ejemplos:

Listing 158: borrado.cpp

```
// Borrando todos los elementos:

std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9},
    {3, 4}, {6, 7} };
mmp.clear(); //vacia al multimap

//Borrando un elementos con el iterador:

std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9},
    {3, 4}, {6, 7} };
     // {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 7}, {8, 9}
auto it = mmp.begin();
std::advance(it,3); // mueve el cursor a {6, 5}
mmp.erase(it); // {1, 2}, {3, 4}, {3, 4}, {6, 7}, {8, 9}

//Borrando un rango de elementos:

std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9},
    {3, 4}, {6, 7} };
         // {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 7}, {8, 9}
auto it = mmp.begin();
auto it2 = it;
t++; //mueve el cursor a {3, 4}
std::advance(it2,3); //mueve el segundo cursor a {6, 5}
mmp.erase(it,it2); // {1, 2}, {6, 5}, {6, 7}, {8, 9}

// Borra todos los elementos de una determinada key:

std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9},
    {3, 4}, {6, 7} };
    // {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 7}, {8, 9}
mmp.erase(6); // {1, 2}, {3, 4}, {3, 4}, {8, 9}
```

12.2.8 *Iterando sobre map o multimap*

Vemos un posible ejemplo en el código 159

Listing 159: iterandomap.cpp

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9},
    {3, 4}, {6, 7} };

for(const auto &x: mmp)
    std::cout<< x.first <<":"<< x.second << std::endl;

//iterando con un iterador de avance
//std::map< int, int >::iterator
for (auto it = mmp.begin(); it != mmp.end(); ++it)
std::cout<< it->first <<":"<< it->second << std::endl;

//iterando con un iterador de retroceso
//std::map< int, int >::reverse_iterator
for (auto it = mmp.rbegin(); it != mmp.rend(); ++it)
std::cout<< it->first <<" "<< it->second << std::endl;
```

12.2.9 *Usando clases propias como claves*

Para poder usar una clase como clave en un map, todo lo que se requiere de la clave es que sea copiable y asignable. El orden dentro del map está definido por el tercer argumento de la plantilla (y el argumento de el constructor, si se usa). Esto por defecto es std::less<KeyType>, que por defecto es el operador <, pero no hay requisito para utilizar los valores predeterminados. Simplemente escriba un operador de comparación (preferiblemente como un objeto funcional):

Listing 160: clavespropias.cpp

```
struct CmpMyType
{
    bool operator()( MyType const& lhs, MyType const& rhs ) const
    {
        // ...
    }
}
```

En C++, el predicado "compare" debe ser un ordenamiento débil estricto. En particular, compare(X,X) debe devolver falso para cualquier X. es decir, si CmpMyType()(a, b) devuelve verdadero, entonces CmpMyType()(b, a) debe devolver falso, y si ambos devuelven falso, los elementos se consideran iguales (miembros de la misma clase de equivalencia).

12.2.9.1 *Ordenamiento débil estricto*

Es el término que define la relación entre dos objetos.

> Dos objetos x y y son equivalentes si ambos $f(x, y)$ y $f(y, x)$ son falses. Note que un objeto es siempre (por irreflexividad invariante) equivalente con si mismo.

En términos de C++, esto significa que si tiene dos objetos de un tipo dado, debe devolver los siguientes valores cuando en comparación con el operador <.

X a; X b;		
Condición	Test	Resultado
a es equivalente a b	a < b	false
a es equivalente a b	b < a	false
a es menor a b	a < b	true
a es menor a b	b < a	false
b es menor a a	a < b	false
b es menor a a	b < a	true

Table 17: Strict Weak Ordering

Part III

APPENDIX

What is ‘‘Object-Oriented Programming’’? (1991 revised version)

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

‘‘Object-Oriented Programming’’ and ‘‘Data Abstraction’’ have become very common terms. Unfortunately, few people agree on what they mean. I will offer informal definitions that appear to make sense in the context of languages like Ada, C++, Modula-2, Simula, and Smalltalk. The general idea is to equate ‘‘support for data abstraction’’ with the ability to define and use new types and equate ‘‘support for object-oriented programming’’ with the ability to express type hierarchies. Features necessary to support these programming styles in a general purpose programming language will be discussed. The presentation centers around C++ but is not limited to facilities provided by that language.

1 Introduction

Not all programming languages can be ‘‘object oriented’’. Yet claims have been made to the effect that APL, Ada, Clu, C++, LOOPS, and Smalltalk are object-oriented programming languages. I have heard discussions of object-oriented design in C, Pascal, Modula-2, and CHILL. Could there somewhere be proponents of object-oriented Fortran and Cobol programming? I think there must be. ‘‘Object-oriented’’ has in many circles become a high-tech synonym for ‘‘good’’, and when you examine discussions in the trade press, you can find arguments that appear to boil down to syllogisms like:

```
Ada is good
Object oriented is good
-----------------------------------
Ada is object oriented
```

This paper presents one view of what ‘‘object oriented’’ ought to mean in the context of a general purpose programming language.

- §2 Distinguishes ‘‘object-oriented programming’’ and ‘‘data abstraction’’ from each other and from other styles of programming and presents the mechanisms that are essential for supporting the various styles of programming.
- §3 Presents features needed to make data abstraction effective.
- §4 Discusses facilities needed to support object-oriented programming.
- §5 Presents some limits imposed on data abstraction and object-oriented programming by traditional hardware architectures and operating systems.

Examples will be presented in C++. The reason for this is partly to introduce C++ and partly because C++ is one of the few languages that supports both data abstraction and object-oriented programming in addition to traditional programming techniques. Issues of concurrency and of hardware support for specific higher-level language constructs are ignored in this paper.

2 Programming Paradigms

Object-oriented programming is a technique for programming − a paradigm for writing ‘‘good’’ programs for a set of problems. If the term ‘‘object-oriented programming language’’ means anything it must mean a programming language that provides mechanisms that support the object-oriented style of programming well.

There is an important distinction here. A language is said to *support* a style of programming if it provides facilities that makes it convenient (reasonably easy, safe, and efficient) to use that style. A language does not support a technique if it takes exceptional effort or exceptional skill to write such programs; it merely *enables* the technique to be used. For example, you can write structured programs in Fortran, write type-secure programs in C, and use data abstraction in Modula-2, but it is unnecessarily hard to do because these languages do not support those techniques.

Support for a paradigm comes not only in the obvious form of language facilities that allow direct use of the paradigm, but also in the more subtle form of compile-time and/or run-time checks against unintentional deviation from the paradigm. Type checking is the most obvious example of this; ambiguity detection and run-time checks can be used to extend linguistic support for paradigms. Extra-linguistic facilities such as standard libraries and programming environments can also provide significant support for paradigms.

A language is not necessarily better than another because it possesses a feature the other does not. There are many example to the contrary. The important issue is not so much what features a language possesses but that the features it does possess are sufficient to support the desired programming styles in the desired application areas:

[1] All features must be cleanly and elegantly integrated into the language.

[2] It must be possible to use features in combination to achieve solutions that would otherwise have required extra separate features.

[3] There should be as few spurious and ‘‘special purpose’’ features as possible.

[4] A feature should be such that its implementation does not impose significant overheads on programs that do not require it.

[5] A user need only know about the subset of the language explicitly used to write a program.

The last two principles can be summarized as ‘‘what you don’t know won’t hurt you.’’ If there are any doubts about the usefulness of a feature it is better left out. It is *much* easier to add a feature to a language than to remove or modify one that has found its way into the compilers or the literature.

I will now present some programming styles and the key language mechanisms necessary for supporting them. The presentation of language features is not intended to be exhaustive.

2.1 Procedural Programming

The original (and probably still the most commonly used) programming paradigm is:

> *Decide which procedures you want;*
> *use the best algorithms you can find.*

The focus is on the design of the processing, the algorithm needed to perform the desired computation. Languages support this paradigm by facilities for passing arguments to functions and returning values from functions. The literature related to this way of thinking is filled with discussion of ways of passing arguments, ways of distinguishing different kinds of arguments, different kinds of functions (procedures, routines, macros, ...), etc. Fortran is the original procedural language; Algol60, Algol68, C, and Pascal are later inventions in the same tradition.

A typical example of ‘‘good style’’ is a square root function. It neatly produces a result given an argument. To do this, it performs a well understood mathematical computation:

```
double sqrt(double arg)
{
    // the code for calculating a square root
}
```

```
void some_function()
{
    double root2 = sqrt(2);
    // ...
}
```

From a program organization point of view, functions are used to create order in a maze of algorithms.

2.2 Data Hiding

Over the years, the emphasis in the design of programs has shifted away from the design of procedures towards the organization of data. Among other things, this reflects an increase in the program size. A set of related procedures with the data they manipulate is often called a *module*. The programming paradigm becomes:

> *Decide which modules you want;*
> *partition the program so that data is hidden in modules.*

This paradigm is also known as the ‘‘data hiding principle’’. Where there is no grouping of procedures with related data the procedural programming style suffices. In particular, the techniques for designing ‘‘good procedures’’ are now applied for each procedure in a module. The most common example is a definition of a stack module. The main problems that have to be solved for a good solution are:

[1] Provide a user interface for the stack (for example, functions `push()` and `pop()`).

[2] Ensure that the representation of the stack (for example, a vector of elements) can only be accessed through this user interface.

[3] Ensure that the stack is initialized before its first use.

Here is a plausible external interface for a stack module:

```
// declaration of the interface of module stack of characters
char pop();
void push(char);
const stack_size = 100;
```

Assuming that this interface is found in a file called `stack.h`, the ‘‘internals’’ can be defined like this:

```
#include "stack.h"
static char v[stack_size];    // ‘‘static’’ means local to this file/module
static char* p = v;           // the stack is initially empty

char pop()
{
    // check for underflow and pop
}

void push(char c)
{
    // check for overflow and push
}
```

It would be quite feasible to change the representation of this stack to a linked list. A user does not have access to the representation anyway (since `v` and `p` were declared `static`, that is local to the file/module in which they were declared). Such a stack can be used like this:

```
#include "stack.h"

void some_function()
{
    char c = pop(push('c'));
    if (c != 'c') error("impossible");
}
```

Pascal (as originally defined) doesn't provide any satisfactory facilities for such grouping: the only mechanism for hiding a name from ''the rest of the program'' is to make it local to a procedure. This leads to strange procedure nestings and over-reliance on global data.

C fares somewhat better. As shown in the example above, you can define a ''module'' by grouping related function and data definitions together in a single source file. The programmer can then control which names are seen by the rest of the program (a name can be seen by the rest of the program *unless* it has been declared `static`). Consequently, in C you can achieve a degree of modularity. However, there is no generally accepted paradigm for using this facility and the technique of relying on `static` declarations is rather low level.

One of Pascal's successors, Modula-2, goes a bit further. It formalizes the concept of a module, making it a fundamental language construct with well defined module declarations, explicit control of the scopes of names (import/export), a module initialization mechanism, and a set of generally known and accepted styles of usage.

The differences between C and Modula-2 in this area can be summarized by saying that C only *enables* the decomposition of a program into modules, while Modula-2 *supports* that technique.

2.3 Data Abstraction

Programming with modules leads to the centralization of all data of a type under the control of a type manager module. If one wanted two stacks, one would define a stack manager module with an interface like this:

```
class stack_id;  // stack_id is a type
                 // no details about stacks or stack_ids are known here

stack_id create_stack(int size); // make a stack and return its identifier
destroy_stack(stack_id);         // call when stack is no longer needed

void push(stack_id, char);
char pop(stack_id);
```

This is certainly a great improvement over the traditional unstructured mess, but ''types'' implemented this way are clearly very different from the built-in types in a language. Each type manager module must define a separate mechanism for creating ''variables'' of its type, there is no established norm for assigning object identifiers, a ''variable'' of such a type has no name known to the compiler or programming environment, nor do such ''variables'' do not obey the usual scope rules or argument passing rules.

A type created through a module mechanism is in most important aspects different from a built-in type and enjoys support inferior to the support provided for built-in types. For example:

```
void f()
{
    stack_id s1;
    stack_id s2;

    s1 = create_stack(200);
    // Oops: forgot to create s2

    char c1 = pop(s1,push(s1,'a'));
    if (c1 != 'c') error("impossible");
```

```
    char c2 = pop(s2,push(s2,'a'));
    if (c2 != 'c') error("impossible");

    destroy(s2);
    // Oops: forgot to destroy s1
}
```

In other words, the module concept that supports the data hiding paradigm enables this style of programming, but it does not support it.

Languages such as Ada, Clu, and C++ attack this problem by allowing a user to define types that behave in (nearly) the same way as built-in types. Such a type is often called an *abstract data type*†. The programming paradigm becomes:

> *Decide which types you want;*
> *provide a full set of operations for each type.*

Where there is no need for more that one object of a type the data hiding programming style using modules suffices. Arithmetic types such as rational and complex numbers are common examples of user-defined types:

```
class complex {
    double re, im;
public:
    complex(double r, double i) { re=r; im=i; }
    complex(double r) { re=r; im=0; }   // float->complex conversion

    friend complex operator+(complex, complex);
    friend complex operator-(complex, complex);   // binary minus
    friend complex operator-(complex);            // unary minus
    friend complex operator*(complex, complex);
    friend complex operator/(complex, complex);
    // ...
};
```

The declaration of class (that is, user-defined type) `complex` specifies the representation of a complex number and the set of operations on a complex number. The representation is *private*; that is, `re` and `im` are accessible only to the functions specified in the declaration of class `complex`. Such functions can be defined like this:

```
complex operator+(complex a1, complex a2)
{
    return complex(a1.re+a2.re,a1.im+a2.im);
}
```

and used like this:

```
complex a = 2.3;
complex b = 1/a;
complex c = a+b*complex(1,2.3);
// ...
c = -(a/b)+2;
```

Most, but not all, modules are better expressed as user defined types. For concepts where the ''module representation'' is desirable even when a proper facility for defining types is available, the programmer can declare a type and only a single object of that type. Alternatively, a language might provide a module

† I prefer the term ''user-defined type'': ''*Those types are not "abstract"; they are as real as* `int` *and* `float`.'' − Doug McIlroy. An alternative definition of *abstract data types* would require a mathematical ''abstract'' specification of all types (both built-in and user-defined). What is referred to as types in this paper would, given such a specification, be concrete specifications of such truly abstract entities.

concept in addition to and distinct from the class concept.

2.4 Problems with Data Abstraction

An abstract data type defines a sort of black box. Once it has been defined, it does not really interact with the rest of the program. There is no way of adapting it to new uses except by modifying its definition. This can lead to severe inflexibility. Consider defining a type `shape` for use in a graphics system. Assume for the moment that the system has to support circles, triangles, and squares. Assume also that you have some classes:

```
class point{ /* ... */ };
class color{ /* ... */ };
```

You might define a shape like this:

```
enum kind { circle, triangle, square };

class shape {
    point center;
    color col;
    kind k;
    // representation of shape
public:
    point where()       { return center; }
    void move(point to) { center = to; draw(); }
    void draw();
    void rotate(int);
    // more operations
};
```

The ‘‘type field’’ `k` is necessary to allow operations such as `draw()` and `rotate()` to determine what kind of shape they are dealing with (in a Pascal-like language, one might use a variant record with tag `k`). The function `draw()` might be defined like this:

```
void shape::draw()
{
    switch (k) {
    case circle:
        // draw a circle
        break;
    case triangle:
        // draw a triangle
        break;
    case square:
        // draw a square
    }
}
```

This is a mess. Functions such as `draw()` must ‘‘know about’’ all the kinds of shapes there are. Therefore the code for any such function grows each time a new shape is added to the system. If you define a new shape, every operation on a shape must be examined and (possibly) modified. You are not able to add a new shape to a system unless you have access to the source code for every operation. Since adding a new shape involves ‘‘touching’’ the code of every important operation on shapes, it requires great skill and potentially introduces bugs into the code handling other (older) shapes. The choice of representation of particular shapes can get severely cramped by the requirement that (at least some of) their representation must fit into the typically fixed sized framework presented by the definition of the general type `shape`.

2.5 Object-Oriented Programming

The problem is that there is no distinction between the general properties of any shape (a shape has a color, it can be drawn, etc.) and the properties of a specific shape (a circle is a shape that has a radius, is drawn by a circle-drawing function, etc.). Expressing this distinction and taking advantage of it defines object-oriented programming. A language with constructs that allows this distinction to be expressed and

used supports object-oriented programming. Other languages don't.

The Simula inheritance mechanism provides a solution. First, specify a class that defines the general properties of all shapes:

```
class shape {
    point center;
    color col;
    // ...
public:
    point where() { return center; }
    void move(point to) { center = to; draw(); }
    virtual void draw();
    virtual void rotate(int);
    // ...
};
```

The functions for which the calling interface can be defined, but where the implementation cannot be defined except for a specific shape, have been marked ''virtual'' (the Simula and C++ term for ''may be re-defined later in a class derived from this one''). Given this definition, we can write general functions manipulating shapes:

```
void rotate_all(shape* v, int size, int angle)
// rotate all members of vector "v" of size "size" "angle" degrees
{
    for (int i = 0; i < size; i++) v[i].rotate(angle);
}
```

To define a particular shape, we must say that it is a shape and specify its particular properties (including the virtual functions).

```
class circle : public shape {
    int radius;
public:
    void draw() { /* ... */ };
    void rotate(int) {}    // yes, the null function
};
```

In C++, class `circle` is said to be *derived* from class `shape`, and class `shape` is said to be a *base* of class `circle`. An alternative terminology calls `circle` and `shape` subclass and superclass, respectively.

The programming paradigm is:

> *Decide which classes you want;*
> *provide a full set of operations for each class;*
> *make commonality explicit by using inheritance.*

Where there is no such commonality data abstraction suffices. The amount of commonality between types that can be exploited by using inheritance and virtual functions is the litmus test of the applicability of object-oriented programming to an application area. In some areas, such as interactive graphics, there is clearly enormous scope for object-oriented programming. For other areas, such as classical arithmetic types and computations based on them, there appears to be hardly any scope for more than data abstraction and the facilities needed for the support of object-oriented programming seem unnecessary†.

Finding commonality among types in a system is not a trivial process. The amount of commonality to be exploited is affected by the way the system is designed. When designing a system, commonality must be actively sought, both by designing classes specifically as building blocks for other types, and by examining classes to see if they exhibit similarities that can be exploited in a common base class.

† However, more advanced mathematics may benefit from the use of inheritance: Fields are specializations of rings; vector spaces a special case of modules.

For attempts to explain what object-oriented programming is without recourse to specific programming language constructs see Nygaard[13] and Kerr[9]. For a case study in object-oriented programming see Cargill[4].

3 Support for Data Abstraction

The basic support for programming with data abstraction consists of facilities for defining a set of operations for a type and for restricting the access to objects of the type to that set of operations. Once that is done, however, the programmer soon finds that language refinements are needed for convenient definition and use of the new types. Operator overloading is a good example of this.

3.1 Initialization and Cleanup

When the representation of a type is hidden some mechanism must be provided for a user to initialize variables of that type. A simple solution is to require a user to call some function to initialize a variable before using it. For example:

```
class vector {
    int  sz;
    int* v;
public:
    void init(int size);   // call init to initialize sz and v
                           // before the first use of a vector
    // ...
};

vector v;
// don't use v here
v.init(10);
// use v here
```

This is error prone and inelegant. A better solution is to allow the designer of a type to provide a distinguished function to do the initialization. Given such a function, allocation and initialization of a variable becomes a single operation (often called instantiation) instead of two separate operations. Such an initialization function is often called a constructor. In cases where construction of objects of a type is non-trivial, one often needs a complementary operation to clean up objects after their last use. In C++, such a cleanup function is called a destructor. Consider a vector type:

```
class vector {
    int  sz;                    // number of elements
    int* v;                     // pointer to integers
public:
    vector(int);                // constructor
    ~vector();                  // destructor
    int& operator[](int index);   // subscript operator
};
```

The `vector` constructor can be defined to allocate space like this:

```
vector::vector(int s)
{
    if (s<=0) error("bad vector size");
    sz = s;
    v = new int[s];   // allocate an array of "s" integers
}
```

The `vector` destructor frees the storage used:

```
vector::~vector()
{
    delete v;         // deallocate the memory pointed to by v
}
```

C++ does not support garbage collection. This is compensated for, however, by enabling a type to maintain

its own storage management without requiring intervention by a user. This is a common use for the constructor/destructor mechanism, but many uses of this mechanism are unrelated to storage management.

3.2 Assignment and Initialization

Controlling construction and destruction of objects is sufficient for many types, but not for all. It can also be necessary to control all copy operations. Consider class `vector`:

```
vector v1(100);
vector v2 = v1;  // make a new vector v2 initialized to v1
v1 = v2;         // assign v2 to v1
```

It must be possible to define the meaning of the initialization of `v2` and the assignment to `v1`. Alternatively it should be possible to prohibit such copy operations; preferably both alternatives should be available. For example:

```
class vector {
    int* v;
    int  sz;
public:
    // ...
    void operator=(vector&);  // assignment
    vector(vector&);          // initialization
};
```

specifies that user-defined operations should be used to interpret `vector` assignment and initialization. Assignment might be defined like this:

```
vector::operator=(vector& a) // check size and copy elements
{
    if (sz != a.sz) error("bad vector size for =");
    for (int i = 0; i<sz; i++) v[i] = a.v[i];
}
```

Since the assignment operation relies on the ‘‘old value’’ of the vector being assigned to, the initialization operation *must* be different. For example:

```
vector::vector(vector& a)    // initialize a vector from another vector
{
    sz = a.sz;               // same size
    v = new int[sz];         // allocate element array
    for (int i = 0; i<sz; i++) v[i] = a.v[i];    // copy elements
}
```

In C++, a constructor of the form `X(X&)` defines all initialization of objects of type `X` with another object of type `X`. In addition to explicit initialization constructors of the form `X(X&)` are used to handle arguments passed ‘‘by value’’ and function return values.

In C++ assignment of an object of class `X` can be prohibited by declaring assignment private:

```
class X {
    void operator=(X&);   // only members of X can
    X(X&);                // copy an X
    // ...
public:
    // ...
};
```

Ada does not support constructors, destructors, overloading of assignment, or user-defined control of argument passing and function return. This severely limits the class of types that can be defined and forces the programmer back to ‘‘data hiding techniques’’; that is, the user must design and use type manager modules rather than proper types.

3.3 Parameterized Types

Why would you want to define a vector of integers anyway? A user typically needs a vector of elements of some type unknown to the writer of the `vector` type. Consequently the vector type ought to be expressed in such a way that it takes the element type as an argument:

```
class vector<class T> {     // vector of elements of type T
    T* v;
    int sz;
public:
    vector(int s)
    {
        if (s <= 0) error("bad vector size");
        v = new T[sz = s];    // allocate an array of "s" "T"s
    }
    T& operator[](int i);
    int size() { return sz; }
    // ...
};
```

Vectors of specific types can now be defined and used:

```
vector<int> v1(100);      // v1 is a vector of 100 integers
vector<complex> v2(200);  // v2 is a vector of 200 complex numbers

v2[i] = complex(v1[x],v1[y]);
```

Ada, Clu, and ML support parameterized types. Unfortunately, C++ does not; the notation used here is simply devised for illustration. Where needed, parameterized classes are ‘‘faked’’ using macros. There need not be any run-time overheads compared with a class where all types involved are specified directly.

Typically a parameterized type will have to depend on at least some aspect of a type parameter. For example, some of the vector operations must assume that assignment is defined for objects of the parameter type. How can one ensure that? One solution to this problem is to require the designer of the parameterized class to state the dependency. For example, ‘‘T must be a type for which = is defined’’. A better solution is not to or to take a specification of an argument type as a partial specification. A compiler can detect a ‘‘missing operation’’ if it is applied and give an error message such as. For example:

```
cannot define vector(non_copy)::operator[](non_copy&):
    type non_copy does not have operator=
```

This technique allows the definition of types where the dependency on attributes of a parameter type is handled at the level of the individual operation of the type. For example, one might define a vector with a sort operation. The sort operation might use `<`, `==`, and `=` on objects of the parameter type. It would still be possible to define vectors of a type for which `'<'` was not defined as long as the vector sorting operation was not actually invoked.

A problem with parameterized types is that each instantiation creates an independent type. For example, the type `vector<char>` is unrelated to the type `vector<complex>`. Ideally one would like to be able to express and utilize the commonality of types generated from the same parameterized type. For example, both `vector<char>` and `vector<complex>` have a `size()` function that is independent of the parameter type. It is possible, but not trivial, to deduce this from the definition of class `vector` and then allow `size()` to be applied to any `vector`. An interpreted language or a language supporting both parameterized types and inheritance has an advantage here.

3.4 Exception Handling

As programs grow, and especially when libraries are used extensively, standards for handling errors (or more generally: ‘‘exceptional circumstances’’) become important. Ada, Algol68, and Clu each support a standard way of handling exceptions. Unfortunately, C++ does not. Where needed exceptions are ‘‘faked’’ using pointers to functions, ‘‘exception objects’’, ‘‘error states’’, and the C library `signal` and `longjmp` facilities. This is not satisfactory in general and fails even to provide a standard framework for error handling.

Consider again the `vector` example. What *ought* to be done when an out of range index value is

passed to the subscript operator? The designer of the `vector` class should be able to provide a default behavior for this. For example:

```
class vector {
    ...
    except vector_range {
        // define an exception called vector_range
        // and specify default code for handling it
            error("global: vector range error");
            exit(99);
    }
}
```

Instead of calling an error function, `vector::operator[]()` can invoke the exception handling code, ‘‘raise the exception’’:

```
int& vector::operator[](int i)
{
    if (0<i || sz<=i) raise vector_range;
    return v[i];
}
```

This will cause the call stack to be unraveled until an exception handler for `vector_range` is found; this handler will than be executed.

An exception handler may be defined for a specific block:

```
void f() {
    vector v(10);
    try {               // errors here are handled by the local
            // exception handler defined below
        // ...
        int i = g();    // g might cause a range error using some vector
        v[i] = 7;       // potential range error
    }
    except {
    vector::vector_range:
        error("f(): vector range error");
        return;
    }                   // errors here are handled by the global
            // exception handler defined in vector

    int i = g();        // g might cause a range error using some vector
    v[i] = 7;           // potential range error
}
```

There are many ways of defining exceptions and the behavior of exception handlers. The facility sketched here resembles the ones found in Clu and Modula-2+. This style of exception handling can be implemented so that code is not executed unless an exception is raised† or portably across most C implementations by using `setjmp()` and `longjmp()`.††

Could exceptions, as defined above, be completely ‘‘faked’’ in a language such as C++? Unfortunately, no. The snag is that when an exception occurs, the run-time stack must be unraveled up to a point where a handler is defined. To do this properly in C++ involves invoking destructors defined in the scopes involved. This is not done by a C `longjmp()` and cannot in general be done by the user.

† except possibly for some initialization code at the start of a program.
†† see the C library manual for your system.

3.5 Coercions

User-defined coercions, such as the one from floating point numbers to complex numbers implied by the constructor `complex(double)`, have proven unexpectedly useful in C++. Such coercions can be applied explicitly or the programmer can rely on the compiler to add them implicitly where necessary and unambiguous:

```
complex a = complex(1);
complex b = 1;          // implicit: 1 -> complex(1)
a = b+complex(2);
a = b+2;                // implicit: 2 -> complex(2)
```

Coercions were introduced into C++ because mixed mode arithmetic is the norm in languages for numerical work and because most user-defined types used for ‘‘calculation’’ (for example, matrices, character strings, and machine addresses) have natural mappings to and/or from other types.

One use of coercions has proven especially useful from a program organization point of view:

```
complex a = 2;
complex b = a+2;  // interpreted as operator+(a,complex(2))
b = 2+a;          // interpreted as operator+(complex(2),a)
```

Only one function is needed to interpret ‘‘+’’ operations and the two operands are handled identically by the type system. Furthermore, class `complex` is written without any need to modify the concept of integers to enable the smooth and natural integration of the two concepts. This is in contrast to a ‘‘pure object-oriented system’’ where the operations would be interpreted like this:

```
a+2;    // a.operator+(2)
2+a;    // 2.operator+(a)
```

making it necessary to modify class `integer` to make `2+a` legal. Modifying existing code should be avoided as far as possible when adding new facilities to a system. Typically, object-oriented programming offers superior facilities for adding to a system without modifying existing code. In this case, however, data abstraction facilities provide a better solution.

3.6 Iterators

It has been claimed that a language supporting data abstraction must provide a way of defining control structures[11]. In particular, a mechanism that allows a user to define a loop over the elements of some type containing elements is often needed. This must be achieved without forcing a user to depend on details of the implementation of the user-defined type. Given a sufficiently powerful mechanism for defining new types and the ability to overload operators, this can be handled without a separate mechanism for defining control structures.

For a vector, defining an iterator is not necessary since an ordering is available to a user through the indices. I’ll define one anyway to demonstrate the technique. There are several possible styles of iterators. My favorite relies on overloading the function application operator `()`†††:

```
class vector_iterator {
    vector& v;
    int i;
public:
    vector_iterator(vector& r) { i = 0; v = r; }
    int operator()() { return i<v.size() ? v.elem(i++) : 0; }
};
```

A `vector_iterator` can now be declared and used for a `vector` like this:

††† This style also relies on the existence of a distinguished value to represent ‘‘end of iteration’’. Often, in particular for C++ pointer types, `0` can be used.

```
vector v(sz);
vector_iterator next(v);
int i;
while (i=next()) print(i);
```

More than one iterator can be active for a single object at one time, and a type may have several different iterator types defined for it so that different kinds of iteration may be performed. An iterator is a rather simple control structure. More general mechanisms can also be defined. For example, the C++ standard library provides a co-routine class[15].

For many ‘‘container’’ types, such as `vector,` one can avoid introducing a separate iterator type by defining an iteration mechanism as part of the type itself. A `vector` might be defined to have a ‘‘current element’’:

```
class vector {
    int* v;
    int  sz;
    int current;
public:
    // ...
    int next() { return (current++<sz) ? v[current] : 0; }
    int prev() { return (0<--current) ? v[current] : 0; }
};
```

Then the iteration can be performed like this:

```
vector v(sz);
int i;
while (i=v.next()) print(i);
```

This solution is not as general as the iterator solution, but avoids overhead in the important special case where only one kind of iteration is needed and where only one iteration at a time is needed for a vector. If necessary, a more general solution can be applied in addition to this simple one. Note that the ‘‘simple’’ solution requires more foresight from the designer of the container class than the iterator solution does. The iterator-type technique can also be used to define iterators that can be bound to several different container types thus providing a mechanism for iterating over different container types with a single iterator type.

3.7 Implementation Issues

The support needed for data abstraction is primarily provided in the form of language features implemented by a compiler. However, parameterized types are best implemented with support from a linker with some knowledge of the language semantics, and exception handling requires support from the run-time environment. Both can be implemented to meet the strictest criteria for both compile time speed and efficiency without compromising generality or programmer convenience.

As the power to define types increases, programs to a larger degree depend on types from libraries (and not just those described in the language manual). This naturally puts greater demands on facilities to express what is inserted into or retrieved from a library, facilities for finding out what a library contains, facilities for determining what parts of a library are actually used by a program, etc.

For a compiled language facilities for calculating the minimal compilation necessary after a change become important. It is essential that the linker/loader is capable of bringing a program into memory for execution without also bringing in large amounts of related, but unused, code. In particular, a library/linker/loader system that brings the code for every operation on a type into core just because the programmer used one or two operations on the type is worse than useless.

4 Support for Object-Oriented programming

The basic support a programmer needs to write object-oriented programs consists of a class mechanism with inheritance and a mechanism that allows calls of member functions to depend on the actual type of an object (in cases where the actual type is unknown at compile time). The design of the member function calling mechanism is critical. In addition, facilities supporting data abstraction techniques (as described

above) are important because the arguments for data abstraction and for its refinements to support elegant use of types are equally valid where support for object-oriented programming is available. The success of both techniques hinges on the design of types and on the ease, flexibility, and efficiency of such types. Object-oriented programming simply allows user-defined types to be far more flexible and general than the ones designed using only data abstraction techniques.

4.1 Calling Mechanisms

The key language facility supporting object-oriented programming is the mechanism by which a member function is invoked for a given object. For example, given a pointer `p`, how is a call `p->f(arg)` handled? There is a range of choices.

In languages such as C++ and Simula, where static type checking is extensively used, the type system can be employed to select between different calling mechanisms. In C++, two alternatives are available:

[1] A normal function call: the member function to be called is determined at compile time (through a lookup in the compiler's symbol tables) and called using the standard function call mechanism with an argument added to identify the object for which the function is called. Where the ''standard function call'' is not considered efficient enough, the programmer can declare a function `inline` and the compiler will attempt to inline expand its body. In this way, one can achieve the efficiency of a macro expansion without compromising the standard function semantics. This optimization is equally valuable as a support for data abstraction.

[2] A virtual function call: The function to be called depends on the type of the object for which it is called. This type cannot in general be determined until run time. Typically, the pointer `p` will be of some base class `B` and the object will be an object of some derived class `D` (as was the case with the base class `shape` and the derived class `circle` above). The call mechanism must look into the object and find some information placed there by the compiler to determine which function `f` is to be called. Once that function is found, say `D::f`, it can be called using the mechanism described above. The name `f` is at compile time converted into an index into a table of pointers to functions. This virtual call mechanism can be made essentially as efficient as the ''normal function call'' mechanism. In the standard C++ implementation, only five additional memory references are used.

In languages with weak static type checking a more elaborate mechanism must be employed. What is done in a language like Smalltalk is to store a list of the names of all member functions (methods) of a class so that they can be found at run time:

[3] A method invocation: First the appropriate table of method names is found by examining the object pointed to by `p`. In this table (or set of tables) the string `"f"` is looked up to see if the object has an `f()`. If an `f()` is found it is called; otherwise some error handling takes place. This lookup differs from the lookup done at compiler time in a statically checked language in that the method invocation uses a method table for the actual object.

A method invocation is inefficient compared with a virtual function call, but more flexible. Since static type checking of arguments typically cannot be done for a method invocation, the use of methods must be supported by dynamic type checking.

4.2 Type Checking

The shape example showed the power of virtual functions. What, in addition to this, does a method invocation mechanism do for you? You can attempt to invoke *any* method for *any* object.

The ability to invoke any method for any object enables the designer of general purpose libraries to push the responsibility for handling types onto the user. Naturally this simplifies the design of libraries. For example:

```
class stack {   // assume class any has a member next
    any* v;
    void push(any* p)
    {
        p->next = v;
        v = p;
    }
    any* pop()
    {
        if (v == 0) return error_obj;
        any* r = v;
        v = v->next;
        return r;
    }
};
```

It becomes the responsibility of the user to avoid type mismatches like this:

```
stack<any*> cs;

cs.push(new Saab900);
cs.push(new Saab37B);

plane* p = (plane*)cs.pop();
p->takeoff();

p = (plane*)cs.pop();
p->takeoff();        // Oops! Run time error: a Saab 900 is a car
                     // a car does not have a takeoff method.
```

An attempt to use a `car` as a `plane` will be detected by the message handler and an appropriate error handler will be called. However, that is only a consolation when the user is also the programmer. The absence of static type checking makes it difficult to guarantee that errors of this class are not present in systems delivered to end-users. Naturally, a language designed with methods and without static types can express this example with fewer keystrokes.

Combinations of parameterized classes and the use of virtual functions can approach the flexibility, ease of design, and ease of use of libraries designed with method lookup without relaxing the static type checking or incurring measurable run time overheads (in time or space). For example:

```
stack<plane*> cs;

cs.push(new Saab900);    // Compile time error:
                         // type mismatch: car* passed, plane* expected
cs.push(new Saab37B);

plane* p = cs.pop();
p->takeoff();            // fine: a Saab 37B is a plane

p = cs.pop();
p->takeoff();
```

The use of static type checking and virtual function calls leads to a somewhat different style of programming than does dynamic type checking and method invocation. For example, a Simula or C++ class specifies a fixed interface to a set of objects (of any derived class) whereas a Smalltalk class specifies an initial set of operations for objects (of any subclass). In other words, a Smalltalk class is a minimal specification and the user is free to try operations not specified whereas a C++ class is an exact specification and the user is guaranteed that only operations specified in the class declaration will be accepted by the compiler.

4.3 Inheritance

Consider a language having some form of method lookup without having an inheritance mechanism. Could that language be said to support object-oriented programming? I think not. Clearly, you could do interesting things with the method table to adapt the objects' behavior to suit conditions. However, to avoid chaos, there must be some systematic way of associating methods and the data structures they assume for their object representation. To enable a user of an object to know what kind of behavior to expect, there would also have to be some standard way of expressing what is common to the different behaviors the object might adopt. This ''systematic and standard way'' would be an inheritance mechanism.

Consider a language having an inheritance mechanism without virtual functions or methods. Could that language be said to support object-oriented programming? I think not: the shape example does not have a good solution in such a language. However, such a language would be noticeably more powerful than a ''plain'' data abstraction language. This contention is supported by the observation that many Simula and C++ programs are structured using class hierarchies without virtual functions. The ability to express commonality (factoring) is an extremely powerful tool. For example, the problems associated with the need to have a common representation of all shapes could be solved. No union would be needed. However, in the absence of virtual functions, the programmer would have to resort to the use of ''type fields'' to determine actual types of objects, so the problems with the lack of modularity of the code would remain†.

This implies that class derivation (subclassing) is an important programming tool in its own right. It can be used to support object-oriented programming, but it has wider uses. This is particularly true if one identifies the use of inheritance in object-oriented programming with the idea that a base class expresses a general concept of which all derived classes are specializations. This idea captures only part of the expressive power of inheritance, but it is strongly encouraged by languages where every member function is virtual (or a method). Given suitable controls of what is inherited (see Snyder[17] and Stroustrup[18]), class derivation can be a powerful tool for creating new types. Given a class, derivation can be used to add and/or subtract features. The relation of the resulting class to its base cannot always be completely described in terms of specialization; factoring may be a better term.

Derivation is another tool in the hands of a programmer and there is no foolproof way of predicting how it is going to be used − and it is too early (even after 20 years of Simula) to tell which uses are simply misuses.

4.4 Multiple Inheritance

When a class A is a base of class B, a B inherits the attributes of an A; that is, a B is an A in addition to whatever else it might be. Given this explanation it seems obvious that it might be useful to have a class B inherit from two base classes A1 and A2. This is called multiple inheritance[22].

A fairly standard example of the use of multiple inheritance would be to provide two library classes `displayed` and `task` for representing objects under the control of a display manager and co-routines under the control of a scheduler, respectively. A programmer could then create classes such as

```
class my_displayed_task : public displayed, public task {
    // my stuff
};

class my_task : public task {  // not displayed
    // my stuff
};

class my_displayed : public displayed {  // not a task
    // my stuff
};
```

Using (only) single inheritance only two of these three choices would be open to the programmer. This leads to either code replication or loss of flexibility − and typically both. In C++ this example can be handled as shown above with to no significant overheads (in time or space) compared to single inheritance and without sacrifysing static type checking[19].

† This is the problem with Simula's `inspect` statement and the reason it does not have a counterpart in C++.

Ambiguities are handled at compile time:

```
class A { public: f(); ... };
class B { public: f(); ... };
class C : public A, public B { ... };

void g() {
    C* p;
    p->f(); // error: ambiguous
}
```

In this, C++ differs from the object-oriented Lisp dialects that support multiple inheritance. In these Lisp dialects ambiguities are resolved by considering the order of declarations significant, by considering objects of the same name in different base classes identical, or by combining methods of the same name in base classes into a more complex method of the highest class.

In C++, one would typically resolve the ambiguity by adding a function:

```
class C : public A, public B {
public:
    f()
    {
        // C's own stuff
        A::f();
        B::f();
    }
    ...
}
```

In addition to this fairly straightforward concept of independent multiple inheritance there appears to be a need for a more general mechanism for expressing dependencies between classes in a multiple inheritance lattice. In C++, the requirement that a sub-object should be shared by all other sub-objects in a class object is expressed through the mechanism of a virtual base class:

```
class W { ... };

class Bwindow       // window with border
: public virtual W
{ ... };

class Mwindow       // window with menu
: public virtual W
{ ... };

class BMW       // window with border and menu
: public Bwindow, public Mwindow
{ ... };
```

Here the (single) `window` sub-object is shared by the `Bwindow` and `Bwindow` sub-objects of a `BMW`. The Lisp dialects provide concepts of method combination to ease programming using such complicated class hierarchies. C++ does not.

4.5 Encapsulation

Consider a class member (either a data member or a function member) that needs to be protected from ''unauthorized access''. What choices can be reasonable for delimiting the set of functions that may access that member? The ''obvious'' answer for a language supporting object-oriented programming is ''all operations defined for this object''; that is, all member functions. A non-obvious implication of this answer is that there cannot be a complete and final list of all functions that may access the protected member since one can always add another by deriving a new class from the protected member's class and define a member function of that derived class. This approach combines a large degree of protection from accident (since you do not easily define a new derived class ''by accident'') with the flexibility needed for ''tool building'' using class hierarchies (since you can ''grant yourself access'' to protected members by deriving

a class).

Unfortunately, the ‘‘obvious’’ answer for a language oriented towards data abstraction is different: ‘‘list the functions that needs access in the class declaration’’. There is nothing special about these functions. In particular, they need not be member functions. A non-member function with access to private class members is called a `friend` in C++. Class `complex` above was defined using `friend` functions. It is sometimes important that a function may be specified as a `friend` in more than one class. Having the full list of members and friends available is a great advantage when you are trying to understand the behavior of a type and especially when you want to modify it.

Here is an example that demonstrate some of the range of choices for encapsulation in C++:

```
class B {
                    // class members are default private
    int i1;
    void f1();
protected:
    int i2;
    void f2();
public:
    int i3;
    void f3();

friend void g(B*);   // any function can be designated as a friend
};
```

Private and protected members are not generally accessible:

```
void h(B* p)
{
    p->f1();     // error: B::f1 is private
    p->f2();     // error: B::f2 is protected
    p->f3();     // fine:  B::f1 is public
}
```

Protected members, but not private members are accessible to members of a derived class:

```
class D : public B {
public:
    void g()
    {
    f1();     // error: B::f1 is private
    f2();     // fine:  B::f2 is protected, but D is derived from B
    f3();     // fine:  B::f1 is public
    }
};
```

Friend functions have access to private and protected members just like member functions:

```
void g(B* p)
{
    p->f1();     // fine: B::f1 is private, but g() is a friend of B
    p->f2();     // fine: B::f2 is protected, but g() is a friend of B
    p->f3();     // fine: B::f1 is public
}
```

Encapsulation issues increase dramatically in importance with the size of the program and with the number and geographical dispersion of its users. See Snyder[17] and Stroustrup[18] for more detailed discussions of language support for encapsulation.

4.6 Implementation Issues

The support needed for object-oriented programming is primarily provided by the run-time system and by the programming environment. Part of the reason is that object-oriented programming builds on the language improvements already pushed to their limit to support for data abstraction so that relatively few additions are needed†.

† This assumes that an object-oriented language does indeed support data abstraction. However, the support for data abstraction is often deficient in such languages. Conversely, languages that support data abstraction are typically deficient in their support of object-

The use of object-oriented programming blurs the distinction between a programming language and its environment further. Since more powerful special- and general-purpose user-defined types can be defined their use pervades user programs. This requires further development of both the run-time system, library facilities, debuggers, performance measuring, monitoring tools, etc. Ideally these are integrated into a unified programming environment. Smalltalk is the best example of this.

5 Limits to Perfection

A major problem with a language defined to exploit the techniques of data hiding, data abstraction, and object-oriented programming is that to claim to be a general purpose programming language it must

[1] Run on traditional machines.

[2] Coexist with traditional operating systems.

[3] Compete with traditional programming languages in terms of run time efficiency.

[4] Cope with every major application area.

This implies that facilities must be available for effective numerical work (floating point arithmetic without overheads that would make Fortran appear attractive), and that facilities must be available for access to memory in a way that allows device drivers to be written. It must also be possible to write calls that conform to the often rather strange standards required for traditional operating system interfaces. In addition, it should be possible to call functions written in other languages from a object-oriented programming language and for functions written in the object-oriented programming language to be called from a program written in another language.

Another implication is that an object-oriented programming language cannot completely rely on mechanisms that cannot be efficiently implemented on a traditional architecture and still expect to be used as a general purpose language. A very general implementation of method invocation can be a liability unless there are alternative ways of requesting a service.

Similarly, garbage collection can become a performance and portability bottleneck. Most object-oriented programming languages employ garbage collection to simplify the task of the programmer and to reduce the complexity of the language and its compiler. However, it ought to be possible to use garbage collection in non-critical areas while retaining control of storage use in areas where it matters. As an alternative, it is feasible to have a language without garbage collection and then provide sufficient expressive power to enable the design of types that maintain their own storage. C++ is an example of this.

Exception handling and concurrency features are other potential problem areas. Any feature that is best implemented with help from a linker is likely to become a portability problem.

The alternative to having ‘‘low level’’ features in a language is to handle major application areas using separate ‘‘low level’’ languages.

6 Conclusions

Object-oriented programming is programming using inheritance. Data abstraction is programming using user-defined types. With few exceptions, object-oriented programming can and ought to be a superset of data abstraction. These techniques need proper support to be effective. Data abstraction primarily needs support in the form of language features and object-oriented programming needs further support from a programming environment. To be general purpose, a language supporting data abstraction or object-oriented programming must enable effective use of traditional hardware.

7 Acknowledgements

An earlier version of this paper was presented to the Association of Simula Users meeting in Stockholm. The discussions there caused many improvements both in style and contents. Brian Kernighan and Ravi Sethi made many constructive comments. Also thanks to all who helped shape C++.

---

oriented programming.

8 References

[1] Birtwistle, Graham et.al.: *SIMULA BEGIN*. Studentlitteratur, Lund, Sweden. 1971. Chartwell-Bratt ltd, UK. 1980.

[2] Bobrow, D. and Stefik, M.: *The LOOPS Manual*. Xerox Parc 1983.

[3] Dahl, O-J. and Hoare, C.A.R.: *Hierarchical Program Structures*. In *Structured Programming*. Academic Press 1972.

[4] Cargill, Tom A.: *PI: A Case Study in Object-Oriented Programming*. SIGPLAN Notices, November 1986, pp 350-360.

[5] C.C.I.T.T Study Group XI: *CHILL User's Manual*. CHILL Bulletin no 1. vol 4. March 1984.

[6] Goldberg, A. and Robson, D.: *Smalltalk-80: The Language and its Implementation*. Addison-Wesley 1983.

[7] Ichbiah, J.D. et.al.: *Rationale for the Design of the Ada Programming Language*. SIGPLAN Notices, June 1979.

[8] Kernighan, B.W. and Ritchie, D.M.: *The C Programming Language*. Prentice-Hall 1978.

[9] Kerr, Ron: *Object-Based Programming: A Foundation for Reliable Software*. Proceedings of the 14th SIMULA Users' Conference. August 1986, pp 159-165. An abbreviated version of this paper can be found under the title *A Materialistic View of the Software ''Engineering'' Analogy* in SIGPLAN Notices, March 1987, pp 123-125.

[10]Liskov, Barbara et. al.: *Clu Reference Manual*. MIT/LCS/TR-225, October 1979.

[11]Liskov, Barbara et. al.: *Abstraction Mechanisms in Clu. CACM vol 20, no 8, August 1977, pp 564-576.*

[12]Milner, Robert: *A Proposal for Standard ML*. ACM Symposium on Lisp and Functional Programming. 1984, pp 184-197.

[13]Nygaard, Kristen: *Basic Concepts in Object Oriented Programming*. SIGPLAN Notices, October 1986, pp 128-132.

[14]Rovner, Paul: *Extending Modula-2 to Build Large, Integrated Systems*. IEEE Software, Vol. 3. No. 6. November 1986, pp 46-57.

[15]Shopiro, Jonathan: *Extending the C++ Task System for Real-Time Applications*. Proc. USENIX C++ Workshop, Santa Fe, November 1987.

[16]SIMULA Standards Group, 1984: *SIMULA Standard*. ASU Secretariat, Simula a.s. Post Box 150 Refstad, 0513 Oslo 5, Norway.

[17]Snyder, Alan: *Encapsulation and Inheritance in Object-Oriented Programming Languages*. SIGPLAN Notices, November 1986, pp 38-45.

[18]Stroustrup, Bjarne: *The C++ Programming Language*. Addison-Wesley, 1986.

[19]Stroustrup, Bjarne: *Multiple Inheritance for C++*. Proceedings of the Spring'87 EUUG Conference. Helsinki, May 1987.

[20]Stroustrup, Bjarne: *The Evolution of C++: 1985-1987*. Proc. USENIX C++ Workshop, Santa Fe, November 1987.

[21]Stroustrup, Bjarne: *Possible Directions for C++: 1985-1987*. Proc. USENIX C++ Workshop, Santa Fe, November 1987.

[22]Weinreb, D. and Moon, D.: *Lisp Machine Manual*. Symbolics, Inc. 1981.

[23]Wirth, Niklaus: *Programming in modula-2*. Springer-Verlag, 1982.

[24]Woodward, P.M. and Bond, S.G.: *Algol 68-R Users Guide*. Her Majesty's Stationery Office, London. 1974.

BIBLIOGRAPHY

[1] Grady Booch. *Object–Oriented. Design, with Applications*. Redwood City, CA, USA: Benjamin/Cummings, 1991.

[2] Peter Coad and Edward Yourdon. *Object–Oriented Analysis*. 2nd. Englewood Cliffs, NJ, USA: Yourdon Press; Prentice Hall, 1990.

[3] Donald G. Firesmith. *Oriented–Object Requirements Analysis and Logical Design*. 1st. Wiley Professional Computing. Wiley, 1996.

[4] Ivar Jacobson, Magnus Christerson, Patrick Jonsson, and Gummar Övergaard. *Object Oriented Software Engineering: A Use Case Driven Approach*. Reading, MA, USA: Addison–Wesley, 1992.

[5] James Martin and James Odell. *Object Oriented Analysis and Design*. Englewood Cliffs, NJ, USA: Prentice Hall, 1992.

[6] James Martin and James Odell. *Object Oriented Design. A Foundation*. Englewood Cliffs, NJ, USA: Prentice Hall, 1994.

[7] OMG. *Object Analysis and Design, Volume 1, Reference Model*. Draft 7.0. Framinghan, MA, USA: Hartley & Marks Publishers, 1992.

[8] Sally Shlaer and Stephen Mellor. *Object Lifcycles. Modeling the World in States*. Englewood Cliffs, NJ, USA: Yourdon Press; Prentice Hall, 1992.

[9] V.A. *Webster's New World Dictionary*. New York: Simon and Shuster, 1980.