PARADIGMAS DE PROGRAMACIÓN

Lic. María I. Mentz de Acosta
Lic. Griselda María Luccioni

Facultad de Ciencias Exactas y Tecnología
Universidad Nacional de Tucumán

PARADIGMAS DE PROGRAMACIÓN

I.1 Introducción

Un **paradigma de programación** es un modelo básico de diseño e implementación de programas, que permite producir programas conforme con directrices específicas, tales como: estructura modular, fuerte cohesión, alta reusabilidad, etc.

Aunque la mayoría de los programadores están familiarizados con el paradigma de la programación procedimental, existen diferentes paradigmas, atendiendo a alguna peculiaridad metodológica o funcional, por ejemplo:

Paradigma basado en reglas: De gran aplicación en la Ingeniería del conocimiento para el desarrollo de sistemas expertos, en el que el núcleo del mismo son las reglas de producción "If-then".
Paradigma Lógico: Basado en asertos y reglas lógicas, que define un entorno de programación de tipo conversacional, deductivo, simbólico y no determinista.
Paradigma Funcional: Basado en funciones, formas funcionales para crear funciones y mecanismos para aplicar los argumentos y que define un entorno de programación interpretativo funcional y aplicativo.
Paradigma de Programación Heurística: Que aplica para la resolución de problemas "reglas de buena lógica" (heurísticas) que presentan visos de ser correctas aunque no se garantiza su éxito, modelizando el problema de una forma adecuada para aplicar esta heurísticas atendiendo a su representación, estrategias de búsqueda y métodos de resolución.
Paradigma de Programación Paralela
Paradigma basado en restricciones
Paradigma basado en el flujo de datos, **Paradigma orientado a objetos**, etc.

> Un **paradigma de programación** es, pues, una colección de modelos conceptuales que juntos modelan el proceso de diseño y determinan, al final, la estructura de un programa.

Esa estructura conceptual de modelos está pensada de forma que esos modelos determinan la forma correcta de los programas y controlan el modo en que pensamos y formulamos soluciones, e incluso si alcanzamos la solución. Una vez que visualizamos una solución vía modelos conceptuales de un paradigma, debemos expresarlo con un lenguaje de programación, cuyas características reflejen adecuadamente los modelos conceptuales de ese paradigma. Es decir que **el lenguaje soporta el paradigma**. En la práctica, un lenguaje que soporta correctamente un paradigma es difícil de distinguir del propio paradigma, por lo que se identifica con él.

I.2 Tipos de Paradigmas de Programación

Existen, de acuerdo a Floyd, tres categorías de paradigmas de programación:

a) Los que soportan técnicas de programación de **bajo nivel** (Por ejemplo: copias de archivos vs. estructuras de datos compartidos).

b) Los que soportan métodos de **diseño de algoritmos** (por ejemplo: divide y conquistarás, programación dinámica, etc.)

c) Los que soportan soluciones de programación de **alto nivel** como los descriptos en el punto anterior (Por ejemplo: paradigmas basados en reglas, lógico, funcional, etc.)

Floyd también señala lo diferentes que resultan los lenguajes de programación que soportan cada una de estas categorías de paradigmas. Nos centraremos en el nivel más relevante: los paradigmas que soportan la programación de alto nivel.

De acuerdo a la solución que aportan para resolver el problema, se agrupan en **tres categorías**:

i) **Solución procedimental u operacional**: Describe etapa a etapa el modo de construir la solución. Señala **como** obtener la solución.

ii) **Solución demostrativa**: Es una variante de la procedimental. Especifica la solución describiendo ejemplos y permitiendo que el sistema generalice la solución de esos ejemplos para otros casos.

iii) **Solución Declarativa**: Señala las características que debe tener la solución, sin describir como procesarla. Es decir, señala **que** se desea obtener pero **no como** obtenerlo.

I.2.1 Paradigmas Procedimentales u Operacionales

La característica principal de estos paradigmas es la secuencia computacional realizada etapa a etapa para resolver el problema. Su mayor dificultad reside en determinar si el valor computado es una solución correcta del problema, por lo que se han desarrollado muchísimas técnicas de depuración y verificación para probar la corrección de los problemas implementados con este tipo de paradigmas.

Básicamente, los paradigmas procedimentales son de dos tipos:

i) Los que actúan modificando repetidamente la representación de sus datos (**efecto lateral**). Usan un modelo en el que las variables están estrechamente relacionadas con direcciones de memoria de la computadora. Cuando se ejecuta el programa, el contenido de estas direcciones es actualizado repetidamente (las variables reciben múltiples asignaciones) y cuando finaliza la ejecución, los valores finales de las variables representan el resultado.
Existen dos paradigmas con efecto lateral:

- **Paradigma Imperativo**
- **Paradigma Orientado a Objetos**

ii) Los que actúan creando continuamente nuevos datos (**sin efecto lateral**). Incluyen a los **paradigmas funcionales**, aunque es importante distinguir la solución funcional procedimental de la solución funcional declarativa.

Los paradigmas procedimentales definen la secuencia explícitamente, pero esa secuencia se puede procesar en **serie** o en **paralelo**. Si se procesa en paralelo, este procesamiento puede ser **asíncrono** (cooperación de procesos paralelos) o **síncrono** (procesos simples aplicados simultáneamente a muchos objetos).

I.2.2 Paradigmas Declarativos

En este tipo de paradigma, un programa se construye señalando hechos, reglas, restricciones, ecuaciones, transformaciones y otras propiedades derivadas del conjunto de valores que configuran la solución.

A partir de esta información se debe proporcionar un esquema que incluyan el orden de evaluación que compute una solución. No se describen las diferentes etapas a seguir para alcanzar una solución.

Se usan variables para almacenar valores intermedios pero no para actualizar estados de información.

Estos paradigmas especifican la solución sin indicar como construirla, por lo tanto eliminan la necesidad de probar que el valor computado es el valor solución. Pero en la práctica, aunque muchos paradigmas no tienen secuencia de control y efecto lateral que requiera la noción de estado, las soluciones son todavía producidas como construcciones más que como especificaciones. Por lo que los paradigmas resultantes y los lenguajes que los soportan no son verdaderamente declarativos sino **pseudo-declarativos**. En este grupo se encuentran los paradigmas **funcional**, **lógico** y el **de transformación**.

Los paradigmas **basados en formas** y el de **flujo de datos** evitan estas dificultades a través de restricciones sobre la forma de las ecuaciones permitidas y con los modelos empleados de evaluación por dependencias.

El paradigma **basado en restricciones** evita las secuencias de control con las mismas restricciones que incorpora.

Como los paradigmas declarativos no dirigen la secuencia de control no son necesariamente soluciones de tipo serie o paralelo. Sin embargo, los paradigmas pseudo-declarativos requieren al menos un limitado grado de secuencia y por lo tato admiten versiones en serie o en paralelo.

I.2.3 Paradigmas Demostrativos

Cuando se programa bajo un paradigma demostrativo ("Programación por ejemplos" o "Por demostración") se presentan soluciones a problemas similares y se permite que el sistema generalice una solución procedimental a partir de estas demostraciones. Los esquemas individuales para generalizar tales soluciones van desde simular una secuencia procedimental o inferir intenciones. El programador no especifica procedimentalmente como construir una solución.

Los sistemas que hacen inferencias intentan generalizar usando razonamiento basado en conocimiento. Este tipo de solución intenta determinar en qué son similares un grupo de datos u objetos, y a partir de ello, generalizar estas similitudes.

Otra solución es la programación asistida: en sistema observa acciones que el programador ejecuta, y si son similares o acciones pasadas intentará inferir cuál es la próxima acción que tomará el programador.

Las dos principales **objeciones** a los sistemas de inferencia son:

a) Si no se comprueba exhaustivamente pueden producir programas erróneos que trabajen correctamente con los ejemplos de prueba pero que fallen con otros ejemplos.

b) La capacidad de inferencia es tan limitada que el usuario debe guiar el proceso en la mayoría de los casos.

Las áreas donde los sistemas de inferencia ofrecen mayores resultados son aquellas limitadas donde el sistema tenía un conocimiento semántico importante de la aplicación.

El principal problema que presentan los sistemas demostrativos es conocer cuando un sistema es correcto. El algoritmo se mantiene en una representación interna y su estudio se sale del ámbito de estos sistemas. La veracidad de la decisión se hace exclusivamente sobre la base de la eficiencia del algoritmo sobre casos específicos de prueba.

La solución demostrativa es "bottom-up".

I.3 Paradigma Imperativo

Está encuadrado en el tipo procedimental con efecto lateral. Este paradigma se caracteriza por un modelo abstracto de computadora que tiene una gran capacidad de almacenamiento o memoria.

La computadora almacena una representación codificada de una computación y ejecuta una secuencia de comandos que modifican el contenido de ese almacenamiento. (En este sentido, el paradigma está representado por la arquitectura Von Neumann, ya que usa un modelo de máquina para conceptualizar la solución "Existe un programa en memoria que se va ejecutando secuencialmente, que toma datos de memoria, efectúa cálculos y actualiza la memoria").

Programar en este paradigma consiste en determinar qué datos son requeridos para el cálculo, asociar a esos datos unas direcciones de memoria, y efectuar, paso a paso, una secuencia de transformaciones en los datos almacenados, de forma tal que el resultado final represente el resultado correcto.

En su forma pura, este paradigma únicamente soporta sentencias simples que modifican la memoria y efectúan bifurcaciones condicionales o incondicionales. Incluso, al añadir una forma simple de abstracción procedimental, el modelo no cambia ya que los parámetros de los procedimientos son "alias" de zonas de memoria.

El paradigma imperativo recibe su nombre debido al papel dominante que desempeñan las sentencias imperativas. Su esencia es el cálculo iterativo, paso a paso, de valores de nivel inferior y su asignación a posiciones memoria.

Las **características principales** de este tipo de paradigma son:

- **Concepto de celda de memoria ("variable") para almacenar valores**: Ya que el principal componente de la arquitectura es la memoria, compuesta por un gran número de celdas donde se almacenan los datos. Las celdas tienen nombres que las referencian y sobre las que se producen efectos laterales y definiciones de alias.

- **Operaciones de asignación**: Estrechamente ligado a la arquitectura de la memoria se encuentra la idea de que cada valor calculado debe ser almacenado, (asignado a una celda). La sentencia de asignación es de suma importancia en el paradigma, se extiende a todos los lenguajes de programación y fuerzan al programador a un estilo de pensamiento adaptado al modelo Von Neumann.

- **Repetición**: Un programa imperativo normalmente realiza su tarea ejecutando repetidamente una secuencia de pasos elementales.

Este paradigma recibe a veces también el nombre de **paradigma algorítmico** ya que se entiende que el concepto de algoritmo es privativo de la programación procedimental porque su característica principal es la secuencia computacional.

Así pues, Según N. Wirth, un programa viene definido por la ecuación:

Algoritmos + Estructuras de datos = Programas

Los lenguajes imperativos se clasifican en:

- Orientados a expresiones:	Es un dominio simple, regular y jerárquico: Fortran, Algol, Pascal, C, etc.
- Orientados a sentencias:	Está más influenciado por las características de la máquina: Cobol y PL/1.

EJEMPLOS

1. Generar los números primos en el rango 2..n usando la criba de Eratóstenes.

```
Algoritmo: primos
Datos:
Salida: i:entero (2..n)
Constantes: n = 50
Variables: iprimo: booleana; j: entero (2..25)

p1. HACER (n-2+1) VECES (para i de 2 a n)
        p1.1 calcular-primos
p2. Parar
```

p1.1. calcular-primos

```
p1.1.1 j <-- 2
p1.1.2 iprimo <-- verdadero
p1.1.3 MIENTRAS iprimo y (j <= i DIV 2) HACER
                SI ((i MOD j) <>  0) ENTONCES
                                j <-- j + 1
                SINO
                                iprimo <-- falso
p1.1.4 SI iprimo ENTONCES
                ESCRIBIR(i)
```

I.4 Paradigma Funcional

Este paradigma se basa en el modelo matemático de composición de funciones. El resultado de un cálculo es la entrada del siguiente, y así sucesivamente hasta que una composición produce el resultado deseado.

No existe el concepto de celda de memoria que es asignada o modificada, más bien existen valores intermedios que son el resultado de cálculos anteriores y las entradas a cálculos subsiguientes. Tampoco existen sentencias imperativas y todas las funciones tienen transparencia referencial.

La programación funcional incorpora el concepto de función como objeto de primera clase, es decir que las funciones pueden ser tratadas como datos (se pueden pasar como parámetros, pueden calcularse y ser devueltas como valores normales y mezclarse en el cálculo con otras formas de datos).

La solución se concibe como una composición de funciones. Por ejemplo, para ordenar una lista, se puede diseñar la solución como una concatenación de listas más pequeñas, cada una de las cuales ya está clasificada.

La forma en que se especifican las funciones puede variar, ya sea especificándolas procedimentalmente o matemáticamente mediante su definición sin secuencia de control.

En la **solución procedimental** se controla explícitamente el orden en que se efectúa el cálculo (para ordenar una lista primero se determina si está vacía o no, y se aplica el tratamiento si no lo está). Esto origina una sobrespecificación que caracteriza a este tipo de solución. En la práctica, los lenguajes que soportan este paradigma incluyen

construcciones imperativas que destruyen el natural no-efecto lateral del mismo y obligan a considerar la secuencia del programa en su construcción.

En cambio, en el **estilo declarativo** se define el problema como una colección de transformaciones disjuntas que, tomadas colectivamente, definen una función computacionalmente.

A menudo, este modelo declarativo de tipo matemático usa evaluación diferida (evaluación no estricta) en la que los argumentos de la función no son evaluados hasta que son usados. Esto se contrapone con la evaluación convencional (llamada "ansiosa") que evalúa todos los argumentos antes de invocar a la función. Dado que la evaluación diferida se realiza cuando se procesa un determinado valor, es que se pueden efectuar definiciones de funciones que generen secuencias infinitas.

> **Un programa funcional es un conjunto compuesto de funciones que juntas definen un objetivo específico de aplicación.**

Desde el punto de vista informático, una función es una pieza individual de codificación que siempre efectúa el mismo conjunto de operaciones pero que puede operar con distintos datos, y proporciona algún tipo de resultado, llamado "valor de la función".

Un programa funcional se define por la ecuación:

Funciones + Estructuras de Datos = Programas

Los elementos fundamentales de este paradigma son:

- **Las funciones**: Son reglas de correspondencia o asociación entre miembros de un conjunto (el dominio) y miembros de otro conjunto (el rango). La definición de una función especifica el dominio, el rango y la regla de correspondencia para la función. Por ejemplo:

  cuadrado(x) = x2 para todo x perteneciente a los reales

- **La recursividad**: que es la iteración con asignación no destructiva. este mecanismo, especialmente utilizado en la construcción de funciones, es de exigencia en la programación funcional.

- **Las listas**: Son las estructuras básicas de almacenamiento de información, aunque la programación funcional también usa otros tipos de estructuras. La expresión algebraica carece de la relevancia que tiene en la programación imperativa.

Ya que el mecanismo de operación de un programa funcional consiste en la ejecución sucesiva de funciones, algunas ya definidas por el software del sistema y otras que se van incorporando al programa, este tipo de programación usa un **proceso interpretativo**, mientras que la programación imperativa se basa en lenguajes que usan procesos de compilación, la programación funcional utiliza lenguajes de tipo intérprete.

Peter Bishop (1986) señala que los lenguajes funcionales son aplicativos porque aplican funciones y pertenecen al conjunto de los denominados lenguajes de programación funcional surgidos del cálculo de la función lambda de Church. Un programa en un lenguaje funcional es considerado como un conjunto de descripciones de objetos de datos, más que como un conjunto de almacenamiento de datos, que proporciona los resultados del cómputo de esos datos.

En concreto, los lenguajes funcionales pseudodeclarativos, tipo LISP, HOPE, etc. constan de:

- **Funciones Primitivas** predefinidas en el lenguaje: de control, de estructuras, aritméticas....
- **Formas funcionales** para crear y combinar funciones
- **Mecanismos** para aplicar los argumentos y producir valores de funciones
- **Objetos de datos**, de tipo atómico, listas, etc.

Así, por ejemplo, el lenguaje LISP ("List Procesing) diseñado por John McCarthy, en 1959, en MIT (Massachusetts Technological Institute) incorpora, en su expresión más simple, los siguientes elementos:

a) Funciones primitivas predefinidas:

i) Tratamiento de listas

CAR - Devuelve la cabeza de la lista
(CAR '((A B) C)) --> (AB)
CDR _ Devuelve la cola de la lista
(CDR '((A B) C)) --> (C)

(CADDR '((A B C) D E)) --> E
(es un CDR, CDR y CAR)
CONS - Construye una lista CONS a partir de dos listas
(CONS ' A ' (B C)) --> (A . (B C)) --> (A B C)

ii) Asignación, relaciones booleanas y predicados

SET - Evalúa los dos argumentos y asigna el valor del 2o. al primero. Función no declarativa, porque realiza una asignación destructiva.
SETQ - Evalúa el 2o. argumento y lo asigna al primero. También es no declarativa.

(SETQ x ' programacion) --> asigna el átomo programación a la variable x.

EQ - Comprueba si dos argumentos son iguales.

(EQ x ' programacion) --> T

GREATERP y LESSP - Comparan si el primer argumento es mayor/menor que el segundo.

NULL - Comprueba si la lista es nula.

(NULL x) --> F

ATOM, LISTP y NUMBERP - Averiguan el tipo (átomo, listas o número) de un objeto.

(ATOM x) --> T

OR, AND y NOT - Predicados de suma, multiplicación y negación booleana.

(NOT (OR (eq 3 4) (EQ x ' programacion))) --> NIL

iii) Artiméticas

PLUS, DIFFERENCE, TIMES, QUOTIENT, REMAINDER

(Suma, resta, multiplicación, cociente y resto)

(PLUS 3 4) --> 7
(DIFFERENCE 10 20) --> -10
(TIMES 3 -15) --> -45
(QUOTIENT 17 3) --> 5
(REMAINDER 17 3) --> 2

iv) Condicionales e iterativas

COND - Función condicional con varios predicados. Su formato es:
(COND (predicado1 exp1) ... (predicadon expn))

Si se verifica un predicado se ejecuta la expresión y finaliza la función condicional.

LOOP - Función iterativa no declarativa, incorporada posteriormente. Sus formatos son:

.(LOOP secuencia-operaciones)
.(LOOP (UNTIL predicado exp) (secuencia-operaciones))
Hasta que el predicado no sea igual a la exp se ejecuta la secuencia-operaciones. La pregunta UNTIL se hace al finalizar cada iteración.
.(LOOP (WHILE predicado exp) (secuencia-operaciones)
Mientras el predicado verifique la exp se ejecuta la secuencia-operaciones. La pregunta WHILE se realiza el principio de la iteración del LOOP.

EJEMPLOS:
(COND ((EQ A B) 20) (T 30)
SI A = B ENTONCES resultado = 20 SINO resultado = 30

```
(SETQ Y 0)
(LOOP (UNTIL (GREATERP Y 100)
      (SETQ Y (PLUS Y 5))
      (PRINT Y)
```

Imprime 5 10 15 ... 100 105

b) Formas funcionales para crear y combinar funciones

DEFUN - Función que permite crear otras funciones. Su formato es:

(DEFUN nom-fun (lista de argumentos) cuerpo-fun)

(DEFUN DOBLE(X) (TIMES X 2)

c) Objetos de datos

El LISP como lenguaje orientado a listas dispone de esta estructura como fundamental. En general, un programa LISP usa:

. **Átomos**: Son cadenas de números o caracteres

28, programacion, -27, valor_total

. **Identificadores**: Son átomos no numéricos. Pueden usarse como nombres de funciones o de variables.

X, DOBLE, Y25

. **Listas**: Son átomos y/o sublistas encerrados entre paréntesis.

(20 4 35) (X Y (A B (C) D E))

EJEMPLOS

1. Determinar si un número es primo o no.

```
(DEFUN PRIMO(N))
      (COND((EQ N 2)T)((EQ N 3) T)
            (T (PRIMO1 N (QUOTIENT N 2)))))
(DEFUN PRIMO1 (N I)
      (COND((EQ (REMAINDER N I)0)NIL)
            (EQ I 2)T)
            (T (PRIMO1 N(DIFFERENCE I 1))))
```

Ejecución:

(PRIMO 3) --> T
(PRIMO 8) --> NIL

El LISP puro definido por McCarthy es un buen representante del paradigma funcional. Sin embargo, sus versiones mejoradas, incluido el COMMONLISP soportan igualmente la programación imperativa, al incluir conceptos tales como: efecto lateral

con modificación destructiva de su contenido, tipos de datos con almacenamiento fijo (arrays y otros objetos manipulados básicamente con efectos laterales) y constructores del flujo de control (iteraciones y manejo de excepciones).

En consecuencia, el entorno de programación que proporcionan estos lenguajes es del tipo **pseudodeclarativo** (con componente declarativo y procedimental, en el que la secuencia de control juega un papel importante). Por otra parte son lenguajes **intepretativos**, aunque también admiten la compilación de funciones desde el punto de vista de su optimización y **aplicativos**, ya que se basan en la aplicación sucesiva de estas funciones.

I.5 Paradigma Lógico

Este paradigma asume la definición de un **conjunto de hechos** (o asertos) (Ej: Vicente es padre) y un **conjunto de reglas** que permiten deducir nuevos hechos (Ej. para todo y, Si y es padre, y es varón. Con el aserto anterior se puede deducir que Vicente es varón). Así pues, desde la perspectiva del programador, la programación lógica es una técnica que consiste en expresar adecuadamente todos los hechos y reglas necesarios que definen un problema.

El paradigma lógico fue creado por Robert Kowalski en el Imperial College de Londres y lo implementó en base a las cláusulas de Horn, que son un subconjunto de la lógica de predicados de primer orden.

La notación clausal de la lógica de predicados combina variables, constantes y expresiones para expresar proposiciones condicionales tales como:

casado(X,Y) SI marido (Y,X) o esposa(X,Y)

Las cláusulas de Horn son una forma restrictiva de lógica de predicados con una sola conclusión en cada cláusula.

> **La programación lógica es una instrucción de componentes lógicos de un algoritmo, siendo el sistema el que proporciona la secuencia de control.**

Al separar el control y la lógica, el programa se transforma en un conjunto de declaraciones formales de especificaciones que deben ser correctas por definición, por lo que la corrección del programa debería estar probada automáticamente.

En este tipo de programación, la evaluación empieza definiéndose una meta e intentando probar que ella se ajusta a un hecho o se deduce de alguna regla.

Una **meta** se deduce de una regla si todos los antecedentes de la regla se verifican con la transformación existente; para lo cual, estos antecedentes se convierten en nuevas metas que deben equipararse con hechos o resolverse vía otras reglas. El proceso termina cuando todas las submetas han sido probadas. Y la solución final viene determinada al aplicar los resultados intermedios obtenidos a las variables de la meta inicial.

Si se evalúa según la descripción anterior, se tiene una evaluación "**puramente declarativa**". Y asume que si se selecciona una regla, entonces o bien existe esa posibilidad o la regla que se necesita para alcanzar la solución es la que de algún modo ha sido seleccionada. Se alcanza la solución si existe un conjunto apropiado de reglas y sustituciones, tales que aplicando las sustituciones a las reglas se obtiene un conjunto de reglas base (sin variables libres) que permiten deducir las metas desde los hechos conocidos.

El principal problema se presenta para definir el mecanismo de búsqueda adecuado para seleccionar las reglas, siendo la variante más popular la búsqueda "primero en profundidad" con un proceso de "backtracking". Este algoritmo, cuando necesita seleccionar una regla, selecciona la primera que encuentra, si ella conduce a un punto muerto, selecciona la segunda, y así hasta que todas las posibilidades han sido probadas.

Ordenando cuidadosamente tanto los hechos como las reglas, se puede aumentar la eficiencia de la selección y su terminación, pero esta acción transformaría el paradigma en un pseudodeclarativo. Otra ejecución no declarativa es el uso de cortes, que son dispositivos no lógicos que inhiben el mecanismo de backtracking y fueron introducidos para aumentar la eficiencia de los mecanismos de búsqueda.

Hoy en día, la programación lógica se ha convertido en una programación de propósito general. La lógica pseudodeclarativa se usa para producir programas en la industria informática donde la descripción de hechos y de reglas es apropiada para ciertos tipos de problemas tales como: Algoritmos de búsqueda con backtracking con múltiples soluciones, problemas que se describen de forma natural mediante reglas de producción (ej: la traducción a lenguaje natural), y para la definición e implementación de especificaciones ejecutables mediante prototipos rápidos.

Un programa lógico se configura como un conjunto de hechos (asertos o proposiciones) y de reglas lógicas previamente establecidas, que obtienen conclusiones en base a una serie de preguntas o cuestiones lógicas. Un programa viene definido por la ecuación:

Lógica + Control + Estructuras de datos = Programa

Donde:

Lógica - Está constituida por los asertos y reglas lógicas.

Control - Inherente al sistema, son las estrategias a seguir para investigar las cuestiones lógicas (ej: estrategia de búsqueda "primero en profundidad").

Estructuras de Datos - Son los elementos que soportan la base de conocimiento y cuestiones lógicas (variables, constantes, listas,...)

Los lenguajes lógicos, cuyo componente paradigmático es el PROLOG, diseñado y desarrollado por Alain Colmerauner y sus colegas en la universidad de Marsella en 1972, constan fundamentalmente de:

- Predicados para la declaración de aserciones
- Cláusulas para definir reglas lógicas
- Mecanismos de búsqueda y reconocimiento para resolver cuestiones lógicas
- Estructuras de datos, tales como átomos (que representan valores de instancias), variables, listas, etc.

En el caso del lenguaje PROLOG, estos elementos se implementan básicamente de la siguiente forma:

a) **Predicados**: Son instrucciones que son ciertas o falsas. Se pueden formar con un argumento o con varios. Sus formatos son:

1) Predicado(argumento).
2) Predicado (arg1,arg2,..., argn).

EJEMPLOS

programador(luis).	Luis es programador
pez(sardina).	La sardina es un pez
hijo(luis,fernando,maría).	Luis es hijo de Fernando y María
casado(fernando,maría).	Fernando está casado con María

b) **Cláusulas**: Definen reglas lógicas que permiten inferir otros conceptos al aplicarlas a una Base de Conocimiento que incorpora aserciones y reglas lógicas. Su formato es:

<parte-izquierda-regla> :- <parte-derecha>.

EJEMPLOS

hombre(X) :- varon(X), adulto(X).
X es un hombre Si es varón y adulto

progenitor(X,Y) :- padre(X,Y); madre(X,Y).
X es progenitor de Y si X es el padre o la madre de Y

c) **Mecanismos de Búsqueda**: En PROLOG la búsqueda en la Base de Conocimiento se efectúa de arriba a abajo, usando el algoritmo de "primera búsqueda en profundidad" con aplicación de backtracking. Las reglas se resuelven de izquierda a derecha.

EJEMPLO

Sea la Base de Conocimiento formada por los siguientes asertos y reglas lógicas.

empleado(juan).

```
empleado(luis).
secretaria(rosa).
jefe(pedro).
jefe(julio).
supervisor(X,Y) :- jefe(X),empleado(Y).
supervisor(X,Y) :- jefe(X),secretaria(Y).
```

Si se realiza la pregunta:

```
? - supervisor(X,Y)
```

la respuesta será:

```
X=pedro, Y=juan
X=pedro, Y=luis
X=julio, Y=juan
X=julio, Y=luis
X=pedro, Y=rosa
X=julio, Y=rosa
```

Si se efectúa un mecanismo de corte con el operador ! (cut) en las dos reglas anteriores, para suprimir el mecanismo de backtracking, de la siguiente manera:

```
supervisor(X,Y) :- jefe(X),empleado(Y),!.
supervisor(X,Y) :- jefe(X),secretaria(Y),!.
```

la respuesta sería:

```
X=pedro, Y=juan
X=pedro, Y=rosa
```

d) **Estructuras de datos**: Los tipos de datos simples que usa PROLOG son:

. Constantes: Normalmente incorpora átomos que son nombres simbólicos (luis, juan, etc.), literales ('hola') y valores enteros (20, 30, 100).
. Variables: Que empiezan, en algunas versiones, con mayúsculas o subrayado.
. Listas: Que se representan con la notación entre corchetes: [a, b, c, d], [rojo, verde, amarillo].

En PROLOG; al igual que en LISP es posible seleccionar la cabeza o la cola de una lista, mediante el operador l de la forma:

```
[cabeza l resto]
```

Así, dada la aserción: vocales([a, e, i, o ,u])
se podrán ejecutar operaciones tales como:

```
? - vocales(X).
```

```
X = [a, e, i, o, u]
? - vocales([H l T]).
H = a
T = [e, i, o, u]
? - vocales ([X l _ ])
X = a
```

En resumen, un programa PROLOG se configura como un conjunto de asertos y reglas lógicas, incorporadas a una Base de Conocimiento mediante predicados de la forma:

```
assert(empleado(juan)).
assert(supervisor(X,Y):-jefe(X),empleado(Y)).
```

que usando estrategias de búsqueda diseñadas en el propio sistema, permite contestar a cuestiones lógicas (preguntas), efectuadas interactivamente por el usuario, sobre la base del conocimiento almacenado.

Aunque PROLOG parece ser sinónimo de programación lógica y, por derivación, de programación declarativa, en la actualidad incluye estrategias específicas para la resolución de problemas y una serie de características básicas que no se corresponden con el paradigma declarativo. Mientras sus características lógicas puras implementan el paradigma declarativo, sus otras prestaciones pueden ser usadas de forma tal que encuadrarían en el paradigma operacional, y por ello es considerado un lenguaje pseudo-declarativo.

En general, el entorno actual de la programación lógica presenta las características de ser pseudodeclarativo, recursivo, interactivo (entre el usuario y el módulo de control que actúa sobre la base lógica para establecer la conclusión), simbólico más que numérico y declarativo de asertos y reglas.

I.6 Paradigma Heurístico

Una de las tareas más interesante y difícil de la programación es resolver problemas del tipo: ¿ Cuál es el camino más corto...?, Listar todos los casos posibles..., ¿Existe una disposición que satisfaga...? Las características de dichos problemas implican potencialmente una **búsqueda exhaustiva** de todas las posibles combinaciones de algún conjunto finito.

Se puede definir la Programación Heurística como aquel tipo de programación computacional que **aplica reglas de buena lógica**, denominadas Heurísticas, a la resolución de problemas, las que proporcionan de entre varios cursos de acción el que presenta visos de ser el más prometedor, pero no garantiza necesariamente el curso de acción más efectivo.

Este tipo de programación es de mayor aplicabilidad en el campo de la Inteligencia Artificial, especialmente en el Ingeniería del Conocimiento, dado que el ser humano opera la mayor parte de las veces usando heurísticas.

El Paradigma Heurístico define una modelo de resolución de problemas en el que se incorpora alguna componente heurística en base a:

- Una representación más apropiada de la estructura del problema para resolución con técnicas heurísticas.

- El uso de métodos de resolución de problemas aplicando funciones de evaluación con procedimientos específicos de búsqueda heurística para la consecución de metas.

La programación Heurística se usa y presenta desde diferentes puntos de vista:

a) Como técnicas de búsqueda para la obtención de metas en problemas no algorítmicos o con algoritmos que generan explosión combinatoria (ej: ajedrez, damas, etc.)

b) Como un método aproximado de resolución de problemas usando funciones de evaluación de tipo heurístico (ej: algoritmos A*, AO*)

c) Como método de poda para estrategias de programas que juegan, aunque estos métodos no son realmente heurísticos. (ej. poda alfa-beta).

Se aconseja usar un modelo heurístico cuando:

- Los **datos, limitados e inexactos**, usados para estimar los parámetros del modelo pueden contener errores inherentes muy superiores a los proporcionados por la solución de una buena heurística.

- Se usa un **modelo simplificado**, que ya es una representación imprecisa de un problema real, por lo que la solución óptima es puramente académica.

- **No se dispone de un modelo exacto que sea fiable** para ser aplicado en el modelo del problema; o si existe, es intratable computacionalmente.

- Se desea **mejorar la eficacia de un algoritmo optimizador** aplicado al modelo; por ejemplo, proporcionando buenas soluciones de inicio, guiando la búsqueda o reduciendo el número de soluciones candidatas.

- Se tiene la **necesidad de resolver el mismo problema frecuentemente**, o sobre una base de tiempo real, y el tratamiento heurístico significa un ahorro computacional.

En términos generales, se aconseja el uso de un modelo heurístico si puede proporcionar resultados superiores a los del modelo actual.

Las especificaciones más relevantes del tratamiento heurístico deben tener en cuenta las **características de**: a) la heurística, b) la información y c) las especificaciones del problema de acuerdo a las siguientes **condiciones**:

i) Una buena heurística debe ser simple, con requisitos razonables de memoria, velocidad, la velocidad de búsqueda no debe producir incrementos polinomiales, ni exponenciales, precisa, robusta, que proporciones soluciones múltiples, y que disponga de un buen criterio de parada que incorpore el conocimiento obtenido durante la búsqueda.

ii) La información a tratar es básicamente simbólica, inexacta o limitada, "incremental" y basada en el conocimiento.

iii) Las especificaciones del problema pueden ser: de optimización o de satisfacción; que produzcan una o múltiples soluciones; con tratamiento en tiempo real o no, con decisión interactiva o no, etc.

No se ha producido un lenguaje específico de programación heurística, ya que se puede implementar en cualquier lenguaje que tenga las siguientes características:

. **Lenguaje conversacional** que permita una interacción directa con el programador para la definición e implementación del problema.
. **Tratamiento de estructuras "incrementales**" que implemente programas que vayan ampliando el cuerpo de conocimiento que, en base a la experiencia, configure y refine el modelo heurístico.
. **Tratamiento simbólico** ya que casi todos los problemas a tratar tienen estructura simbólica.
. **Unidades funcionales autónomas**, que permitan modelar una heurística y su mecanismo de ejecución, definiendo módulos independientes.
. **Estructuras de datos** que permitan describir estados del problemas y relaciones entre estados.
. **Estructuras procedimentales de control y proceso** que permitan la ejecución coherente del modelo heurístico, y posibiliten la adquisición y utilización del conocimiento adquirido en el proceso de resolución del problema.

I.7 Paradigma Orientado a Objetos

La Orientación a Objetos se ha transformado en los años 90 en una metodología de diseño y desarrollo de software de gran trascendencia para la producción de software

eficiente y barato. Es una metodología de análisis, diseño y programación que configura las fases fundamentales del ciclo de vida de un sistema informático y se presenta, desde el punto de vista de la educación, como paradigmática en el desarrollo de aplicaciones.

La Orientación a Objetos se puede definir como "**Una disciplina de ingeniería de desarrollo y modelado de software que permite construir más fácilmente sistemas complejos a partir de componentes individuales**". Permite una representación más directa del modelo del mundo real, reduciendo fuertemente la transformación radical normal desde los requisitos del sistema, definidos en términos del usuario, a las especificaciones del sistema, definidas en términos de la computadora.

La tendencia actual de la Ingeniería Informática es la de producir componentes reutilizables para ensamblarlos a otros y obtener así el producto completo. Estas componentes reutilizables reciben el nombre de **"Componentes Integrados de Software (CIS)"**. Así pues, el paradigma orientado a objetos (PaOO) es una filosofía de desarrollo y empaquetamiento de software que permite crear unidades funcionales extensibles y genéricas, de forma que el usuario las pueda aplicar según sus necesidades y de acuerdo con las especificaciones del sistema a desarrollar.

La Orientación a Objetos proporciona **mejores herramientas** para:

. Modelar el mundo real de un modo más cercano a la perspectiva del usuario.
. Interactuar fácilmente con el entorno computacional usando metáforas familiares.
. Construir componentes reutilizables de software y librerías específicas de estos componentes fácilmente extensibles.
. Modificar y ampliar con facilidad la implementación de estos componentes sin afectar el resto de su estructura.

Algunos autores señalan solamente tres elementos fundamentales que configuran el Paradigma Orientado a Objetos:

Los Tipos Abstractos de Datos
La Herencia
La Identidad de los Objetos

mientras que otros indican siete aspectos básicos a considerar para una verdadera orientación a objetos:

. **Estructura modular basada en objetos**, dado que los sistemas en esta metodología son modularizados sobre la base de sus estructuras de datos.

. **Abstracción de datos**, porque los objetos son descriptos como implementaciones de tipos de datos abstractos.

. **Gestión automática de memoria**, de forma tal que los objetos no usados sean desasignados por el propio sistema sin intervención del programador.

. **Clases**, en las que cada tipo no simple sea un módulo, y cada módulo de alto nivel sea un tipo.

. **Herencia** que permita que una clase sea definida como una extensión o restricción de otra.

.**Polimorfismo y enlace dinámico**, de forma que las entidades del programa puedan referenciar en tiempo de ejecución a objetos de diferentes clases.

. **Herencia múltiple y repetida** para que se pueda declarar una clase como heredera de varias e incluso de ella misma.

De acuerdo a la etapa del ciclo de vida que se contemple, el Paradigma Orientado a Objetos presenta diferente terminología.

Análisis OO (AOO) y Diseño OO (DOO)	**Programación OO (POO)**
. Identificación del Objeto	. Tipo de dato Abstracto
. Identificación de las Estructuras y materias	. Objeto
. Definición de Elementos	. Mensaje
. Definición de Atributos	. Clase
. Conexión de instancias	. Sobrecarga
. Definición de Servicios y Conexión de Mensajes	. Enlace Dinámico
	. Polimorfismo
	. Herencia
	. Variables de clase o de instancia
	. Métodos
	. Constructores
	. Destructores
	. Genericidad
	. Aserción
	. Invariante
	. etc.

El paradigma Orientado a objetos, a menudo se describe usando el concepto Objeto/mensaje, en el que cada **objeto (elemento autónomo de información creado en tiempo de ejecución)** es solicitado para realizar un determinado servicio mediante el envío a ese objeto del mensaje adecuado. El solicitante no necesita conocer cómo el objeto proporciona el servicio pedido; la implementación es interna al objeto y la gestiona el suministrador del objeto. El énfasis se pone en qué se puede obtener más que en cómo se lo obtiene.

Este paradigma se define con la ecuación:

Objetos + Mensajes = Programa

donde el objeto es una instancia de una clase, la que implementa un tipo abstracto de dato (TAD). Y el mensaje es la información específica que se envía al objeto para que ejecute una determinada tarea.

Un TAD define conjuntos encapsulados de objetos similares, con una colección asociada de operaciones; y especifica la estructura y el comportamiento de los objetos. Las especificaciones estructuradas del TAD describen las características de los objetos pertenecientes a ese TAD, y las especificaciones de comportamiento describen qué mensajes son aplicables a cada objeto.

Un objeto de un TAD consta de dos componentes:

. **Variables de clase y de instancia** (atributos, entidades,...) que definen el estado interno del objeto.

. **Métodos** (rutinas, miembros, función,...) que definen el comportamiento de las instancias y son invocados mediante el envío de mensajes a los objetos. Un mensaje implica un objeto destino, el selector (nombre del método) y los argumentos del operador.

Algunos lenguajes orientados a objetos como el C++, Object LISP, etc. son extensiones de lenguajes convencionales que fueron diseñados básicamente como imperativos, por lo que estas extensiones no consideran a los objetos como entidades activas que reciben mensajes; más bien invocan procedimientos y funciones que pasan objetos; no obstante incorporan las características más relevantes del PaOO tales como el encapsulamiento de datos, el polimorfismo con enlace dinámico, la generalidad y la herencia.

El lenguaje paradigmático de este tipo de programación es el SMALLTALK, diseñando a principios de los años 70 en el centro de investigación de Xerox, Palo Alto, U.S.A. para disponer de un potente sistema informático donde el usuario pudiera almacenar y manipular la información ágilmente.

Dada la relevancia que está adquiriendo este tipo de programación es usual que todos los lenguajes de programación de propósito general, C, PAscal y Lisp, incorporen los conceptos que definen el PaOO quedando SMALLTALK como lenguaje puramente académico.

En resumen, se señala que el **entorno de programación orientada a objetos** es un entorno:

a) **Procedimental**, que incorpora el concepto de encapsulamiento de datos.
b) **Extensible**, ya que permite ampliar los componentes software sin necesidad de Reprogramar todo el componente.
c) **Reutilizable**, porque permite crear un programa en base a una serie de componentes definidos en bibliotecas de componentes.
d) **Interactivo**, a través de una interfaz gráfica de usuario que proporciona la funcionalidad necesaria en materia de edición, puesta a punto de programas y archivos de información.

II. Programación Orientada a Objetos - Un enfoque intuitivo

> La tecnología orientada a objetos será la más importante de las tecnologías que surjan en los años 90. Bill Gates.

II.1 La revolución industrial del software

Una de las preocupaciones más urgentes de la industria de la computación es la de crear software y sistemas corporativos más pronto y a más bajo costo. Para poder usar computadoras, que cada vez tienen mayor complejidad necesitamos software de mayor complejidad y de mayor confiabilidad.

Las técnicas orientadas a objetos permiten que el software se construya a partir de **objetos** de comportamiento específico. Estos, a su vez , se pueden construir a partir de otros objetos que también pueden estar formados por otros objetos.

El término **Revolución Industrial del software** se usa para describir el paso hacia una era en la que el software será compilado a partir de objetos reutilizables, es decir, un ensamblado a partir de componentes y paquetes de muchos proveedores.

II.2 Definiciones básicas

Algunas de las ideas fundamentales que existen en la tecnología orientada a objetos son las siguientes:

- **objetos y clases**
- **métodos**
- **solicitudes**
- **herencia**
- **encapsulado**

Aunque estos conceptos son la base del software OO, son similares a los que forman la base de todos los seres vivientes.

II.2.1 Objetos

Las personas nos formamos conceptos desde temprana edad. Cada concepto es una idea o comprensión particular de nuestro mundo. Los conceptos adquiridos nos permiten sentir y razonar acerca de las cosas en el mundo. A estas cosas a las que se aplican nuestros conceptos llamamos **objetos**. Un objeto puede ser real o abstracto. (Por ejemplo: una factura, una organización, una pantalla con la que interactúa el usuario, un plano de ingeniería, un avión, el piloto de un avión, etc.)

Desde el punto de vista del análisis y diseño orientado a objetos, nos interesa el **comportamiento** del objeto, ya que si construimos software OO, los módulos se basan en los tipos de objetos. El software que implementa el objeto contiene estructuras de datos y operaciones que expresan dicho comportamiento.

Entonces definimos, inicialmente

Un objeto es una cosa, real o abstracta, acerca de la cual almacenamos datos y los métodos que controlan dichos datos.

La estructura compuesta que pueden tener los objetos, permite definir objetos más complejos.

II.2.2 Tipos de Objetos

Un **tipo de objetos** es una categoría de objetos. Desde ese punto de vista, un **objeto** es una instancia de un tipo de objetos.

En el mundo OO, la estructura de datos y los métodos de cada tipo de objetos de manejan juntos, como una sola unidad. No se puede tener acceso o control de la estructura de datos excepto mediante métodos que forman parte del tipo de objetos.

Por ejemplo, para el tipo de objetos **empleado** pueden existir muchas instancias tales como:

Juan Pérez
María Sánchez, etc.

Para el tipo de objetos **factura**, una instancia podría ser la factura #51783.

II.2.3 Métodos

Los **métodos** especifican la forma en que controlan los datos de un objeto. Sólo hacen referencia a las estructuras de datos de ese tipo de objeto ya que no pueden tener acceso directo a las estructuras de datos de otros objetos. Para poder usar estructuras de datos de otros objetos, el objeto actual debe enviarles un mensaje.

Un objeto es una cosa cuyas propiedades están representadas por tipos de datos y su comportamiento por métodos.

Por ejemplo, un método asociado al tipo de objetos Factura podría ser aquél que calculo el importe total de la factura. Otro podría ser el de remitir la factura a un cliente, etc.

II.2.4 Encapsulado

El empaquetamiento conjunto de datos y métodos recibe el nombre de **encapsulado**. El objeto esconde sus datos a los demás objetos y permite el acceso a ellos mediante método propios. Esta propiedad recibe el nombre de **ocultamiento de información** y obviamente es un producto resultante del encapsulado. Así se evita la corrupción de los datos y se los protege de un uso arbitrario o no pretendido.

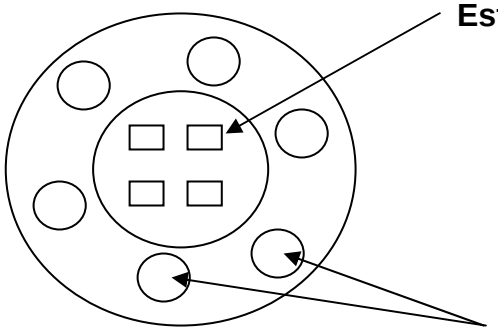
Los usuarios conocen las operaciones que pueden solicitar al objeto pero no los detalles de cómo se va a llevar a cabo esa operación. Todos los detalles de los datos del objeto y de la implementación de sus operaciones están fuera del alcance del

usuario. El encapsulado oculta los detalles de su implementación interna a los usuarios de un objeto.

El encapsulado resulta en el ocultamiento de los detalles de implementación de un objeto respecto de su usuario.

La importancia del encapsulado reside en que separa el comportamiento del objeto de su implementación, lo que permite que se modifique la implementación del objeto sin tener que modificar las aplicaciones que lo utilizan.

La siguiente figura ilustra un objeto. La estructura de datos en el centro sólo puede ser usada por los métodos del anillo exterior.



Cada objeto encapsula una estructura de datos y métodos. La estructura de datos se ubica en la parte central del objeto. El objeto es controlado por métodos que implementan las operaciones permitidas. La estructura de datos sólo puede ser utilizada por dichos métodos. A esta restricción se llama encapsulado. El encapsulado evita la corrupción de los datos.

Una videocassetera es un ejemplo de objeto. Sony es un tipo de objetos. Usted no necesita conocer sus componentes internos para usarla y no tiene acceso a la electrónica de sus datos. Solamente puede usar los métodos predeterminados. El encapsulado evita la interferencia con los aspectos internos y oculta la complejidad de sus componentes. Usted se preocupa por su funcionamiento de acuerdo a la manera descripta en el manual de uso. Los métodos grabar, cargar, activar el timer, etc. Permiten usar el objeto.

II.2.5 Mensajes

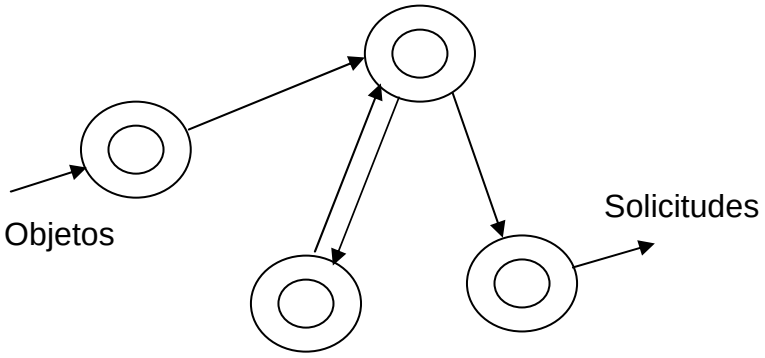
Para que el objeto realice una tarea debemos solicitárselo. Esto hace que se produzca una operación. La operación ejecuta el método apropiado y, de manera opcional, produce una respuesta.

El mensaje que constituye la solicitud contiene: el nombre del objeto, el nombre de la operación solicitada y, en caso de ser necesario, un conjunto de parámetros.

Una solicitud invoca una operación específica, con uno o más objetos como parámetros.

Un objeto puede ser muy complejo ya que puede contener sub-objetos, y estos a su vez otros sub-objetos. La persona que utilice el objeto no tiene que conocer su complejidad interna sino solamente la forma de comunicarse con él y la forma en que responde.

Por ejemplo, usted se comunica con su videocassetera usando un control remoto, al que el aparato responde con alguna acción y presenta su respuesta en pantalla.



Los objetos se comunican mediante solicitudes. Una solicitud es un mensaje que especifica la realización de una operación con uno o más objetos que, de manera opcional, devuelve un resultado.

II.2.6 Clases

Se llama clase a la implementación en software de un tipo de objetos.

> **Una clase es una implementación de un tipo de objetos. Especifica una estructura de datos y los métodos operativos permitidos que se aplican a cada uno de los objetos.**

La implementación de clases especifica la estructura de datos para cada uno de sus objetos. Además, cada clase define un conjunto de operaciones que permiten el acceso y modificación de los datos del objeto. Los detalles del método operativo se especifican en la clase.

Con esta definición se puede concluir:

- **Una clase**, que implementa un tipo de objetos, es una descripción general de muchos objetos, todos del mismo tipo. Es decir , una clase describe la estructura de todos los objetos que la componen.
- Un **objeto** es una instancia particular de una clase.
- La **estructura de datos** que describe la clase es la estructura de datos común a todos los objetos que la componen.
- Cada objeto particular que se cree hará una **copia local** de esa estructura de datos y le asignará su valor particular a las variables que integran la estructura de datos. Esos valores particulares determinarán el **estado** del objeto.
- Los **métodos** que almacena la clase son las operaciones que son comunes a todos los objetos de la clase. Como no son particulares de cada objeto, ellos no hacen una copia local de los métodos sino que existe una **única copia** de la implementación del método en la clase y todos los objetos de la misma la comparten.

Se llama Biblioteca de Clases al depósito de clases existentes, disponibles para que un analista o diseñador pueda utilizar.

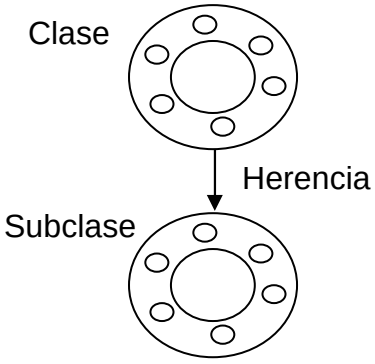
II.2.7 Herencia

Un tipo de objetos puede tener sub-tipos. Es decir, un tipo de objetos de alto nivel puede **especializarse** en tipos de objetos de niveles más bajos.

Una clase implementa un tipo de objetos. Una **subclase hereda** propiedades de su clase padre o predecesor; una sub-subclase hereda propiedades de la subclase predecesora, etc.

Una subclase puede heredar la estructura de datos y los métodos, o algunos de ellos, de su superclase. También puede agregar sus propios métodos e incluso tipos de datos propios.

Puede ocurrir que una clase herede propiedades de más de una superclase. Esto recibe el nombre de **herencia múltiple**.



Una clase puede tener sus propios métodos y su propia estructura de datos, además de los que herede de su superclase.

II.2.8 Percepción y realidad

Con el software jugamos con la realidad. Podemos hacer que algo muestre cierto comportamiento cuando en realidad su comportamiento sea muy distinto. Podemos crear una interfaz sencilla para algo muy complejo y ocultar así su complejidad.

El término **transparente** se usa para indicar algo que parece no existir pero si existe.

En cambio, el término **virtual** se usa para indicar algo que parece existir pero que en realidad no existe.

II.2.9 Herramientas de trabajo

CASE (Ingeniería de Software asistida por computadoras)

Son herramientas que usan representaciones gráficas en la pantalla par automatizar el planeamiento, análisis, diseño y la generación del software.

I-CASE (CASE integrado)

Se refiere a un conjunto de herramientas CASE que soportan todas las etapas del ciclo de vida del software, incluyendo la generación de códigos, en un único depósito lógicamente consistente.

Programación Visual

Permite a los diseñadores de software introducir, comprender, reflexionar, hacer pruebas y controlar programas en la máquina, mediante notaciones gráficas.

Generadores de Código

Lo generadores de código producen códigos sin errores de sintaxis a partir de diseños, tablas o especificaciones de alto nivel.

Depósito

Depósito es un mecanismo para la definición, el almacenamiento y la administración de la información relativa a una empresa, a sus datos y sistemas.
El coordinador del depósito aplica métodos a los datos del depósito para garantizar que los datos y sus representaciones CASE tengan consistencia e integridad.

Metodologías basadas en depósitos

Está diseñada para aprovechar al máximo el conjunto de herramientas I-CASE y maximizar la reutilización.

Ingeniería de Información

Aplica las técnicas de modelado y diseño integral a la empresa como un todo en vez de solamente a un proyecto.

Bases de Datos Orientadas a Objetos

Está diseñada para almacenar datos y métodos objeto mediante técnicas adecuadas para el procesamiento orientado a objetos.

Lenguajes no por procedimiento

Definen lo que se desea hacer y no la forma como se programa.

Motor de Inferencias

Es un software que hace deducciones a partir de hechos y reglas mediante las técnicas de inferencia lógica.

Tecnología Cliente-Despachador

El cliente es un módulo de software que solicita una operación.
El despachador es un módulo de software que responde a dicha solicitud.

Biblioteca de clases

Contiene la implementación reutilizable de tipos de objetos. Intenta lograr el máximo grado de reutilización en el desarrollo de software. El software para las bibliotecas de clase permite a los diseñadores encontrar, adaptar y utilizar las clases que se necesiten.

Análisis y Diseño Orientado a Objetos

Modela el mundo en términos de objetos que tienen propiedades y comportamiento, y eventos que activan operaciones que modifican el estado de los objetos. Los objetos interactúan de manera formal con otros objetos.

II.3 Por qué Orientado a objetos?

El estado actual de la mayor parte de la Ingeniería de Software está algo atrasado con respecto a las demás áreas de la Ingeniería porque la mayoría de los productos de software llevan asociada una **renuncia a la garantía**, es decir, se espera que funcionen de manera correcta pero su funcionamiento no se garantiza.

El mundo de las técnicas orientadas a objetos, con herramientas CASE permiten que el diseñador piense en términos de objetos y su comportamiento, y se genera el código. Esto permite construir programas sin pensar en lazos, ramificaciones o estructuras de control de programa. El constructor del programa **aprende a pensar de otra forma**.

Los eventos producen **cambios en el estado** de los objetos. La mayoría de estos cambios de estados requiere pequeñas partes de código menos propensas a errores. Se construyen tipos de objetos a partir de tipos de objetos más sencillos. Una vez que los tipos de objetos funcionan bien, el diseñador los considera como cajas negras, de modo que nadie pueda ver su interior.

En el mundo orientado a objetos, las estructuras de datos se relacionan con los objetos y sólo pueden ser utilizadas mediante métodos diseñados para ese tipo de objetos.

De hecho, **cada objeto lleva a cabo una función específica e independiente** de los demás objeto. Responde a mensajes, sin conocer la razón de envío de éstos ni las consecuencias de su acción. Ya que los objetos actúan en forma individual, cada clase se puede modificar, en gran medida, de manera independiente de las demás clases. Esto facilita la prueba y modificación de clases. El mantenimiento de sistemas orientados a objetos es mucho más sencillo que el mantenimiento de sistemas convencionales.

El mundo orientado a objetos tiene mayor disciplina que el de las técnicas de estructura convencional. Esto nos lleva a un mundo de **clases reutilizables** donde la mayor parte del proceso de construcción de software consistirá en ensamblajes de clases ya existentes y probadas.

Las técnicas OO ligadas a herramientas CASE con un generador de códigos y depósitos (también orientados a objetos) constituyen el mejor camino conocido para construir una verdadera Ingeniería de Software.

II.3.1 Características de las técnicas OO

1. Cambian nuestra forma de pensar sobre los sistemas. Es una forma más natural de pensar que el análisis estructurado.

2. Los objetos pueden construirse a partir de objetos ya existentes. Lleva a un alto grado de reutilización, ahorro de dinero y tiempo de desarrollo y a una mayor confiabilidad del sistema.

3. La complejidad de los objetos que podemos utilizar sigue en aumento puesto que los objetos se construyen a partir de otros, que a su vez, están construidos por objetos, etc.

4. El depósito CASE debe contener una creciente biblioteca de tipos de objetos, ya sea comprados o construidos en casa., Es probable que estos objetos sean más poderosos conforme crezca su complejidad. La mayoría de estos tipos de objetos serán diseñados de forma que se adapten a las necesidades de cada caso.

5. La creación de sistemas con un funcionamiento correcto es más fácil con las técnicas OO ya que las clases OO está diseñadas para reutilizarse, están auto-contenidas y divididas en métodos. Cada método se puede construir, depurar y modificar con cierta facilidad.

6. Las técnicas OO se ajustan de manera natural a la tecnología CASE. Existen ciertas herramientas elegantes y poderosas para la implementación OO. Muchas otras herramientas CASE necesitan ciertas mejoras para controlar el análisis y diseño orientado a objetos.

II.3.2 Beneficios de la tecnología OO

• **Reutilización**: Las clases están diseñadas para que se reutilicen en muchos sistemas. Para maximizar la reutilización se construyen las clases de modo que se puedan adaptar. Un depósito está poblado de una creciente colección de clases reutilizables. Las bibliotecas de clases crecen rápidamente. El objetivo fundamental de las técnicas OO es lograr la reutilización masiva al construir el software.

• **Estabilidad**: Las clases diseñadas para la reutilización repetida se vuelven estables.

• **El diseñador piensa en términos del comportamiento de objetos y no en detalles de bajo nivel**: El encapsulado oculta los detalles de implementación de las clases y hace que clase complejas sean fáciles de utilizar. Las clases son como cajas negras; quien las usa solamente debe entender su comportamiento y cómo comunicarse con ella.

• **Se construyen clases cada vez más complejas**: Se construyen clases a partir de otras clases, las cuales se integran mediante clases. El software se crea a partir de

clases ya existentes y probadas. Esto permite construir componentes complejos de software, que a su vez se convierten en bloques de construcción de software cada vez más complejos.

- **Confiabilidad**: Es probable que el software construido a partir de clases estables ya probadas tenga menos fallas que el software elaborado a partir de cero.

- **Nuevos mercados para el software**: La era de los paquetes monolíticos viene siendo reemplazada por el software que incorpora clases y paquetes encapsulados de muchos proveedores distintos.

- **Un diseño más rápido**: Las aplicaciones se crean a partir de componentes ya existentes, muchos de los cuales están construidos de modo tal que se puedan adaptar para un diseño particular y ligar entre sí en la pantalla de herramientas CASE.

- **Diseño de mayor calidad**: Ya que se integran a partir de componentes ya probados que han sido verificados y pulidos varias veces.

- **Integridad:** Las estructuras de datos sólo pueden ser utilizadas por métodos específicos. Esto es de vital importancia en los sistemas cliente-despachador y los sistemas distribuidos en los que usuarios desconocidos pueden intentar un acceso al sistema.

- **Programación más sencilla**: Los programas se conforman a partir de pequeñas piezas, cada una de las cuales se crea fácilmente. El programador crea un método para una clase a la vez. El método cambia el estado de los objetos en formas que suelen ser sencillas cuando se las considera por sí mismas.

- **Mantenimiento más sencillo:** El programador encargado del mantenimiento cambia un método de la clase por vez. Cada clase realiza sus funciones independientemente de las demás.

- **Invención**: Se pueden generar ideas con rapidez con las herramientas OO del CASE ejecutándose en una estación de trabajo.

- **Ciclo de vida dinámico**: Se facilitan los cambios a mitad del ciclo de vida. Permite a los implementadores conocer mejor a los usuarios finales, adaptarse a los cambios en las empresas, refinar los objetivos conforme el sistema se consolida y mejorar de manera constante el diseño durante la implementación.
- **Esmero en la construcción**: Lleva a mejores resultados finales. Las mejores obras creativas se refinan una y otra vez. Las herramientas OO del CASE proporcionan al constructor del software la capacidad de refinar el diseño conforme este se implementa.

- **Modelado más realista**: El análisis OO modela el área de aplicación de manera que sea lo más cercana posible a la realidad. Esto se debe a que el análisis, el diseño y la implementación utilizan el mismo paradigma y refinan de manera sucesiva.

- **Mejor comunicación entre los profesionales del sistema y los empresarios**: Los empresarios comprenden más fácilmente el paradigma OO. Piensan en términos de

eventos, objetos y políticas empresariales que describen el comportamiento de los objetos.

- **Modelos empresariales inteligentes**: Los modelos empresariales deben describir las reglas con las que los ejecutivos desean controlar su empresa. Estas se deben expresar en términos de eventos y la forma en que éstos deben modificar el estado de los objetos de la empresa. A partir del modelo de empresa se deben deducir diseños de aplicación con la mayor automatización posible.

- **Especificaciones declarativas y diseño**: Las especificaciones y el diseño construidos mediante los formalismos de las herramientas CASE deben ser declarativos (establecer explícitamente lo que se necesita). Esto permite que el diseñador piense como usuario final y no como una computadora.
- **Una interfaz en pantalla sugestiva para el usuario**: Se usa una interfaz gráfica de modo que el usuario apunte a íconos o elementos de menú desplegado, relacionados con los objetos.

- **Imágenes, video y expresión**: Se almacenan **Binary Large Objects (BLOBs)** (que son grandes cadenas de bits que además de diagramas, imágenes, música, etc., contienen métodos que permiten mostrarlos y utilizarlos) que representen imágenes, videos, expresiones, textos sin formato o algún otro flujo de bits de gran tamaño. Con el objeto se usan métodos como: compresión, descompresión, cifrado o descifrado y técnicas de presentación.

- **Independencia del diseño**: Las clases están diseñadas para ser independientes del ambiente, de plataformas, del hardware y del software. Usan solicitudes y respuestas con formatos estándares. Esto permite que sean usadas por múltiples sistemas operativos, controladores de bases de datos, controladores de redes, interfaces gráficas de usuarios, etc.

- **Interacción**: El software de varios proveedores distintos debe funcionar como conjunto ya que un proveedor usa clases de otro proveedor. Se usan los estándares del **Object Management Group**.

• **Computación cliente-despachador**: En este tipo de software, las clases cliente deben enviar solicitudes a las clases en el software despachador y recibir sus respuestas. Una clase despachador puede ser usada por diferentes clientes. Estos sólo pueden tener acceso a los datos del despachador a través de los métodos de la clase. Por lo tanto, los datos están protegidos contra su corrupción.

• **Computación de distribución masiva**: La redes de nivel mundial usan directorios de software de objetos accesibles.

• **Computación paralela**: La rapidez de las máquinas mejorará en gran medida mediante la construcción de computadoras paralelas. El procesamiento concurrente se realizará al mismo tiempo en varios chips de procesadores. Los objetos de procesadores distintos se ejecutarán, de manera simultánea actuando cada uno en forma independiente.

Otras ventajas serían: Mayor nivel de automatización de las bases de datos, mayor eficacia de las máquinas, mayor capacidad de migración, mejores herramientas CASE, bibliotecas de clases para las industrias y las empresas.

II.3.3 Medida de la complejidad de un programa

La complejidad de un programa es una medida de su comprensibilidad. Cuando más complejo sea un programa, más difícil será de comprender. La complejidad es una función de la cantidad de posibles rutas de ejecución del programa y de la dificultad de rastreo de las rutas. Si se evita la complejidad excesiva de un programa mejorará su confiabilidad y se reducirá el esfuerzo requerido para su desarrollo y mantenimiento.

II.3.4 Reutilización

Parece que en general se puede alcanzar una cifra del 80% de reutilización (sólo el 20% del código de los sistemas nuevos es nuevo; el resto proviene de clases reutilizables) con un manejo cuidadoso de la biblioteca de clases y una buena motivación por parte de los diseñadores de software en la revisión del diseño. Se requiere un esfuerzo adicional para obtener este nivel de reutilización, pero ese esfuerzo produce un desarrollo más rápido, más barato y un software de mayor calidad.

La reusabilidad es un acto administrativo. Para lograr una reusabilidad alta debe diseñarse un ambiente de desarrollo muy controlado.

II.4 El futuro del software

El reto futuro de la empresa humana será fusionar la capacidad humana y la de la máquina para lograr una mejor sinergia entre las personas y las máquinas.

La computadoras futuras no serán robots ni tendrán capacidades humanas, pero serán más interesantes en varias formas ya que usarán redes a nivel mundial de un inmenso ancho de banda y tendrán acceso a grandes cantidades de datos y enormes bibliotecas de complicado software orientado a objetos, absolutamente

preciso en su funcionamiento., Para esto se necesita software complejo, mucho más que el existente.

Además, todas las computadoras usarán discos ópticos capaces de contener cientos de millones de líneas de código. Serán en CD-ROM que se venderán las bibliotecas de aplicaciones y los objetos diseñados para su reutilización.

Nos veremos en la necesidad de crear herramientas poderosas que modificarán los métodos de construcción y deberá evolucionar la producción de software haciéndose más rápida y efectiva.

La automatización del desarrollo de software es el comienzo de una reacción en cadena.

III. La Programación Orientada a Objetos - Formalización
III.1 Introducción

Se observa que la complejidad inherente al software proviene de cuatro elementos básicos:

a) **La complejidad del dominio del problema**: Los problemas a resolver hoy en día incorporan componentes de una gran complejidad, formados por cientos de requisitos a veces inconsistentes y contradictorios.
b) **La dificultad de gestionar el desarrollo de los procesos**: Los programas constan de cientos de miles o millones de instrucciones que una sola persona no puede retener completamente, sino que se debe formar un equipo de desarrolladores, situación que presenta además el problema de salvaguardar la unidad e integridad del diseño.
c) **La flexibilidad existente en el desarrollo de software**: La industria del software necesita verificar y validar en cada desarrollo cada uno de sus componentes.
d) **Los problemas de caracterizar el comportamiento de sistemas discretos**: Los sistemas reales son continuos, cuando los modelizamos en una computadora los discretizamos, representando un estado dado por una conjunto completo de variables con sus correspondientes valores. En realidad son dos sistemas diferentes. En un sistema continuo, pequeños cambios en la entrada producen pequeñas alteraciones en la salida, mientras que en un sistema discreto los eventos externos pueden afectar cualquier parte del estado interno del sistema.

La técnica de dirigir esta complejidad consiste en la técnica "Divide y conquistarás". Cuando se diseña un sistema de software complejo es esencial descomponerlo en partes pequeñas, cada una de las cuales se puede refinar independientemente. Una descomposición inteligente reduce drásticamente la inherente complejidad del software al producir una división coherente del espacio de estado del sistema.

Uno de los modos de dividir el problema es usando una descomposición orientada a objetos que centra la atención en los objetos que configuran el problema, en sus propiedades y en el entorno e interacción con otros objetos. Se ve el problema como un conjunto de agentes autónomos (los objetos) que colaboran entre sí para ejecutar una tarea superior.

Así pues, la Programación Orientada a Objetos (POO) es:

> "**Un método de implementación en el que los programas son organizados como colecciones cooperativas de objetos, cada uno de los cuales representa una instancia de alguna clase, y cuyas clases son miembros de jerarquías de clases unidas a través de una relación de herencia**".

Existen tres partes importantes en esta definición:

a) La POO usa objetos, y no algoritmos, como elementos fundamentales para construir programas.

b) Cada objeto es una instancia de alguna clase.

c) Las clases están relacionadas entre sí a través de jerarquías de herencia.

Si un lenguaje no permite implementar estos tres aspectos, entonces no es un lenguaje OO y no puede producir verdaderos programas OO. Específicamente, la programación sin herencia no es una POO y se la denomina "Programación con tipos abstractos de datos" o "Programación basada en objetos". (Por ejemplo el lenguaje ADA).

Se señala que **un lenguaje es OO si y sólo si satisface los siguientes requisitos**:

. Soporta objetos que son abstracciones de datos que presentan una interfaz de operaciones accesibles y un estado local oculto.

. Los objetos tienen un tipo asociado que es la clase.

. Los tipo o clases pueden heredar atributos de supertipos o superclases.

En particular, SMALLTALK, C++, Object Pascal, Objective C, etc. son lenguajes OO (LOO).

Los tres **aspectos fundamentales que definen la POO** son:

a) Los tipos abstractos de datos, que son implementados por las clases.

b) La herencia como estructura jerárquica de generalización y especialización de objetos.

c) La identidad de los objetos, invariante al objeto durante toda su vida.

III.2 Tipos abstractos de datos

Sabemos ya que un tipo abstracto de datos (TAD) describe una estructura de datos en base a una lista de operaciones disponibles sobre esa estructura (las especificaciones) y a las propiedades formales de esas operaciones (los axiomas). Un TAD proporciona el mecanismo para separar la interfaz y la implementación de un tipo de datos.

Definamos por ejemplo el TAD "número natural"

En la definición de un TAD, además de las funciones o servicios que proporciona la estructura y los axiomas que implementan esos servicios se suele incorporar condiciones que se deben producir antes de ejecutar esos servicios (precondiciones) o después de que el servicio ha sido ejecutado (postcondiciones); pudiendo también el TAD definir estructuras genéricas.

Presentemos, a modo de ejemplo, la especificación formal del TAD Pila, siguiendo la terminología de Meyer:

Types
Pila[X]
Functions
es-vacía: Pila --> boolean
crear:-->Pila[X]
insertar: X * Pila[X] --> Pila[X]
eliminar: Pila[X] -/->Pila[X]
cabeza:Pila[X] -/-> X
Preconditions
pre eliminar (s : Pila[X]) = (**not** es-vacía(s))
pre cabeza (s : Pila[X]) = (**not** es-vacía(s))
Axioms
for all x: X, s: Pila[X]:
es-vacía(crear());
not es-vacía(insertar(x,s));
cabeza(insertar(x,s)) = x;
eliminar(insertar(x,s)) = s
End

Este tipo particular de estructura de datos al ser definido como Pila[X] que es una TAD genérico, con un parámetro formal X que representa un tipo cualquiera. La **genericidad** permite que una misma especificación sirva para distintos propósitos (ej: Pila[integer], Pila[Float], Pila[empleado] si empleado es un registro, etc.). En otras palabras, la especificación debe capturar las propiedades comunes mediante un tipo parametrizado.

Un TAD especifica claramente los servicios que pueden solicitarse a los objetos pertenecientes a él. A este conjunto de servicios se le llama su **interfaz o protocolo**. Por ejemplo, para el TAD Pila, la interfaz está formada por:

es-vacía (pregunta si la pila está vacía)
insertar (introduce un elemento en la cabeza de la pila)
eliminar (elimina el elemento cabeza de la pila)
cabeza (proporciona el elemento superior de la pila)

Se puede acotar que crear no es un servicio que proporcionan los objetos del TAD, sino un servicio que proporciona el TAD para crear objetos.

Por otra parte, un TAD proporciona un mecanismo adicional que permite separar claramente la interfaz y la implementación del tipo de dato.

La **implementación** de un TAD consta de:

- La representación interna de la estructura de datos; y
- Las operaciones o algoritmos que implementan los servicios del TAD.

Ambos elementos suelen ser privados al TAD, invisibles a sus usuarios o clientes.

Una **interfaz** señala el modo en que un objeto perteneciente a un TAD puede actuar, constituyendo el punto de vista externo (estático y dinámico) de la abstracción.

Todo TAD tiene propiedades estáticas y dinámicas que se reflejan en sus objetos. Por ejemplo, para el TAD Pila, sus propiedades estáticas son: el espacio que necesita sobre memoria principal o un periférico, su nombre y contenido. En cambio, el valor de cada una de estas propiedades es dinámico (variables en la vida del objeto). Así una pila puede aumentar o disminuir, y su contenido o nombre puede cambiar.

La implementación de la representación interna de la estructura y de los servicios del TAD incorpora el concepto de **encapsulamiento**. Mientras que la interfaz presenta la parte externa del objeto (sus prestaciones), el encapsulamiento o el "**ocultamiento de información**" impide que el usuario del objeto vea su interior, que es donde se implementa el comportamiento del TAD.

Para que un TAD funcione bien, la implementación de su representación y de sus servicios debe estar encapsulada, es decir,

> **Cada clase (construcción normalmente usada para definir un TAD) debe tener dos partes: una interfaz y una implementación**.

La interfaz de la clase debe presentar la parte externa, especificando los servicios que proporcionan todas las instancias de la clase (objetos). La implementación de la clase comprende la representación de la abstracción, así como todos los mecanismos que proporcionan el comportamiento deseado de cada servicio.

Esta división explícita entre interfaz e implementación representa una clara separación de conceptos: a) La interfaz de una clase es el lugar donde se especifican todas las peticiones que un usuario puede efectuar a una instancia de la clase; b) la implementación encapsula los detalles que un usuario no puede usar.

Así pues, el encapsulamiento es el proceso de ocultamiento de todos los detalles de un objeto que no contribuyen a sus características esenciales, y ofrece la posibilidad de cambiar la representación de un TAD sin afectar al comportamiento de sus clientes.

En resumen, las **ventajas del TAD** son:

- Permitir una mejor conceptualización y modelado del mundo real. Categorizan objetos con una misma estructura y comportamiento.

• Aumentar la robustez de un sistema basado en TADs. Si además el lenguaje que implementa los TADs admite comprobación de tipos de los objetos en tiempo de compilación (C++) esto proporciona mayor integridad y corrección al programa.

• Posibilitar la modificación de la implementación sin afectar la interfaz del TAD ya que separa la implementación (axiomas) de las especificaciones (interfaz).

• Permitir la **reusabilidad y extensibilidad** del sistema, posibilitando la creación de bibliotecas de componentes de TAD fáciles de mantener y de ampliar, ya que la expansión de un TAD no afecta las estructuras anteriores ya creadas.

III.3 Objetos

La POO se describe a menudo usando el paradigma objeto/mensaje, en el que cada objeto puede ser solicitado para realizar un servicio requerido por un objeto cliente, mediante el envío del mensaje adecuado.

El solicitante no necesita conocer cómo el objeto proporciona el servicio requerido, ya que esta prestación se genera internamente al objeto suministrador. El énfasis del paradigma se pone en los servicios que se quieren obtener más que en cómo se obtienen.

Desde el punto de vista de la orientación a objetos:

> **Un objeto es un elemento autónomo de información, creado en tiempo de ejecución que proporciona una serie de servicios sobre la estructura de datos que implementa**.

En otras palabras, un objeto es una instancia de un TAD que incorpora, en una estructura autónoma, su interfaz y la implementación de su representación interna y de sus servicios.

Un objeto tiene: un **estado**, un **comportamiento** y una **identidad** que lo diferencian de los demás objetos. La estructura y comportamiento del conjunto de objetos similares vienen definidos en un tipo abstracto de datos (en los lenguajes OO se implementa mediante una **clase**).

a) Estado de un objeto

El estado de un objeto viene determinado por el conjunto de propiedades o atributos que tiene el objeto (su estructura estática), junto con los valores que pueden asumir cada una de estas propiedades (su estructura dinámica).

El hecho de que cada objeto tenga un estado implica que ocupa un espacio, que existe durante un tiempo, que cambia el valor de sus atributos, que es instanciado y que puede ser creado y destruido.

b) Comportamiento

El comportamiento de un objeto viene determinado analizando como reacciona al recibir un mensaje desde otro objeto y cómo cambia su estado.

Un **mensaje** es una acción que se manda a un objeto para que realice una operación con un propósito específico.

En la práctica existen cinco (5) tipos de operaciones que suele realizar un objeto:

- Modificar — Operación que altera el estado del objeto. (Ej. operación de almacenamiento).
- Seleccionar — Operación que accede al estado del objeto, pero no lo altera. (Ej. Mostrar acumulador)
- Iterar — Operación que permite que todas las partes de un objeto sean accedidas en un orden bien definido.(Ej: Si la operación "Mostrar estado" hubiera sido definida como visualizar secuencialmente la memoria, el acumulador y el registro operando en una calculadora).
- Constructor — Operación que crea un objeto e inicializa su estado.
- Destructor — Operación que libera el estado de un objeto y (opcionalmente) destruye el propio objeto.

Ya que estos cinco tipos de operaciones son diferentes, es usual implementarlas con un estilo de programación diferente. Parece apropiado declarar las **operaciones de modificar como procedimientos**, las **operaciones de seleccionar como funciones** y las de **iterar como procedimientos genéricos**. Las operaciones **constructor y destructor** son declaradas como parte de la definición de una clase en C++.

La existencia de un estado dentro de un objeto implica que el orden en que se realicen las operaciones es importante.

c) Identidad de un objeto

Se entiende por identidad de un objeto la propiedad característica que tiene ese objeto que lo distingue de todos los demás.

Existen tres técnicas básicas para identificar un objeto que se aplican en los lenguajes de programación, bases de datos y sistemas operativos.

1) Utilizar direcciones o referencias de memoria
2) Utilizar nombres definidos por el usuario
3) Mediante claves identificadoras en el caso de colecciones.

La mayoría de los lenguajes de programación y de las bases de datos aplican las dos primeras técnicas para distinguir los objetos, mezclando con ello la dirección con la identidad.

> **En POO, cada objeto tiene una identidad, inherente al objeto, que permanentemente está asociada a él. Un objeto tiene una identidad única durante toda su vida, incluso puede cambiar su nombre, pero permanece su identidad, y su estado evoluciona**.

En los primeros lenguajes de programación, la identidad del objeto se correspondía con su nombre, siendo imposible separar ambos conceptos. Cuando se incorpora a los lenguajes de programación la asignación dinámica de memoria desaparece esta correspondencia. Así, es posible asociar a un nombre una dirección de memoria, y posteriormente, asignar a ese nombre otra dirección de memoria. Por ejemplo, en el lenguaje C, la sentencia:

int * nombre;nombre = malloc(size(**int**));

asigna a la variable **nombre** la dirección de una zona del stack del sistema con capacidad para almacenar y operar con un entero. En este caso, el objeto es una zona de 2 bytes del stack, asociado al tipo de datos "integer".

Si posteriormente se ejecuta la sentencia

nombre = malloc(size(**int**));

el sistema asigna a la variable **nombre** otra dirección de la zona del stack con las mismas características que la anterior, quedando el primer objeto colgado con su identidad, pero sin poderlo referenciar.

Aunque no se use el mecanismo de asignación dinámica de memoria, también se desvirtúa la relación nombre/identidad del objeto con el concepto de referencia, que permite que un mismo objeto tenga varios nombres (**"alias"**).

Así, en C++ se puede declarar:

```
int nombre = 20;
int & nombre1 = nombre;
```

lo que produce que nombre y nombre1 direccionen la misma zona de memoria de tipo entero (es decir, el mismo objeto)

Otro caso usual en los lenguajes de programación que muestra esta disociación entre el nombre del objeto y su identidad, se produce con el uso de subprogramas, en que las variables locales a los mismos direccionan diferentes objetos del stack del sistema en cada ejecución del subprograma.

Veamos, a modo de ejemplo, la función **intercambiar** codificada en C:

```
void intercambiar(int * ob1, int * ob2) {
int temp;
temp = *ob1;
*ob1 = *ob2;
*ob2 = temp;
}
```

presenta a la variable temp, cuya dirección es un objeto de tipo entero ubicado en el stack del sistema, que cambia de dirección en cada ejecución de "intercambiar", siendo de esta manera, distinto el objeto asociado a temp en cada ejecución.

Otra manera de identificar un objeto consiste en usar "**claves identificadoras**". Este mecanismo es muy usado en los sistemas de gestión de bases de datos y consiste en aplicar como identificación de un objeto, un conjunto de atributos del mismo que son únicos y lo diferencian de los demás. Por ejemplo, en una base de datos de personal, usar como clave identificadora, el nombre o el DNI de una persona.

Este mecanismo de identificación confunde el concepto de identidad con el de valor de ese o esos atributos, generando tres tipos de problemas:

a) Modificación de claves identificadoras:

Como la clave identificadora se corresponde con la identidad del objeto, si su contenido cambia se produce un grave problema en la base de datos. (Ej. Si un individuo cambia su nombre, en realidad, la identidad del individuo no debería cambiar).

b) No uniformidad

Para un mismo objeto, las claves identificadoras pueden ser diferentes a lo largo de la gestión de la base de datos. Por ejemplo, en un caso, la clave de identificación de una persona puede ser su DNI, y en otro, su No. de Registro de Personal, por lo que la mezcla de ambos archivos requerirá que se cambie una de las claves, originando una discontinuidad en la identidad del personal en uno de los archivos.

c) Uniones no naturales

El uso de claves identificadoras causan, en los sistemas de gestión de bases de datos, el uso de uniones entre ellos para la recuperación de información

En resumen, la identidad de un objeto es una característica fundamental del mismo en la Orientación a Objetos, y su implementación a través de direcciones, referencias, nombres del objeto y claves identificadoras, que es la forma que usan para implementarla los actuales lenguajes de programación, genera mucha controversia debido a que no son los mecanismos adecuados de identificar unívoca y permanentemente al objeto.

III.4 Clases

Una clase es un marco que permite crear objetos de la misma estructura. Por lo general, son elementos estáticos que corresponden al texto del programa y que definen Tipos Abstractos de Datos.

> **Las clases son descripciones estáticas de un conjunto de objetos que son las instancias de la clase**.

A modo de ejemplo, definamos la clase ***Persona*** con los siguientes atributos: DNI, Nombre, Apellidos, Edad, Sexo, Profesión, Salarios, Complementos, etc. Y con las siguientes operaciones:

- Persona: Crea una instancia con los atributos específicos de una persona.
- Calcular-Salario: Obtiene el salario de una persona en base a sus características personales.
- Cambiar-Complemento: Cambia el valor de una retribución complementaria.
- Listar-Datos-Persona: Lista los atributos de una persona específica.
- Importe-Salario: Proporciona el importe del atributo salario.
- etc.

Esta plantilla de definición, que configura el TAD ***Persona***, conforma el concepto de **clase**. Usualmente, el término clase va unido a la codificación del TAD en un lenguaje de programación, y en lenguajes como C++ se suele usar la palabra **class** en dicha codificación.

> **En términos generales, una clase se define como un grupo o conjunto caracterizado por uno o varios atributos comunes**.

En el contexto de la POO:

> **Una clase es un conjunto de objetos que comparten una estructura y comportamiento comunes**.

Así como un TAD consta de atributos y servicios, las clases implementan estos conceptos a través de las ***variables*** (de instancia o de clase) y de los ***métodos***.

a) Variables

Denominadas también "entidades" (Eiffel), "miembros datos" (C++) o simplemente "atributos", constituyen las representación interna de la clase y mantienen el estado del objeto.

En C++, las variables de instancia o de clase incorporan a su nombre, el tipo de dicha variable, efectuándose en tiempo de compilación la comprobación de ese tipo.

Ejemplo: Implementemos en C++ las variables de instancia de la clase ***Persona***:

```
long DNI;
char * Nombre, * Apellidos;
int Edad;
char Sexo;
char * Profesion;
long salario;
long complementos [10];
```

b) Métodos

Denominados también "rutinas" (Eiffel), "funciones miembro" (C++), etc; definen el comportamiento de las instancias al implementar los servicios especificados en su TAD.

Los métodos se pueden **clasificar** en tres tipos:

. **De acceso**: Acceden a valores de las variables de instancia recuperando su valor. Se implementan mediante funciones.

. **De Actualización**: Son métodos que cambian el estado o valor de una variable de instancia. Se implementan mediante procedimientos.

. **Constructores y Destructores**: Son invocados para crear, inicializar, copiar y destruir un objeto. Su implementación es específica de cada lenguaje.

Por ejemplo, la codificación de la clase "***Persona***", incorporando sus métodos sería:

```
class Persona {
// variables.
long DNI;
char * Nombre, * Apellidos;
int Edad;
char Sexo;
char * Profesion;
long salario;
long complementos [10];
public ;
```

```
// métodos.
Persona () {... ...} //constructor.
void Calcular-Salario (int mes);
void Cambiar-Complemento (long importe, int numero);//asigna al
// complemento "numero" el valor "importe".
void Listar_Datos_Personal();
long Importe_Salario();
... ...
- Persona() {... ...}//destructor
    };
void Persona :: Calcular_Salario() {... ...}
... ...
long Persona :: Importe_salario() {... ...}
... ...
```

Los métodos son invocados enviando mensajes a los objetos, cuya ejecución escenifica el conjunto de operaciones a realizar para la resolución del problema.

En general, un **mensaje consta de**:

- El **nombre del objeto destino** receptor del mensaje
- El **selector o nombre del método** que ese objeto debe ejecutar
- Los **argumentos** del método (si es que los lleva)

En el ejemplo en C++, tendríamos el siguiente conjunto de mensajes:

```
// Se instancia un objeto p de la clase "Persona".
Persona p;
// Calcular el salario del mes de Julio.
p.Calcular_Salario(7); // mensaje.
                        // p : objeto receptor del mensaje.
                        // Calcular_Salario : El selector.
                        // 7 : argumento del método.
```

El receptor del mensaje elige el método apropiado de todos los asociados con ese objeto, en base al nombre del método y a sus argumentos.

Mientras que un objeto es una entidad individual que ejecuta un papel en el sistema global, la clase define la estructura y comportamiento comunes a todos los objetos relativos a ella, sirviendo pues, como enlace contractual entre una abstracción y todos sus usuarios.

Este punto de vista de enlace nos permite diferenciar entre el exterior y el interior de la clase. La **interfaz** de una clase, que comprende la estructura de los objetos y el conjunto de declaraciones de los métodos aplicables a las instancias de la clase, proporciona el punto de vista del usuario de la clase; es decir, qué operaciones se pueden realizar sobre los objetos de la clase. **Enfatiza**, por lo tanto, **la abstracción**, mientras oculta al usuario su comportamiento.

La **implementación de la clase** es la visión interna del diseñador de la clase, y comprende la implementación de todos los métodos definidos en la interfaz de la clase.

En el ejemplo de la clase "Persona", la interfaz está formada por las siguientes declaraciones:

```
// Declaraciones de las variables de instancia o clase.
long DNI;
char * Nombre, * Apellidos;
int Edad;
char Sexo;
char * Profesion;
long salario;
long complementos [10];
// Declaraciones de los métodos.
Persona () {... ...} //constructor.
void Calcular-Salario (int mes);
void Cambiar-Complemento (long importe, int numero);
void Listar_Datos_Personal();
long Importe_Salario();
... ...
- Persona() {... ...}
```

Y la implementación la constituye la implementación de los métodos.
La interfaz de una clase, según su visibilidad, se puede dividir en tras partes:

a) **Parte pública**: Declaraciones visibles a todos los usuarios de esa clase.
b) **Parte Protegida**: Declaraciones visibles a la propia clase a sus subclases.
c) **Parte Privada**: Declaraciones visibles únicamente a la propia clase.

Por otra parte, el **estado de un objeto** se suele representar mediante un conjunto de declaraciones de constantes y variables situadas en la parte privada de la interfaz. De esta manera, la representación del objeto está **encapsulada** y sus cambios afectan funcionalmente al usuario. Esta representación se hace así por razones prácticas, con el fin de que el compilador conozca la cantidad de memoria que debe asignar al objeto.

III.5 Polimorfismo y Sobrecarga

De acuerdo al diccionario de la Real Academia, la palabra ***polimorfismo*** representa "**la calidad o estado de un elemento de asumir varias formas**"; en otras palabras, es la capacidad de una entidad de tomar varias formas.

En programación, el polimorfismo indica que el mismo componente del lenguajes puede asumir diferentes tipos o manipular diferentes tipos de objetos, siendo la **sobrecarga** una forma de polimorfismo.

Estos dos conceptos de la POO son muy poderosos y relevantes, por los que objetos de distinto tipo pueden ejecutar operaciones con el mismo nombre y distinta semántica. Ambos conceptos son muy similares, lo que se presta a confusiones.

La sobrecarga es una cualidad polimórfica de una elemento (operador, función, etc.) de poderse aplicar a diferentes tipos de objetos. Por ejemplo, el operador "-" para restar enteros, reales, etc.

En los lenguajes de programación modernos este concepto se denomina sobrecarga. C++ permite además declarar procedimientos y funciones con el mismo nombre pero variando el número o el tipo de los argumentos o de su valor resultado.

Así, en C++ son válidas las siguientes declaraciones de función sobrecargada:

```
void imprimir(int);
void imprimir(char);
void imprimir(int,float);
```

cuya utilización sería la siguiente:

```
main () {
int i;
float f;
char c;
... ...
imprimir (i);
imprimir (c);
imprimir (i,f);
     }
```

También se puede definir el **Polimorfismo Paramétrico**, por el cual un elemento (una función, operador, etc.) se define como una planilla que implementa la misma ejecución pero con diferentes tipos de datos.

Así, la función imprimir codificada en C++ con estructura polimorfa paramétrica para que permita tratar datos del tipo int, float, char, double, etc. sería:

```
template <class T>
void imprimir (T x) {... ...}
```

y podría ser referenciada como:

```
imprimir(20); // si se trata de un entero
imprimir(20.0f); // si se trata de un real
imprimir(20.0); // si se trata de un real (double)
```

Sin polimorfismo paramétrico, el diseñador necesita usar sentencias condicionales del tipo CASE o SWITCH para seleccionar la opción apropiada, y desarrollar el código de programación de cada una de esas opciones.

El concepto de polimorfismo se incorpora en POO con el nombre de ***genericidad*** que "**la capacidad que tiene un lenguaje de formar funciones o clases paramétricas** (arrays, listas, árboles, etc.) **que describen estructuras de datos genéricas**".

En el ejemplo anterior en C++, "**void** imprimir (T x)" es una función paramétrica y x es un parámetro genérico formal.

La genericidad se aplica satisfactoriamente sólo en lenguajes tipados, donde es posible controlar que una operación se use con su tipo correcto.

La ventaje más importante de la genericidad, y por lo tanto del polimorfismo paramétrico, es la de compartir código, ya que si el lenguaje no soporta esta propiedad hay que definir código para cada una de las implementaciones.

III.6 Enlace Dinámico

En los lenguajes convencionales de programación los operadores, las funciones, etc. se ligan a sus respectivos operandos en tiempo de compilación (encadenamiento estático). En tales casos, cada operador tiene un único nombre y una única operación.

Por ejemplo, en Pascal:

```
var
  i,j,k : integer;
  ... ...
begin
  ... ...
  k := i + j;
  ... ...
  imprimir(k);
  ... ...
end.
```

Los lenguajes orientados a objetos permiten que el enlace de un operador o función a una operación específica se haga en tiempo de ejecución, **enlace dinámico** o **enlace diferido**, lo que configura el polimorfismo dinámico. Este concepto es una de las propiedades más importantes de la OO y una de sus ventajas. Significa que el sistema enlaza los selectores de mensajes a los métodos que los implementan durante la ejecución del programa.

Ejemplo: en C++

```
class polígono {
... ...
virtual int perímetro () {...}  //Perímetro puede ser redefinido en
... ...};                       subclases.
class rectángulo : public poligono //rectángulo es subclase de
... ...                                    poligono
int perímetro () {... ...} // redefinición de perímetro
... ...};
void main () {
poligono * pp;          // pp es un puntero a un objeto polígono.
rectangulo r;           // r es un objeto rectángulo.
pp = & r ;              // pp apunta a un rectangulo.
pp --> perimetro (); // llama a perímetro de rectángulo.
... }
```

En este ejemplo, aunque parece que se invoca al método *perímetro* de la clase *polígono*, debido a que pp es un objeto puntero de esta clase, en realidad pp es en el momento de la llamada a *perímetro ()* un puntero que apunta a un objeto de la clase

rectángulo, por lo que el método *perímetro()* que se ejecutará es el correspondiente a esta clase.

Si este enlace se hubiera realizado durante la compilación, el programa hubiera actuado erróneamente, pero gracias al enlace diferido en la fase de ejecución, el programa se ejecuta correctamente.

Se ve también en este ejemplo la íntima relación que existe entre el polimorfismo dinámico y el enlace diferido ya que con el polimorfismo dinámico, el enlace de un método a un nombre no se determina hasta el momento de la ejecución.

En C++, el programador puede controlar si una función miembro usará enlace diferido o estático, especificando o no el método como ***virtual*** y en ese caso la función es considerada polimórfica.

Las ventajas del enlace dinámico son:

a) Extensibilidad

El mismo operador se aplica a instancias de muchas clases sin modificar su código.

b) Desarrollo de código más compacto

La sobrecarga de operadores se puede combinar con el enlace diferido, lo que permite evitar constructores condicionales de ramificación tales como if-then-else o el switch, que comprueban el tipo de objeto e invocan al operador correspondiente.

c) Claridad

EL código generado es más legible y comprensible. Ello realza la robustez y eficiencia del esfuerzo de desarrollo.

III.7 Herencia
III.7.1 Las relaciones entre clases y herencia

Antes de definir en profundidad el concepto de herencia desde el punto de vista OO, interesa definir las clases de relaciones que existen entre objetos.

Básicamente, existen tres clases de relaciones:

a) Generalización: Esta relación especifica una "***clase de***" objeto. Por ejemplo, un boxer es una clase de perro, significando con ello que el boxer es una subclase especializada de una clase más general como es perro.

b) Agregación: Esta relación especifica una "***parte de***" otro objeto. Así, por ejemplo, una puerta es parte de una habitación.

c) Asociación: Especifica una ***conexión semántica*** entre objetos no relacionados. Por ejemplo, las rosas y las velas son objetos no relacionados, pero representan cosas que podemos usar para decorar la mesa de una cena íntima.

La herencia es la forma más poderosa que se ha desarrollado en los lenguajes de programación para implementar la generalización y la asociación.

En OO, la herencia es el mecanismo fundamental para implementar la reusabilidad y la extensibilidad del software. A través de ella, los diseñadores pueden construir nuevas clases partiendo de una jerarquía de clases ya existentes (comprobadas y verificadas) evitando con ello el rediseño, la recodificación y verificación de la parte (clases) ya implementada.

Como **aspectos fundamentales de la herencia** cabe distinguir:

a) Carácter estructural - Las subclases heredan las estructuras de sus clases base. Por ejemplo, todo lo que se pueda definir para la clase *persona* es válido para la clase *empleado*, ya que el empleado **hereda** todas las características estructurales de una persona.
Ejemplo: en C++

```
class persona {
nombre,edad,peso,altura,dirección,etc.//datos estructurales.
persona,cambio_dirección,...//métodos.
};
class empleado : public persona {
dpto,tipo_trabajo,salario,tipo_formacion,...etc.//datos est.
empleado,cambio_dpto,calculo_salario,et.//métodos.
};
```

donde:

El nombre, la edad, el peso, etc. son datos inherentes a la estructura de un objeto de la clase *empleado* al derivar, por herencia, de la clase *persona*.

b) Características de Comportamiento: Las sublcases, normalmente, también heredan los métodos de sus superclases o clases base. Así, para crear una instancia del tipo empleado, se usará también el constructor *persona* de la clase base. Y el método "*cambio_dirección*", si es público, será utilizado por las instancias de la clase *empleado*.

c) Especialización: La herencia es contemplada como una especialización desde el punto de vista del tipo. En el ejemplo, el tipo empleado está más especializado que el tipo persona. Desde este punto de vista, es más conveniente hablar de "**heredero**" y "**descendiente**" que de subclase y superclase.
La relación "***es-un***" es una relación de especialización, porque la clase derivada define un conjunto más reducido y especializado de objetos, que su clase base.

d) Extensión: Por otro lado, la relación de herencia se puede contemplar como una extensión de la clase base, si se tiene en cuenta que los objetos derivados tienen más atributos y servicios que los correspondientes de la clase base. Por ejemplo, la sublcase *empleado* puede tener todos los atributos y métodos de la clase *persona*, más los suyos propios.

e) Generalización: La clase más general en una estructura de clases es la "clase base". Ella representa la categoría de abstracción más generalizada dentro del dominio dado. Algunos lenguajes le dan un nombre específico a la clase base más alta que sirve como última superclase de todas las clases (**Object** en SMALLTALK, y **Tobject** en OBJECT PASCAL). C++ no tiene nombre específico.

III.7.2 La Herencia y el ocultamiento de información

En general, una clase tiene dos tipos de usuarios: las instancias y las subclases.

Por una parte, el diseñador puede definir (dependiendo del lenguaje) qué variables de instancia y métodos son **visibles** a las instancias de una clase. Es decir, cuál es el tipo de interfaz que tienen los objetos de esa clase.

En C++, esto se consigue con los especificadores **private**, **protected** y **public**.

En cuanto a las subclases, que heredan las estructuras de las superclases, el diseñador puede controlar en C++ qué miembros (variables y métodos) son visibles a las subclases mediante los especificadores:

- **private** Ningún miembro privado de la superclase es visible en la subclase, y los miembros protected y public de la superclase pasan como privados a la subclase.

- **protected** Los miembros privados de la superclase no son visibles en la subclase, y los miembros public y protected de la superclase pasan como protected a la subclase.

- **public** Todos los miembros de la superclase (excepto los privados) son visibles en la subclase con la visibilidad que tenían en la superclase.

Ejemplo:

```
class persona {
// variables privadas (visibilidad por defecto)
char *nombre;
int edad;
float peso, altura;
protected:
// variables protegidas
char * direccion;
... ...
public:
// variables públicas
persona (char *,int,float,char *);
cambio_direccion(char *);
... ...};
class empleado: private persona {
// variables privadas (visibilidad por defecto)
int dpto;
char *tipo_trabajo;
int salario;
char *tipo_formación;
... ...
public:
empleado (int,char *,int,char *);
cambio_dpto(int);
calculo_salario(int);
... ...};
... ...
```

Las variables de instancia: *nombre*, *edad*, *peso* y *altura* de la clase **persona** no son visibles en la clase **empleado**, y por lo tanto no se pueden utilizar desde esta clase. Mientras que la variable de instancia *dirección* y los métodos *persona*, *cambio de dirección*, etc. son visibles desde la subclase **empleado**, y por lo tanto se pueden usar en esta subclase, pero no son visibles para las instancias de la subclase **empleado**.

En C++ se admite ***herencia selectiva***, es decir que se puede limitar, con el especificador **export** la visibilidad de las variables y métodos; en otras palabras, el acceso a los usuarios de la clase.

Por ejemplo, en C++

```
class POLIGONO export
vertices, traslados, rotacion, perimetro
feature
vertices: Lista_encadenada[PUNTO];
traslado (a,b: REAL) is do ... end;
rotacion (centro: PUNTO; angulo : REAL) is do ... end;
... ...
end -- clase POLIGONO
```

class RECTANGULO **export**
traslado, rotacion, perimetro, diagonal
inherit POLIGONO **redefine** perimetro
feature
diagonal: REAL;
perimetro : REAL **is do ... end**;
... ...
end -- Rectangulo

La subclase RECTANGULO hereda todas las variables de instancia y métodos de la clase POLIGONO (vértices, traslado,...) mientras que las instancias de la subclase RECTANGULO tienen acceso a las variables y rutinas (traslado,..., diagonal) especificadas entre **export** e **inherit** (por ejemplo, no pueden acceder a *vértice*).

III.7.3 La herencia y la tipificación

Aunque en algunos lenguajes estos conceptos se confunden, la herencia trata con implementaciones mientras que la tipificación es una relación semántica entre tipos de objetos.

Diremos que un tipo T2 es un subtipo del tipo T1 si cada instancia de T2 es también una instancia de T1.

Por ejemplo, T1 número enteros, T2 números enteros pares.

Y una instancia de T2 se puede usar en operaciones definidas para instancias de T1 (Los números pares se pueden usar en sumas, productos y cocientes de enteros, pero no al revés).

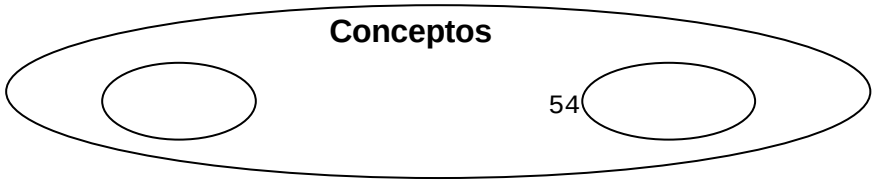
En la herencia, las subclases son contempladas como subtipos de sus clases base, y las instancias de una subclase son consideradas también como subtipos de sus clases padre.

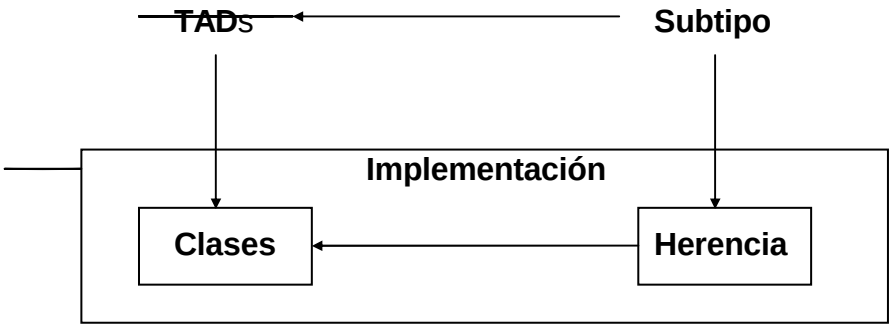
No obstante, la herencia es más restrictiva que **el subtipo**, ya que este puede ser contemplado como **una jerarquía de comportamientos**.

El SET (conjunto de elementos distintos) es un subtipo de BAG (conjunto de elementos iguales y diferentes) y las operaciones sobre SET (Union, Intersección, pertenencia, etc.) tienen su correspondiente utilización sobre BAG, pero se puede efectuar una representación diferente para SET (Ej. un array) que para BAG (ej. lista encadenada).

La herencia es una jerarquía de implementaciones.

La relación entre herencia y subtipado es similar a la existente entre TADs y clases. En los lenguajes de programación OO, las clases se usan para implementar TADs y la herencia soporta el subtipado entre clases.





C++ permite la conversión de valor de un objeto de un tipo a otro *únicamente* si existe una relación de superclase/subclase entre los dos.

IV. Clases en C++: Un ejemplo motivador

> **Una clase abarca el concepto de un tipo abstracto de datos definiendo, tanto el conjunto de valores de un cierto tipo, como el conjunto de operaciones que se ejecutan sobre tales valores.**

Un variable de un tipo de clase se conoce como un **objeto** y las operaciones sobre tal tipo se conocen como **métodos**.

Cuando un objeto A invoca un método m sobre otro objeto B se dice que **"A está enviando un mensaje m a B**". Se considera que B recibe el mensaje y realiza una transformación en respuesta al mismo.

Una clase C++ se apoya en el concepto de estructura en C:

> **Una estructura de C es un conjunto de campos nombrados mientras que una clase C++ es un conjunto de campos nombrados y métodos (o funciones) que se aplican a los objetos de tal tipo de clase**.

Además. C++ incluye el concepto de ocultar información, limitando el acceso a ciertos elementos de la clase, a los métodos de la misma clase.

Definamos, a modo de ejemplo, una clase C++ que implementa el ADT racional, que modela un número racional que consiste de dos componentes (un numerador y un denominador) y define métodos para comprobar si dos racionales son iguales, para sumar, multiplicar dos racionales y para crear un racional a partir de dos enteros.

```
class racional
{
    long numerador;
    long denominador;
    void reducir(void);
public:
    racional add(racional);
    racional mult(racional);
    racional divide(racional);
    int equal(racional);
    void print(void);
    void setracional(long,long);
};
```

Miembros privados por default. Sólo se puede accederlos por medio de los métodos de la clase.

Datos miembros

Función miembro para reducir la representación interna del racional a términos mínimos.

Miembros públicos

Funciones miembros que aquí solamente se definen. Sus implementaciones se proporcionan después. Forman la interfaz pública de la clase racional.

Observe que los métodos mencionan un único parámetro a pesar de ser métodos binarios (para dos operandos). Esto se debe a que el objeto para el cual se invoca cada método es un **parámetro implícito** para cada uno de ellos.

IV.1 Implementación de los métodos

Los métodos de una clase se pueden implementar dentro de la declaración de la clase o fuera de ella.

Definamos, por ejemplo, el método *setracional*, que asigna a un objeto de clase racional un valor particular, dentro de la clase.

```
class racional
{
        long numerador;
        long denominador;
        void reducir(void);
public:
        racional add(racional);
        racional mult(racional);
        racional divide(racional);
        int equal(racional);
        void print(void);
        void setracional(long n, long d)
        {
                if (d == 0)
                        error("Error: denominador es cero");
                numerador = n;
                denominador = d;
                reducir();              // reduce a términos mínimos
                }                       // fin de setracional
        };                              // fin de la clase racional
```

Si, por el contrario, se quiere proporcionar el cuerpo de cada método después de terminar la declaración de la clase racional, lo único que hay que hacer es, en el archivo en que se defina el método, cambiar su encabezado por:

```
void racional :: setracional(long n, long d)
```

Donde el operador **::**, llamado **operador de resolución de alcance**, introduce la noción de alcance de la clase y especifica que lo que se define es un método de la clase racional. Es decir, el operador amplía el alcance de la clase. Una vez abierto este alcance se pueden usar los miembros datos privados *numerador* y *denominador* y el método privado *reducir*, directamente por su nombre, como si se estuviera dentro del alcance de la clase.

Para implementar la función privada *reducir*, primero debemos asegurarnos que **numerador** y **denominador** sean positivos, verificando el signo.

```
void racional :: reducir(void)
{
        int a,b,rem,sign;
        if (numerador == 0)
                denominador = 1;
        sign = 1;                       // lo supone positivo
        // verifica si hay negativos
        if (numerador < 0 && denominador < 0)
        {numerador = - numerador; denominador = - denominador;}
        if (numerador < 0)
        {numerador = - denominador; sign = -1;}
        if (denominador < 0)
        {denominador = - denominador; sign = -1;}
        if (numerador > denominador)
                {a = numerador; b = denominador;}
        else
                {a = denominador; b = numerador;}
        // ahora calcula el MCD del numerador y del denominador
        while (b != 0)
                {rem = a % b; a = b; b = rem;}
        numerador = sign * numerador / a;
        denominador = denominador / a;
};              // fin de reducir
```

Para implementar la función *suma* de dos número racionales, primeros debemos reducir cada uno de sus términos a mínimo, luego multiplicamos los denominadores para producir un denominador resultante, y luego multiplicamos cada numerador cada numerador por el denominador del otro racional y sumamos los productos para obtener el numerador resultante. Por último, reducimos la suma.

El riesgo que se corre en este caso es que el producto de los denominadores sea demasiado grande, aún para una variable long. Para evitarlo usamos el siguiente algoritmo para sumar a/b + c/d, suponiendo que el valor rden(x,y) denota el denominador de x/y reducido a términos mínimos.

```
k ← rden(b,d)
// denominador resultante
denom ← b * k
// numerador resultante
num ← a * k + c * (denom/d)
```

Demostración

$$
\begin{aligned}
a / b + c / d &= a / b + (cb / db) \\
&= (a + (cb/d)) / b \\
&= k(a + (cb/d)) / bk \\
&= (ak + (cbk/d)) / bk \\
&= (ak + c(bk/d)) / bk
\end{aligned}
$$

Entonces, la suma se implementa:

```
racional racional :: add(racional r)
{
        int k, denom, num;
        racional rnl;
        // primero se reducen ambos racionales
        reducir();              // reduce el implícito
        r.reducir();            // envía el mensaje reducir a r
        // implementa la linea k = rden(b,d) del algoritmo
        rnl.setracional(denominador,r.denominador);
        rnl.reducir();  // envía el mensaje reducir a rnl
        k = rnl.denominador;
        // calcular el denominador y el numerador
        denom = denominador * k;
        num = numerador * k + r.numerador *(denom/r.denominador);
        // forma el resultado y lo reduce
        rnl.setracional(num,denom);
        rnl.reducir();
        return rnl;
};              // fin de add
```

Para implementar el método *multiplicar* para multiplicar dos racionales nuevamente reducimos a/b y c/d a términos mínimos, y después reducimos a/d y c/b para asegurarnos así que a*c no tiene términos comunes con b*d y que los productos son lo más pequeños posible.

```
racional racional :: mult(racional r)
{
        racional rnl, rnl1, rnl2;
        int num,denom;
        // primero se reducen ambos racionales
        reducir();              // reduce el implícito
        r.reducir();            // envía el mensaje reducir a r
        // intercambia y reduce
        rnl1.setracional(numerador,r.denominador);
        rnl1.reducir(); // envía el mensaje reducir a rnl1
        rnl2.setracional(r.numerador,denominador);
        rnl2.reducir(); // envía el mensaje reducir a rnl2
        // calcula el resultado
        num = rnl1.numerador * rnl2.numerador;
        denom = rnl1.denominador * rnl2.denominador;
        // forma el resultado
        rnl.setracional(num,denom);
        return rnl;
};              // fin de mult
```

La implementación del método *divide* simplemente multiplica por un recíproco.

```
racional racional :: divide(racional r)
{
        racional rnl;
        // calcula el reciproco de r
        rnl.setracional(r.denominador, r.numerador);
        // multiplica por el reciproco
        return mult(rnl);
};      // fin de divide
```

El método *equal* reduce ambos racionales y luego verifica la igualdad de numeradores y denominadores.

```
int racional :: equal(racional r)
{
        reducir();
        r.reducir();
        if (numerador == r.nuemrador &&
                        denominador == r.denominador)
                return 1;
        else
                return 0;
};      // fin de equal
```

Para implementar el método *print* definimos el siguiente formato: el numerador seguido de / seguida del denominador.

```
void racional :: print(void)
{
        cout << numerador << "/" << denominador << endl;
};      // fin del print
```

Esta implementación exige la inclusión de **#include <iostream.h>** en el encabezado del programa.

IV.2 Uso de la clase racional

Suponga que la clase racional se declaró en un archivo de encabezado **racional.h**.

Queremos escribir un programa que reciba como datos líneas de la forma:

```
op ra rb;       donde: op es el carácter + o *
                       ra y rb son enteros de la forma a/b, con a y b
                       enteros.
```

Ejemplos:

```
+ 3 7;          → 10
+ 3/4 5;        → 23/4
* 3 4/8;        → 3/2
* 3/4 5/6;      → 5/8
```

El programa debe leer la línea, efectuar el cálculo solicitado e imprimir el resultado.

Supondremos además la existencia de rutina para ayudar con la entrada/salida.

int readtoken (char)** Lee el siguiente operador o entero en forma de carácter("/" o "5"), asigna almacenamiento para la cadena que usa la función calloc de stdlib y desasigna un puntero de tipo char* hacia ella. Si se encuentra al final del archivo, readtoken devuelve el valor EOF y sino retorna un !EOF.

long atol(char*) Es de stdlib y convierte una cadena numérica en un entero.

void error(char*) Imprime sus parámetros como un mensaje de error y detiene la ejecución.

Entonces, el programa ejemplo de uso de la racional, tendría la forma:

```
#include "racional.h"
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
void main()
{
        int readtoken(char**);
        void error(char*);
        char *optr,*token1,*token2,*token3;
        int int1,int2:
        // una declaracion que convierte las variables en objetos de la
        // clase racional donde cada uno de ellos tiene un numerador y
        // un denominador y puede usarse para llamar a los metodos de
        // la clase racional
        racional opnd1,opnd2,result;

        //se lee el operador
        while (readtoken(&optr) != EOF)
                readtoken(&token1);             // lee el primer entero de la cadena
                int1 = atol(token1);            // convierte a entero
                readtoken(&token2);
                if (strcmp(token2,"/") != 0)
                        opnd1.setracional(int1,1);
                        //convierte oprando entero a racional
                else
                {
                        // obtiene el denominador
                        readtoken(&token3);
                        int2 = atol(token3);
                        // convierte a racional
                        opnd1.setracional(int1,in2);
                        readtoken(&token2);
                }
```

```
                // obtiene el segundo operando
                int1 = atol(token2);
                readtoken(&token2);
                if (strcmp(token2,"/") != 0)
                        opnd2.setracional(int1,1);
                else
                {
                        readtoken(&token3);
                        int2 = atol(token3);
                        opnd2.setracional(int1,in2);
                        readtoken(&token2);
                }
                if (strcmp(token2,";") != 0)
                        error("Error: no se encontro el ; esperado");
                // aplica el operador a los dos operandos racionales
                if (*optr == '+')
                        result = opnd1.add(opnd2);
                else if (*optr == '*')
                        result = opnd1.mult(opnd2);
                else
                        error("Error: operador ilegal");
                result.print();
        }       // fin del while
};      // fin del main
```

IV.3 Homonimia

Aunque disponemos de una rutina para sumar dos números racionales, no podemos sumar un racional **r** y un entero **i** sin antes convertir **i** a racional usando setracional(i,1).

Afortunadamente, C++ permite usar el mismo nombre para diferentes funciones siempre que sus parámetros sean de tipos diferentes y/o en cantidades diferentes. En otras palabras, siempre que la **firma** de las dos funciones sea diferente. Podríamos agregar a la clase racional otro método add, incluyendo en la sección pública la línea:

```
racional add(long);          // de distinta firma que el add original
```

cuya implementación directa sería:

```
racional :: racional add(long i);
{
        racional r;
        r.setracional(i,1)
        return add(r);
};
```

que forma el nuevo racional y llama a la función add preexistente para racionales para sumar r al racional del objeto actual.

De esta manera, si i es un entero, la llamada rr.add(i) es una llamada a este segundo método add para sumar un entero al racional rr.

Por el contrario, si r es un racional, la llamada rr.add(r) es una llamada al primer método add, para sumar un racional r a rr.

IV.4 Herencia

Pero esta técnica para sumar un entero a un racional no es del todo satisfactoria, ya que siempre el objeto actual es el primer operando. Esto nos permite implementar **r + i**, pero no podemos implementar directamente **i + r**. En el caso de la suma, que es conmutativa, no es un gran problema. Pero la división no es conmutativa. Podríamos escribir un método racional divide(int) para calcular r/i, pero, ¿Cómo calculamos i/r?

Otra manera de encarar este problema sería escribir una rutina separada, que no sea parte de la clase, con dos parámetros:

```
racional divide(long i, racional r)
{
        racional rr;
        rr.setracional(i,1);
        return rr.divide(r);
};
```

Pero esta solución hace que el operador de división no sea simétrico y rompe con el concepto de operaciones de clase ya que estaríamos definiendo una operación entre objetos racionales y objetos enteros que no es de la clase racional.

Si, por el contrario, consideramos que los enteros son una forma de números racionales, ya que cada entero se puede representar como una racional (i/1). Vamos a definir una nueva clase, la **clase entero**, de modo que **herede** elementos de la clase racional.

Para asegurarnos de que el denominador sea siempre 1 en esta nueva clase, redefiniremos el método setracional, más aún, daremos dos nuevas versiones del mismo.

```
class entero : public racional
{
public:
        void setracional(long,long);
        void setracional(long);
};
```

Así:

La clase racional se denomina **clase base** de la clase entero.
La especificación **public** en el encabezamiento de la clase entero indica que ésta tiene acceso a los elementos protegidos y públicos de la clase racional, que se convertirán en elementos protegidos y públicos de la clase entero.
Si se hubiera escrito **protected** en lugar de public, los elementos públicos de racional serían protegidos en entero.
Si se hubiera escrito **private** en lugar de public, los elementos públicos y protegidos de racional serían privados en entero.

Sin embargo, la clase entero no puede acceder a los datos miembros numerador y denominador, privados de la clase racional. Para que pueda hacerlo, deberían ser definidos como protegidos:

```
class racional
{
protected:
        long numerador;
        long denominador;
        void reducir(void);
public:
        racional add(racional);
        racional mult(racional);
        racional divide(racional);
        int equal(racional);
        void print(void);
        void setracional(long,long)
};
```

Implementamos ahora los dos métodos setracional declarados en la clase entero. El primer setracional se incluye para **invalidar la función setracional de la clase racional, de modo que no se asigne accidentalmente un racional no entero a un objeto de tipo entero**.

```
void entero :: setracional(long num, long denom)
{
        if (denom != 1)
                error("Error: no entero asignado a variable entera);
        numerador = num;
        denominador = 1;
};
```

La otra versión es más común:

```
        void entero :: setracional(long num)
{
        numerador = num;
        denominador = 1;
};
```

Ahora, si **r** es un racional e **i** un entero, son válidas cualesquiera de las siguientes llamadas:

```
r.add(i);
i.add(r);
r.divide(i);
i.divide(r);
```

Los métodos *add* y *divide*, definidos en la clase *racional*, son heredados por la clase *entero* y se invocan para una variable entera.

IV.5 Constructores

Un **constructor es un método especial de una clase que se invoca automáticamente cada vez que se crea un objeto de dicha clase**.

El constructor recibe el mismo nombre que la clase.

En el ejemplo anterior, usamos el método racional para inicializar el objeto de tipo racional, pero podríamos haber usado el constructor.

Definamos las siguientes funciones miembros de la clase racional, todas públicas y todas del mismo nombre: racional, aunque de distinta firma:

racional(void); racional(long); racional(long,long);

y las implementemos de la siguiente manera:

```
racional :: racional(void)
{
        // supone que el racional es 0
        numerador = 0;
        denominador = 1;
};
```

```
racional :: racional(long i)
{
        numerador = i;
        denominador = 1;
};
```

```
racional :: racional(long num, long denom)
{
        numerador = num;
        denominador = denom;
};
```

Así, cada declaración de un objeto racional, invocará al **constructor apropiado**:

racional r;	**inicializa en el 0 racional r(0/1) automáticamente. No hay parámetros.**
racional r(3);	**inicializa r en el racional 3/1 invocando a la segunda versión del constructor.**
racional r(2,5);	**inicializa r en el racional 2/5 invocando a la tercera versión del constructor.**

En C++, el operador **new** aplicado a un identificador de tipo, asigna un nuevo objeto del tipo dado, y devuelve un puntero a él. Cuando se invoca new también se invoca, automáticamente, el constructor. Entonces:

```
racional*p = new racional
```

declara una variable puntero p, le asigna un nuevo objeto de tipo racional, la inicializa a cero (constructor sin argumentos) y hace que p apunte al objeto.

Por otro lado,

```
racional*p = new racional(2,5)
```

hace que p apunte a un objeto racional inicializado en el valor 2/5.

El constructor también se puede usar de la siguiente manera:

```
r = racional(7)
```

que asigna a r el racional 7/1.

V. Clases y Abstracción de datos

Todos los objetos del mundo real tienen en común:

- **atributos** Que describen su tamaño, forma, peso, color, etc.
- **comportamientos** El balón rueda, rebota, se infla, etc.
  El bebé llora, duerme, gatea, parpadea, etc.
  El automóvil se acelera, se frena, gira, etc.

Los seres humanos aprenden lo relacionado con los objetos estudiando sus atributos y observando sus comportamientos.

Objetos diferentes pueden tener muchos atributos iguales y mostrar comportamientos similares. Se pueden así hacer comparaciones. Por ejemplo, entre bebés y adultos, entre seres humanos y chimpancés, entre autos, camiones y patines.

> **La programación Orientada a Objetos (P.O.O.) hace modelos del mundo real mediante sus contrapartes en software. También modela la comunicación entre objetos mediante mensajes.**

La P.O.O. aprovecha:

a) **Relaciones de clase** Donde objetos de una cierta clase tienen las mismas características.

b) **Relaciones de herencia** Donde se derivan clases nuevas heredando características de clases existentes, y agregando características propias, únicas.

> **La P.O.O. encapsula datos (atributos) y funciones (comportamientos) en paquetes llamados objetos.**

En este sentido, los objetos **<u>ocultan información</u>** a sus usuarios ya que éstos saben como comunicarse con ellos mediante interfaces bien definidas, pero no saben cómo funcionan, o más bien, cómo están implementados esos objetos.

Los programadores de C++ crean sus propios tipos definidos por el usuario, conocidos como **clases**.

Cada clase, o mejor dicho, los objetos que componen una clase, cuentan con:

- miembros datos Componentes de datos
- funciones miembros Funciones que manipulan esos datos.

Los **miembros datos**, también llamados **variables de instancia** o **variables de clase**, describen los atributos, características y entidades de los objetos de la clase.

Las **funciones miembros**, también llamadas **métodos**, definen operaciones que pueden realizar esos objetos, es decir, su comportamiento.

V.1 Las estructuras en C

Las clases en C++ son un desarrollo o evolución natural de una **struct** en C.
En el siguiente ejemplo analizamos:

```
struct time
{
        int hour;       // 0-23
        int minute;     // 0-59
        int second;     // 0-59
}
```

• La palabra reservada **struct** presenta la definición de la clase.

• **time** es la etiqueta o nombre de la estructura. Se usa para declarar variables del tipo estructura.

• Los nombres declarados entre las llaves son los miembros de la estructura. Sus nombres debe ser únicos dentro de la misma estructura. Sus miembros pueden ser de cualquier tipo, salvo del mismo tipo estructura, pero si puede haber un puntero a **time** (estructura auto-referenciada), que son útiles para formar estructuras de datos enlazadas.

• La definición de la estructura no reserva espacio en memoria, más bien, crea un nuevo tipo de datos que es usado para declarar variables.

• Las variables del tipo de la estructura se declaran, al igual que las variables de otros tipos:

time timeobj, timearr[10],*timeptr

• Los miembros de una estructura se acceden usando los operadores de acceso a miembros que son:

- . operador de acceso directo vía el nombre de la variable
- → operador de acceso indirecto vía una referencia a la variable.

Ejemplos:

```
cout << timeobject.hour;
cout << timeptr → hour; //cout << (*timeptr).hour;
```

Problemas con las estructuras de C

- La inicialización no es específicamente requerida. Es posible tener datos sin inicializar, o no inicializados correctamente. El programador manipula de forma directa el tipo de datos. No existe interfaz con el programa para asegurarse que el programador use correctamente el tipo de datos y para asegurar que los estados se conservan en estado consistente.

- La estructuras de C no pueden ser impresas ni comparadas como una unidad, sino miembro a miembro.

V.2 Tipo abstracto de datos implementado como una clase

En C++ las clases (**class**, **struct** y **union**) implementan tipos abstractos de datos (TAD), encapsulando elementos de datos y operaciones (métodos) para crear, manipular y destruir objetos producidos a partir de estas estructuras de clases.

Definición 1:

Las clases permiten que el programador modele objetos que tienen atributos (representados por datos miembros) y comportamientos u operaciones (representados por funciones miembros o métodos).

Definición 2:

Una clase es un tipo definido por el usuario; es decir, un marco que permite crear objetos de su misma estructura.

Estos objetos, que pueden ser elementos de duración estática, local o dinámica, constan de:

- **Variables de clase y de instancia** (datos miembro, en C++), que son los descriptores de atributos y entidades de los objetos.

- **Métodos** (funciones miembro en C++), que definen las operaciones que pueden realizar esos objetos.

Los métodos son invocados en respuesta a ***mensajes*** enviados al objeto.
Un mensaje corresponde a una llamada a función miembro.

Una vez declarada y definida la clase, su nombre se puede usar para declarar objetos de la misma.

V.2.1 Declaración de una clase

- La declaración de una clase comienza con la palabra reservada **class | struct | union** que describe el tipo de clase de que se trata.
  La declaración de una clase crea un único tipo que, posteriormente, permite declarar objetos (instancias de la clase) y objetos derivados de ese tipo (tales como punteros, referencias, etc.).

  - **tipo class** Los miembros son **private** por defecto, pero pueden ser definidos como **public** o **protected**. Este tipo de clase puede tener varios constructores de objetos y puede participar en jerarquías de clases (herencia).

  - **tipo struct** Sus miembros son **public** por defecto, pero pueden ser definidos como private o protected. En lo demás son semejantes al tipo class.

  - **tipo union** Todos su miembros son public y no admite ningún tipo de especificador de acceso (public, private o protected). No pueden participar de jerarquías de clases y solamente pueden tener un único constructor.

  Las palabras **struct** y **union** son obligatorias en C para declarar un objeto, pero no en C++. Aquí solamente son necesarias cuando los nombres de las clases Y o Z quedan ocultos.

```
struct clas1
{
        int i,j;
};
int clas1;              // oculta a struct clas1
void main ( )
{
        clas1 = 12;     // correcto, le asigna 12 a la variable int clas1
        clas1 x;        // error, la clase esta oculta
        struct clas1 x;         // correcto
};
```

  El **nombre de la clase** es un identificador (omitible) único en su ámbito. Si se omite, la declaración de la clase constituye una **"plantilla"** que precisa que se declaren los objetos a continuación.

```
// declaración de una plantilla union
union
{
        int i;
        double x;
} mu, *pmu;
```

La "plantilla" no puede tener constructores, ni destructores, ni se puede pasar como argumento, ni retornar un valor, ya que no tiene un nombre que posibilite su equiparación con un argumento formal.

El nombre de una clase **oculta** cualquier nombre de clase, objeto, enumeración o función con el mismo nombre en ese ámbito. Para poder usar el nombre de un objeto oculto hay que aplicar el "especificador" de tipo elaborado: **class**, **struct** o **union**.

- El cuerpo de la clase se delimita mediante { }

- La definición de la clase termina con ;.

- Una clase puede estar acompañada de un especificador de almacenamiento **extern** o **static**. El calificador **extern** (explícito o implícito (default)) indica que la clase tiene visibilidad de todos los archivos que componen el programa y que el enlazado de todos los identificadores de la clase es externo, es decir, durante la ejecución del montador de enlaces. Si, por el contrario, se usa el calificador **static**, la visibilidad queda restringida a ese archivo, con enlazado interno, en tiempo de compilación.

- Una clase puede tener **especificadores de acceso** a miembros que pueden ser **public**, **private** o **protected**. Estos especificadores de acceso terminan con un **:** y pueden aparecer varias veces en una definición de clase.

- Los **miembros datos** de la clase se describen en una lista que puede incluir datos de cualquier tipo: enumeraciones, campos de bits y objetos de otras clases. No puede ser un miembro dato de la misma clase, aunque si un puntero o referencia a ella.

- La declaración de la clase contiene los **prototipos** de la funciones miembros. Las funciones miembros permiten efectuar las operaciones definidas en el tipo abstracto de datos. Se las declara, y si procede, se las define dentro del cuerpo de la clase. El tipo de retorno de una función miembro puede ser cualquiera, incluso **void**, excepto un array, otra función y los punteros a ambas declaraciones.

- Tanto los datos miembros como las funciones miembros admiten el especificador de clase de almacenamiento **static** y los modificadores de acceso **const** (su valor no admite modificaciones) y **volatile** (su valor puede ser modificado fuera del control del compilador).
  La lista de miembros de una clase se escribe entre las { } y define el conjunto completo de miembros de la clase. Por esta razón, los miembros no pueden ser declarados más de una vez o añadidos posteriormente. De la misma manera, un identificador no puede ser simultáneamente un miembro dato y una función miembro.

- Cuando en la definición de la clase se han incluido solamente los prototipos de las funciones miembros, para definir cada función, fuera del alcance de la clase, se debe usar el **operador de resolución de alcance ::**. Esto es, después de la } que termina la definición de la clase. Este operador indica que la función miembro que se está definiendo está dentro del alcance de la clase, aunque se esté definiendo en otro archivo.

- Una función miembro con el mismo nombre de la clase recibe el nombre del **constructor de la clase**. Esta es una función especial que sirve para inicializar un objeto cuando se crea.

- El nombre de la clase se convierte en un especificador de tipo. El programador puede crear nuevos tipos (o clases) y tantos objetos de cada clase como necesite. Por esta razón, se dice que C++ es un lenguaje **extensible**.

- Cuando se produce un objeto de una clase, automáticamente, se llama al constructor de la clase que inicializa a cada miembro dato.

- La implementación de una clase está oculta a sus clientes. **Este ocultamiento de información** promueve la capacidad de modificación del programa y simplifica la percepción que el cliente de la clase tiene de ella.

- Una función con el mismo nombre de la clase, pero precedida por el carácter ~ (tilde) se llama **destructor** de la clase. Esta función se ocupa de los trabajos de terminación de cada objeto de la clase, antes de que la memoria correspondiente al sistema sea reclamada por el mismo. Obviamente, tiene sentido usarla cuando la asignación de memoria se hizo dinámicamente.

- C++ permite especificar **declaraciones de clases incompletas** (sin miembros) y efectuar referencias al nombre de esa clase (con punteros a la misma) antes de que la clase haya sido definida totalmente.

```
class clas1;            // defínición incompleta
int f(clas1 *px) {/*...*/}
... ... ...
class clas1 {/*...*/}   // definición completa
```

V.2.2 Declaración de objetos de tipo clase

Los objetos de una clase son instancias de la clase que se declaran del mismo modo que los objetos de cualquier tipo de datos y se crean en tiempo de ejecución, con la estructura definida en la clase.

Estos objetos pueden ser asignados, pasados como parámetros a funciones, devueltos como valor de retorno de funciones, etc., y pueden soportar otras funciones definidas por el usuario para objetos de la clase.

```
clas1 ob,obj, *obj1, &obj2)ob,obj3[10];

clas1 fun(clas1);      // prototipo de función (fun) con argumento y
                          retorno de la clase clas1.
obj = fun(ob);         // asignación de un objeto y llamada a función con
                          objeto como parámetro.
```

V.2.3 Alcance de clase y acceso a miembros de clase

El **alcance** de una clase incluye los nombres de variables y funciones declaradas en ella, y los nombres de datos y funciones miembros de la misma.

Las funciones no miembros de la clase se definen en **alcance de archivo**.

Es importante preguntarse como se acceden los miembros de una clase dentro y fuera de su alcance:

Dentro del alcance de la clase: Los miembros se acceden directamente, referenciados por su nombre. Obviamente este acceso sólo está permitido a las funciones miembros de la clase.

Fuera del alcance de la clase: Los miembros de la clase se referencian a través del nombre de un objeto de la clase, una referencia a un objeto de la clase o un puntero a un objeto de la clase.

Las funciones miembros pueden tener homónimas dentro de la misma clase.

Las funciones miembros tienen alcance de función dentro de una clase. Si una función miembro define una variable del mismo nombre que una variable con alcance de clase, ésta última queda oculta por la variable con alcance de función., dentro del alcance de la función. Se puede tener acceso a variables ocultas mediante el uso del **operador de resolución de alcance**:

- binario Se construye: <nombre de clase>**::**variable de instancia
- unario Se construye: **::**variable global

Si ***m*** es un miembro de la clase ***X***, y esta clase queda oculta, se puede acceder a ***m*** usando el nombre calificado: ***X :: m***.

Si se usa un objeto de la clase ***Y*** como dato miembro de la clase ***X*** (composición de clases), la clase ***Y*** debería haber sido definida previamente. En caso contrario se producirá un error. Pero si se usa un puntero o referencia a la clase ***Y***, bastaría con una declaración incompleta de Y ("**declaración forward**"), previa a la de ***X***.

```
class Y;        // clase incompleta. Exige una declaración posterior
class X
{
        Y *py;          // objeto puntero de la clase Y
};
```

Para acceder a los miembros de una clase se usan, ya sea, el operador de selección directo • o el operador de selección indirecto → de la misma manera que en la estructuras.

V.3 Constructores y Destructores

Estas son funciones miembros especiales de una clase que especifican la forma en que los objetos de la clase son: creados, inicializados, copiados y destruidos. En general, tienen las mismas características que las funciones miembros de la clase, con algunas particularidades que veremos a continuación:

V.3.1 Constructores

Después que los objetos son creados, sus miembros pueden ser inicializados mediante un constructor, que es una función miembro de la clase, provista por el programador, con el mismo nombre de la clase, que:

- Cada vez que se crea un objeto de la clase, el constructor es invocado automáticamente.
- Los constructores de los objetos globales (declarados fuera de la función) son ejecutados antes que la función **main**, mientras que los objetos locales se crean en el instante en que el ámbito del objeto se hace activo, y en ese momento se referencia al constructor.
- Los miembros datos de la clase no pueden ser inicializados en la definición de la clase.
- Los constructores no pueden especificar ni tipo ni valores de retorno.
- Los constructores pueden ser sujetos de homonimia. (**Una clase puede tener varios constructores**), para permitir la variedad de maneras de inicializar objetos de la clase que se necesiten.
- Cuando se declara un objeto de una clase, se pueden dar **inicializadores**, entre paréntesis, que serán pasados como parámetros al constructor de la clase.

  racional p(1,2); ⇒ p se inicializará en ½.

- Los constructores pueden tener argumentos por omisión y garantizar así que el objeto esté en estado consistente.

  time **::** time( ) { hour = minute = second = 0;}

Si en la llamada al constructor no de dan valores de inicialización, el objeto igualmente estará en estado consistente.

- Si una clase no define un constructor, el compilador creará un **constructor por omisión**, como public, que ejecutará ninguna inicialización. Por esta razón no se garantizará que el objeto esté en estado consistente. La función del constructor será la de crear y copiar objetos.

- No se puede acceder a la dirección de los constructores ni se pueden referenciar como una función miembro.

- No se puede definir un constructor de una clase que tenga objetos de la misma clase como parámetros. Pero si puede tener como parámetros una referencia a su propia clase. En este caso el constructor se llama un constructor copia:

```
class cla1
{
public:
        cla1(const cla1&)    // correcto
        cla1(cla1);          // error
}
```

- Un constructor no puede ser: **const**, **volatile** o **static**. Tampoco puede ser heredado por una clase derivada.

V.3.1.1 Tipos de constructores

Los constructores vienen definidos por los usuarios teniendo en cuenta el tipo de objetos que se quieren crear, copiar o inicializar. Pero existen constructores particulares tales como:

a) Constructor por defecto:

Es aquel que **no tiene argumentos**. Si el usuario no ha definido un constructor para la clase, el sistema genera uno por defecto para poder crear objetos de esa clase. Se diferencia del que tiene por argumento el valor del parámetro por defecto.

b) Constructor sobrecargado

Cuando existe más de un constructor para una clase, se produce una **sobrecarga** del mismo. El sistema resuelve la ambigüedad, cuando crea un objeto, basándose en el número y tipo de los argumentos y de los valores usados en la inicialización.

c) Constructor copia

Es aquél que se define teniendo **como parámetro del constructor una referencia de su propia clase**, de alguna de las dos siguientes maneras:

```
Y :: Y(const Y&)
ó
Y :: Y(const Y& int = 0)
```

Este constructor se invoca cuando se copia un objeto de una clase o cuando se declara un objeto y se inicializa con otro objeto de la clase.

```
X a(20);      // llama al constructor X :: X(int)
X d(a);       // llama al constructor copia X :: X(const X&)
X c = a;      // llama al constructor copia X :: X(const X&)
c = a;        // no llama al constructor copia. Es una asignación
```

C++ genera un constructor copia para la clase si lo precisa en la ejecución del programa y no ha sido definido por el usuario de la clase.

V.3.1.2 Inicialización de los objetos y miembros de una clase

Cuando un objeto se crea, se produce normalmente, un proceso de inicialización asignando el usuario valores al objeto, y el constructor valores a sus miembros.

a) Asignación de valores a los objetos

- Cuando la clase **no tiene un constructor explícito** de usuario, ni datos miembros no estáticos que sean privados o protegidos, ni funciones virtuales, ni tiene jerarquía de clases, entonces es un **"agregado"** y los objetos pueden ser inicializados mediante una ***lista inicializadora***.

```
class ini                    {
public:
        int i,j;
        ...
};
ini z = {20,2};          // z.ini :: i == 20, z.ini :: j == 2;
// los miembros i,j deben ser públicos para poder
// accederlos
```

- Cuando la clase tiene un **constructor explícito**, los objetos suelen ser inicializados mediante una **lista expresión** con paréntesis (valores de los parámetros actuales del constructor, entre paréntesis).

  Alternativamente, se puede usar como inicializador un simple valor, usando el **operador ==**, que crea un objeto temporal y luego lo copia en el objeto que se quiere inicializar.

```
class X
{
        int i,j,k;
public:
        X ( );          // constructor por defecto
        X(int);         // constructor sobrecargado
        X(const X&); // constructor copia
        X(int,int,int); // constructor sobrecargado
        ...
};
void main ( )
{
        X a;            // X :: X( )
        X b(20);        // X :: X(int); 20 es la lista expresión
        X c = a;        // X :: X(const X&)
        X d = 20;       // X :: X(X(20))
        X x(5,20,38); // X :: X(int,int,int);
        // (5,20,38) es la lista expresión
        ...
}
```

b) Asignación del constructor de valores a sus miembros

Por otra parte, el constructor puede asignar valores a sus miembros de dos formas:

- Aceptando los valores de inicialización como parámetros del constructor y asignándolos a los miembros datos dentro del cuerpo del constructor.

```
class Z
{
        int a,b;
public:
        Z (int i, int j) {a = i, b = j;}
        ...
};
```

- Usando una lista de parámetros inicializadores de expresión antes del cuerpo de la función. Esta lista debe figurar en la definición del constructor, no en su declaración, y cada "miembro dato" sólo puede ser invocado una sola vez en la lista.

```
class Z
{
        int a,b;
public:
        Z (int i, int j): a(i), b(a + j) {/*...*/}
        ...
};
void main ( )
{
        Z z (1,2);      // z.Z :: a == 1; z.Z :: b == 3
        //...
}
```

V.3.2 Destructores

Un destructor es una función especial, miembro de la clase, cuyo nombre el igual al nombre de la clase pero precedida por el carácter **tilde (~)**. En cierto sentido, el destructor es el complemento del constructor. La función destructor es referenciada cuando se libera un objeto antes de ser destruido. La función destructor es **única.** (Solamente existe un único modo de destruir objetos de la clase).

- El destructor de una clase es llamado automáticamente cuando el objeto de la clase se sale de alcance. De hecho, el destructor no "destruye" el objeto (no implica la liberación del espacio que ocupa el objeto), sino que realiza **tareas de terminación** antes de que el sistema recupere el espacio de memoria ocupado por el objeto para que pueda ser usado por nuevos objetos.

- El destructor no recibe parámetros ni tiene valor de retorno.
  Tampoco puede ser **static**, ni ser heredado por una clase derivada, ni ser declarado **const** ni **volatile**.

- Una clase solo puede terne un destructor. No se permite la homonimia de destructores.

- El destructor puede ser invocado explícitamente para terminar un objeto, pero en cualquier caso, es referenciado por el sistema cuando el objeto sale de su ámbito. La excepción son los objetos punteros creados con **new** que precisan que se invoque al operador **delete** para que el destructor sea referenciado.

- El destructor de un objeto no destruye el propio objeto cuando es referenciado ya que el objeto se destruye cuando sale de su ámbito. Por ello, es posible llamar explícitamente, varias veces, al destructor si liberar la memoria asignada al objeto. Por el contrario, cuando se trata de objetos creados con el operador **new**, se precisa la invocación del operador **delete** para la liberación de la memoria asignada del **heap** del sistema, ya que el objeto no se destruye cuando sale de su ámbito.

- El destructor puede ser invocado explícitamente de dos modos:

  a) **Indirectamente**: Usando el operador **delete** con un objeto creado con el operador **new**. Los objetos creados con **new** deben ser destruidos con **delete**.

```
class X {
public:
        X( );
        ~X( );
        ...
};
void main( ) {
        X *x = new X;
        delete x;
        // llama al destructor ~X( );
        ...
};
```

  b) **Directamente**: usando el nombre del destructor.

```
X p;          // crea un objeto p de la clase X
p.~X( );      // p.X :: ~X( );
```

  La llamada explícita a un destructor se puede efectuar para un objeto de cualquier tipo, pero si el objeto no tiene destructor, no produce ningún efecto. Por ejemplo, el tipo flota no tiene destructor, y aún así se puede programar:

```
float *pf;
...
pf → float :: ~float( );
```

V.4 Orden de llamada a constructores y destructores

En general, las llamadas a constructores y destructores son realizadas de forma automática.

El orden en que son hechas las llamadas depende del orden en el cual los objetos entran y salen de alcance.

Por lo general, las llamadas a destructor se efectúan en **orden inverso** a las llamadas a constructor. Sin embargo, la duración de almacenamiento (**persistencia**) de los objetos puede modificar el orden en que los destructores son llamados.

- Para objetos declarados de **alcance global**, se llama al constructor al principio de la ejecución del programa y a los destructores correspondientes a la terminación del programa.

- Para objetos **locales automáticos** los constructores se invocan cuando son declarados y los destructores cuando los objetos salen de alcance. Nótese que tanto los constructores como los destructores podrían ser llamados muchas veces, conforme los objetos entren y salgan de alcance.

- Para los objetos **locales estáticos**, los constructores son llamados una vez cuando estos objetos se declaran y los destructores correspondientes a la terminación del programa.

Cuando se crea un **array** de objetos , los constructores de sus elementos son llamados en orden creciente de subíndice.

Ejemplo: Para mostrar el orden en que son llamados los constructores y destructores.

//CREATE.H
//Definición de la clase crearydestruir
//La funciones miembros se definen en el archivo CREATE.CPP

```
#ifndef CREATE_H
#define CREATE_H

class crearydestruir
{
        int data;
public:
        crearydestruir(int);          //constructor
        ~crearydestruir( );           //destructor
};
#endif
```

//CREATE.CPP
//Funciones miembros de la clase crearydestruir

```
#include <iostream.h>
#include “create.h”
crearydestruir :: crearydestruir(int valor)
{
                data = valor;
                cout << “Objeto “ << data << “constructor”;
};
```

```
crearydestruir :: ~crearydestruir( )
{
        cout << “Objeto “ << data << “ destructor” << endl;
};
//PROGRAMA.CPP
//Mostrar el orden de llamada a constructores y destructores
#include <iostream.h>
#include “create.h”

void crear(void);               //prototipo
crearydestruir prim(1);         // objeto global

main ( )
{
        cout << “       (global creado antes de main)\n”;

        crearydestruir seg(2);          //objeto local a main
        cout << “       (local automático en main)\n”;

        static crearydestruir ter(3);  //objeto local static
        cout << “       (local estatico en main)\n”;

        create( );      //llamado a funcion para crear objetos

        crearydestruir cuar(4);         //objeto local a main
        cout << “       (local automático en main)\n”;

        return 0;
};
//Funcion para crear objetos
void create(void)
{
        crearydestruir quin(5);
        cout << “       (local automatico en create)\n”;

        static crearydestruir sext(6);
        cout << “       (local estatico en create)\n”;

        crearydestruir sept(7);
        cout << “       (local automatico en create)\n”;
};
```

La salida de este programa sería:

Objeto 1 constructor	(global creado antes de main)	
Objeto 2 constructor	(local automatico en main)	
Objeto 3 constructor	(local estatico en main)	
Objeto 5 constructor	(local automatico en create)	
Objeto 6 constructor	(local estatico en create)	
Objeto 7 constructor	(local automático en create)	
Objeto 7 destructor		
Objeto 5 destructor	**// son locales automáticos en create**	
Objeto 4 constructor	(local automático en main)	
Objeto 4 destructor		
Objeto 2 destructor	**//son locales automáticos en main**	
Objeto 6 destructor		
Objeto 3 destructor	**//son locales estaticos**	Los destruye al
Objeto 1 destructor	**//es global**	finalizar la ejecución en orden inverso al de creación.

V.5 Como usar datos y funciones miembros

Los **datos miembros privados** de una clase solamente pueden ser manipulados por funciones miembros y amigos de la clase.

Con frecuencia se definen **funciones miembros públicas** para permitir a los clientes de la clase definir (escribir) u obtener (leer) los valores de los datos miembros privados. A menudo, estas funciones son del tipo "**set**" y "**get**", pero no necesitan serlo.

Esta idea parece ser equivalente a definir miembros datos como públicos, pero no lo es. Los miembros datos públicos pueden ser manipulados a voluntad por cualquier función del programa, mientras que los miembros datos privados, leídos por una función pública "get" serán validados por la misma función en cuanto a su formato y exhibición. (Obviamente es el programador quien deberá proveer la verificación de validez para tener el beneficio de la integridad de los datos).

Una **referencia a un objeto** es un seudónimo del objeto mismo, y por lo tanto puede ser usada al lado izquierdo de una asignación. Así, una referencia es un **lvalue** (valor a izquierda) que es aceptable que reciba un valor.

Desafortunadamente, una forma para aprovechar esta capacidad es hacer que una función miembro pública de una clase retorne una referencia a un datos miembro privado de la misma. Esta es en realidad una mala práctica de programación y no debería usarse.

El **operador de asignación (=)** es usado para asignar un objeto a otro objeto del mismo tipo. En realidad se lleva a cabo una **copia miembro a miembro**, o copia por omisión.

Los objetos pueden ser pasados como **argumentos de función** y pueden ser **valores de retorno** de función.

VI. Objetos constantes y funciones miembros const

Algunos objetos necesitan ser modificables y otros o. Se usa la palabra reservada **const** para indicar que un objeto no es modificable y que cualquier intento de modificarlo sería un error.

```
const time mediodia(12,0,0);
```

Este concepto ayuda a que se cumpla el **principio de mínimo privilegio**, ya que cualquier intento accidental de modificar el objeto será detectado, en tiempo de compilación en lugar de causar errores en tiempo de ejecución.

Solamente las funciones const pueden operar sobre objetos const, aunque, obviamente, no pueden modificarlos. En otras palabras, una función miembro const puede ser llamada por objetos constantes y no constantes, mientras que una función miembro no constante sólo puede ser llamada por objetos no constantes.

Una función miembro se define como **const** si tanto en su declaración como en su definiciones inserta la palabra reservada const después de la lista de parámetros de la función:

```
int getvalue( ) const {return miembrodatoprivado;}
```

- Para constructores y destructores de objetos const no se requiere la palabra const.
- Un objeto const no puede ser modificado por asignación, por lo que deberá ser inicializado mediante un inicializador de miembro que proporciona al constructor valores iniciales correspondientes al objeto.

```
class incremento {
public:
        incremento(int c = 0, int I = 1);
        void addincrem( ) {count +=incremento;}
private:
        int count;
        const int incremento;          //dato miembro const
};
//Constructor
incremento :: incremento(int c, int i)
                : incremento( i )       //inicializador de miembro
{count = c;}
```

La notación **: incremento(i)** hace que incremento, que es un dato miembro const de la clase incremento, sea inicializado en el valor i. Si se requieren varios valores inicializadores de miembro, se incluyen después de los **:** en una lista separada por comas.

Tanto los objetos como las variables const deben ser inicializados con esta sintaxis. Las asignaciones no están permitidas.

VI.1 Composición: Clases como miembros de otras clases

Una clase puede tener otras clases como miembros. Por ejemplo, la clase ***relojalarma*** necesita saber cuando se supone que debe sonar la alarma. Se puede incluir un objeto ***time*** como miembro de ***relojalarrma***. Tal capacidad recibe el nombre de **composición**.

Cuando un objeto entra en alcance, su constructor es llamado automáticamente, por lo que es preciso especificar cómo se pasan argumentos a constructores de objetos miembros.

> **Se construyen los objetos miembros antes de que los objetos de clase que los incluyen sean construidos.**

Ejemplo: Para mostrar la composición de clases.

```
//DATE1.H
//Declaracion de la clase date

#ifndef DATE1_H
#define DATE_H

class date{
public:
        date(int=1,int=1,int=1900);          //constructor default
        void print( ) const;                 //imprime fecha en format dd/mm/aa
private:
        int dia;// 1 a 31
        int mes;        //1 a 12
        int anio;       //cualquier anio
        int checkdia(int);                   //funcion utilitarian para validar fecha
};
#endif

//DATE1.CPP
// archivo de definicion de funciones miembros de la clase date
 #include <iostream.h>
#include “date.h”
//Constructor: confirma el valor adecuado de mes;
//llama a funcion utilitaria checkdia, para validar el valor de dia.
date :: date(int dy, int mm, int yr) {
                mes = (mm > 0 && mm <= 12) ? mm:1;          //valida el mes
                anio = yr;
                dia = checkdia(dy);                         //valida el dia
                cout << “Constructor del objeto date: “;
                print( );
                cout << endl;
};
```

```
//Funcion utilitaria checkdia
int date :: checkdia(int testdia){
        static int diaspormes[13]={0,31,28,31,30,31,30,31,31,30,31,30,31};
        if (mes != 2)
        {       if (testdia > 0 && testdia <= diaspormes[mes])
                        return testdia;
        }
        else
        {       int days = (anio % 400 == 0 || anio %4 == 0
                        && anio % 100 != 0) ? 29 : 28;
                if (testdia < 0 && testida <= days)
                                return testdia;
        }
        cout << "Dia " << testdia << " invalido.Seteado a 1\n";
        return 1;       // para dejar el objeto en estado consistente
};
//Imprimir el objeto date en el formato dd/mm/aa
void date :: print( ) const
{
        cout << dia << "/" << mes << "/" << anio;
};
//EMPLE1.H
//Declaracion de la clase empleado
# ifndef EMPLE1_H
#define EMPLE1_H
#include "date1.h"

class empleado {
public:
        empleado(char*, char*,int,int,int,int,int,int);         //constructor
        void print( ) const;
private:
        char apellido[25];
        char nombre[25];
        date nacimiento;                //es una fecha objeto de la clase date
        date ingreso;                   //es una fecha objeto de la clase date
};
#endif

//EMPLE1.CPP
//Archivo que define las funciones miembros de la clase empleado
#include <iostream.h>
#include <string.h>
#include "emple1.h"
#include "date1.h"
```

```
//Constructor de la clase empleado
empleado :: empleado(char* nom,char* apelli, int ndia, int nmes, int nanio,
                     int idia, int imes, int ianio);
          : nacimiento(ndia,nmes,nanio), ingreso(idia,imes,ianio)
          //lista de inicializador de miembro que llama al constructor de
          //date para inicializar los objetos miembros nacimiento e ingreso
{
        strcpy(nombre,nom,24),
        nombre[24] = '\0';
        strcpy(apellido,apelli,24);
        apellido[24] = '\0';
        cout << "Constructor del objeto empleado: " << nombre
                        << ', ' << apellido << endl;
};
//Funcion imprimir de un objeto de la clase empleado
void empleado :: print ( ) const {
        cout << apellido << ", " << nombre << " Tomado el :";
        ingreso.print( );       //llama a la funcion print con un objeto de clase date
        cout << "El cumpleaños es: ";
        nacimiento.print( );    //llama a la funcion print con un objeto de clase date
        cout << endl;
};
//PRUEBA.CPP
/Demostracion de como opera un objeto compuesto por otro objeto
#include <iostream.h>
#include "emple1.h"
main( ) {
        empleado em("Juan","Perez",16,10,55,7,9,76);   //Crea el objeto e inicializa
        cout << endl;
        em.print;

        cout << "\n Prueba del constructor dato con valores invalidos: \n";
        date dd(35,14,94);
        return 0;
};
```

La salida para esta prueba sería:

```
Constructor del objeto date 16/10/55
Constructor del objeto date 7/9/76
Constructor del objeto empleado: Juan Perez

Perez, Juan
Tomado el: 7/9/76 El cumpleaños es: 16/10/55

Prueba del constructor date con valores invalidos
dia 35 invalido. Seteado a 1.
Constructor del objeto date 1/1/94
```

VI.2 Funciones y clases amigas

Una función f es amiga de una clase X, cuando sin ser función miembro de X (está definida fuera del alcance de la clase X), tiene acceso a los miembros **privados** y **protegidos** de X.

Para declarar la función f como amiga de una clase, se antepone a su prototipo dentro de la clase X, la palabra reservada **friend**.

También se puede declarar una clase como amiga. Para declarar ***clasedos*** como amiga de ***claseuno***, se declara, como miembro de ***claseuno***:

```
friend clasedos;
```

- Las declaraciones de amistad no tienen relación con los conceptos de acceso a miembros públicos, privados o protegidos. Estas declaraciones pueden ser escritas en cualquier lugar de la clase. Generalmente se las pone después del encabezamiento de la clase y no se las antecede de ningún especificador de acceso a miembros.

- Las funciones amigas de una clase tienen acceso a todos los miembros de la clase pero no pueden usarlos directamente sin aplicar a un objeto de la clase.

- Si una clase o función se especifican como amigas y no han sido previamente declaradas, su nombre tiene el mismo ámbito que el de la clase que contiene la declaración de amistad. No obstante, no se puede declarar una función miembro de una clase como amiga de otra clase, antes de declarar completamente su clase.

```
class X;                        //Declaración incompleta
class Y {
        int f1( );
        friend int X :: f2( );  //Error. F2 no esta definida
};
class X{
        int f2( );
        friend int Y :: f1( );  //Correcto
};
```

- La amistad es **concedida** y no tomada. Para que la clase B sea un amigo de la clase A, la clase A debe declarar que B es su amigo.
- La amistad **no es simétrica ni transitiva**. Pero si **se hereda**.

La clase A es amigo de la clase B y la clase B es amigo de la clase C **no implica** que:

La clase B sea amigo de la clase A
La clase C sea amigo de la clase B
La clase A sea amigo de la clase C

- Es posible especificar **funciones homónimas** como amigos de una clase. Cada función homónima que se desea como amigo debe ser declarada en forma explícita, en la declaración de la clase, como amigo de ella.

VI.3 Qué es y como usar el puntero this

Aunque cada objeto de la clase mantiene su propia copia de los datos miembro de la clase, sólo existe una copia de cada función miembro. Por ello es necesario, para cada función miembro, conocer la identidad del objeto que la llama a fin de poder manipular correctamente los datos de ese objeto. Con ese fin, C++ usa el puntero **this**, que se pasa como parámetro oculto a la función y especifica la dirección de inicio de ese objeto.

Cada objeto mantiene un apuntador a sí mismo, llamado **puntero this**. Este es un argumento implícito en todas las referencias a miembros incluidos dentro de dicho objeto. De este modo, C++ se asegura de que es referenciado el objeto correcto cuando una función miembro de una clase referencia a otro miembro de una clase en relación con un objeto específico de dicha clase.

- **this** es, pues, una variable local de tipo puntero a la clase, disponible en el cuerpo de cualquier función miembro no estática. Recuerde que las funciones estáticas, al no estar ligadas a un objeto, no disponen del puntero this.

- El puntero this también puede ser usado en **forma explícita**. Usando la palabra reservada this, cada objeto puede determinar su propia dirección.

- El puntero this se usa de **manera implícita** para referenciar miembros datos y funciones miembros de un objeto.

- El **tipo del puntero this** depende del tipo del objeto y de si la función miembro en la que this es usado está declarada const o no. Por ejemplo:

  i) Si la función miembro de la clase empleado **no es const**
  entonces
  this tiene el tipo **empleado * const**
  (apuntador constante a un objeto empleado)

  ii) Si la función miembro de la clase empleado **es const**
  entonces
  this tiene el tipo **empleado const * const**
  (apuntador constante a un objeto empleado constante)

- Cualquier función miembro tiene acceso al puntero this al objeto para el cual está siendo invocada.

Ejemplo de uso del puntero this

```
//PRUEBA.CPP
//Uso del puntero this
#include <iostream.h>

class test {
public:
        test(int =0);           //constructor
        void print( ) const;
private:
        int x;
};
//Constructor
test :: test(int a) const {x = a; }
//Impresion
void test :: print( ) const {
    cout << "   x = " << x << "\n this → x = " << this → x
        << "\n (*this).x = " << (*this).x << "\n"; }
//Programa
main( ) {
    test a(12);
    a.print( );
    return 0; }
```

La salida sería:

```
    x = 12
this → x = 12
(*this).x = 12
```

- Un uso interesante del puntero this es impedir que un objeto sea asignado a sí mismo. La autoasignación puede causar errores serios cuando los objetos contienen punteros a almacenamientos asignados en forma dinámica mediante el operador new.

- Se puede retornar una referencia a un objeto de una clase A para permitir llamadas a funciones miembros de la clase A para **concatenarlas**.

  Por ejemplo, en la clase ***time*** definimos cada una de sus funciones miembros: ***settime, sethora, setminuto, setsegundo*** de una manera análoga a la siguiente:

```
&time time :: setminuto(int m) {
        minuto = (m >=0 && m < 60) ? m : 0;
        return *this;           //habilita el encadenado
}
```

Entonces, cada una retornará *this con un tipo de retorno &time.
De esta manera, el operador punto (**.**) se asocia de izquierda a derecha por la expresión:

t.sethora(18).setminuto(30).setsegundo(20);

donde se evalúa primero t.sethora(18) y retorna una referencia a t como el valor de esta llamada a función. El resto lo interpreta como t.setminuto(30).setsegundo(20). Ejecuta luego la llamada t.setminuto(30) y retorna el equivalente de t. Interpreta el resto como t.setsegundo(20).

VI.4 Gestión dinámica de memoria

Los objetos se ubican en la memoria de una computadora en cuatro zonas diferentes:

Segmento de datos: Para estos objetos (estáticos), la gestión de memoria se realiza en la fase de compilación (compilación - enlazado), y corresponde a objetos globales y a los declarados con **static**.

Pila (stack): Objetos locales para los que la gestión de memoria se realiza en la fase de ejecución, justamente cuando su bloque o función pasa a ser activo, y se desasignan de esa memoria cuando finaliza la ejecución de ese bloque o función. Son los objetos declarados con **auto** o **register**, implícita o explícitamente.

Registros: Idem Pila.

Heap del sistema: Son objetos dinámicos para los que la gestión de memoria se realiza en tiempo de ejecución de forma dinámica mediante el uso de las funciones **malloc**, **calloc**, **realloc** y **free** o de los operadores **new** y **delete**.

VI.4.1 Operador new

Su sintaxis simplificada es:

<objeto-puntero> = **new** <tipo-objeto>[<inicializador>];

donde:
<tipo-objeto> :: Cualquiera excepto "retorno de función"
<inicializador> :: <expresión> |
<expresión>, <inicializador>|
{<inicializador>[,]}

El operador **new** asigna al <objeto-puntero> la memoria necesaria para contener un valor del tipo del objeto <tipo-objeto>, devolviendo la dirección de la memoria asignada o NULL si no existe espacio suficiente.

Se puede asignar cualquier tipo de dato, excepto "retorno a una función", aunque puede ser puntero a una función, y se puede inicializar el objeto asignado. Si no se inicializa el objeto se crea con valor indefinido.

En C, si se tuviera el siguiente código:

```
nomtipo *nomtipoptr;
```

para crear dinámicamente un objeto de tipo nomtipo se diría:

```
nomtipoptr = malloc(sizeof(nomtipo));
```

La función malloc no tiene ningún procedimiento para inicializar el bloque asignado.

En C++, se escribe directamente:

```
nomtipoptr = new nomtipo;
```

El operador **new** crea en forma automática un objeto del tamaño apropiado, si existe un constructor disponible, lo llama y retorna un puntero del tipo correcto.

Para liberar el espacio asignado a este objeto, en C++ se usa:

```
delete nomtipoptr;
```

que automáticamente invoca al destructor de la clase.

C++ permite incluir un **inicializador** para el objeto recién creado:

```
float *thingptr = new float(3.14159);
```

También se puede crear un arreglo y asignarlo a *tableroptr como sigue:

```
tableroptr = new int [8][8];
```

que sería borrado mediante el enunciado: **delete[ ] tableroptr**;

Ejemplo: De uso del operador new

```
int *pi = new int;              // asigna un entero de 2 bytes.
char *pc = new char[80];        // asigna un vector de 80
                                // caracteres
float *p = new float(10.4);     // asigna e inicializa *p == 10.4
```

La acción del operador **new** es similar, aunque superior, a la de la función malloc. Una sentencia **float** *p = **new float**, sería equivalente a:

```
float *p = (float *) malloc(sizeof(float));
```

. Diferencias con malloc

- Ya que **new** es un operador, no necesita un prototipo de función como malloc.

- El operador **new** calcula automáticamente el tamaño del tipo de dato que se está gestionando.

- **new** devuelve automáticamente el tipo correcto de puntero de acuerdo con el tipo de dato asignado. No hace falta hacer un "casting" del puntero que direcciona la memoria del heap.

- Con **new** es posible inicializar el objeto, cuya memoria se está asignando, mediante una expresión entre paréntesis.

```
int *pt = new int (3);          // *pt == 3
```

  Esta inicialización no es aplicable a objetos de tipo array. Los array de clases con varios constructores usan en la inicialización el "constructor por defecto".

```
class X
{
public:
        X( ) {/*...*/}
        X(int i) {/*...*/}
};
void main()
        X *p = new X[10];
        // asigna 10 objetos de la clase X
        // usando el constructor X :: X( )
}
```

- Si el <tipo-objeto> es una array, el <objeto-puntero> de **new** apunta al primer elemento del array. En el caso de que el array sea multidimensional hay que consignar todas las dimensiones, que deben ser expresiones constantes con valor positivo.

```
int *pt1, *pt2;
pt1 = new int [5] [10] [5];     // correcto
pt2 = new int [5] [] [15];      // error
```

VI.4.2 Operador delete

Su sintaxis es:

delete <objeto-puntero>

ó

delete [] <objeto-puntero>; // caso de arrays

La memoria asignada desde el heap por el operador **new** es liberada por el operador **delete**, que actúa sobre el <objeto-puntero> usado en **new**. El resultado tiene tipo **void**.

```
int *pt = new int (4);          // *pt == 4
delete pt;
// libera la memoria gestionada por new para
// ubicar un entero.
```

- Cuando se libera la memoria asignada dinámicamente mediante **new** para un array de objetos de clases, se debe especificar al operador **delete** que se trata de un array usando el delimitador [ ], ya que al aplicar este operador se ejecuta para cada objeto la función miembro destructor.

```
char *pc = new char[20];    // asigna un vector de 20 caracteres
class X {/*...*/};
X *px = new X[10];          // asigna un vector de 10 objetos de X
delete pc;                  // libera la memoria asignada al vector
                            // de caracteres también sería válido
                            // delete[ ] pc;
delete[ ] px;   // libera la memoria asignada al vector de la clase X
```

En resumen, cuando se libera un array de objetos de clases mediante el operador **delete**, es necesario especificar, antes del <objeto-puntero>, el delimitador [ ]. Si el array es de algún tipo estándar predefinido en C++ o de clases sin función destructor, se puede especificar el delimitador [ ], aunque en Borland C++ no es necesario.

VI.5 Miembros de clase estáticos

Cada objeto de una clase, por lo general, tiene su propia copia local de los datos miembros de la clase. Esto resulta bastante razonable ya que el valor particular que tienen estas variables de instancia, determina el estado de cada objeto.

Por otro lado, un miembro **estático** representa información que es aplicable a toda la clase; es un miembro de la clase que es compartido por todos los objetos de la clase. Su declaración se precede, dentro de la definición de clase, de la palabra reservada **static** (especificador de almacenamiento). En realidad, un miembro estático no forma parte de los objetos de la clase.

Se pueden declarar estáticos los datos miembro y las funciones miembro. La declaración de un dato miembro estático en el cuerpo de una clase no es una definición,

por lo que su uso exige una definición posterior. Cuando un miembro se define como estático sólo existe una copia de él para todos los objetos de la clase, por lo que puede ser accedido (el miembro estático) sin referenciar un objeto de la clase.

Si cualquier objeto altera el contenido de un miembro estático, este cambio queda reflejado para todos los objetos.

Los miembros estáticos obedecen a las misma reglas de ocultamiento de datos, cuando son declarados **public**, **private** o **protected**, que en cualquier otro miembro de la clase.

Cabe señalar que la inicialización está permitida en miembros estáticos de clases globales, mientras que los miembros estáticos de clases locales a una función no admiten inicialización.

Al no estar ligadas las funciones miembro estáticas a objetos de la clase, no disponen del puntero **this**. Por ello, no se puede acceder desde estas funciones a los miembros no estáticos, a no ser que se especifique explícitamente este acceso con un objeto de la clase.

Las funciones miembro estáticas no pueden ser declaradas virtuales porque es ilegal tener una función "static" y "no static" con el mismo nombre y tipo de argumentos.

Los miembros estáticos de las clases globales tienen enlazado externo mientras que los correspondientes a clases locales no tienen enlazado y no pueden ser definidos fuera de su declaración de clase, por lo que una clase local no puede tener miembros datos estáticos.

En resumen:

- Aunque parecen ser variables globales, **los datos miembros estáticos tienen alcance de clase**.

- Los datos miembros estáticos pueden ser públicos, privados o protegidos.

- Los datos miembros estáticos deben ser inicializados en alcance de archivo. La inicialización está permitida en miembros estáticos de clases globales, pero los miembros estáticos de clases locales a una función no admiten inicialización. Esto se debe a que los miembros estáticos de las clases globales tienen enlazado externo mientras que los correspondientes a clases locales no tienen enlazado y no pueden ser definidos fuera de la declaración de su clase, razón por la que una clase local no puede tener miembros datos estáticos.

- Cuando un miembro se define como estático sólo existe una copia de él para todos los objetos de la clase, por lo que puede ser accedido sin referenciar un objeto de la clase. Si cualquier objeto altera el contenido de un dato miembro estático, esta modificación queda reflejada para todos los objetos de la clase.

- Los datos miembros estáticos públicos son accesibles a través de cualquier objeto de la clase o a través del nombre de la clase, usando el operador de resolución de alcance binario:

  nombreclase :: miembrodato.

- Para acceder a los datos miembros privados o protegidos se debe usar las funciones miembros públicas de la clase. Por ejemplo, si static int getcount( ); es una función miembro estática pública de la clase empleado, entonces:

  empleado **::** getcount( );

- Los miembros de clase estáticos **existen** aún cuando no existan objetos de dicha clase.

- Una **función miembro puede ser declarada static si no tiene acceso a miembros de clase no estáticos**. Esto se debe a que, una función miembro estática no tiene puntero this ya que los datos miembros estáticos y las funciones miembros estáticas existen independientemente de cualquier objeto de la clase.

- Las funciones miembros estáticas no pueden ser declaradas **virtual** porque es ilegal tener una función static y no static con el mismo nombre y tipo de argumentos.

VII. Polimorfismo y sobrecarga

Polimorfismo es la calidad o estado de un elemento de asumir diferentes formas.

Desde el punto de vista de la Informática:

> **Polimorfismo es el proceso por el cual se puede acceder a diferentes implementaciones de una función u operador usando el mismo nombre.**

El polimorfismo configura una sobrecarga de una función u operador, a los que se accede mediante un cambio de forma que toma un elemento.

C++ soporta polimorfismo en tiempo de compilación y en tiempo de ejecución:

a) En **tiempo de compilación** permite la **sobrecarga de funciones** (normales, miembros de clases y amigas) **y** de **operadores** (unarios y binarios). Se trata de clases y funciones genéricas ("templates").

b) En **tiempo de ejecución**, la sobrecarga, ahora llamada **polimorfismo** se realiza por medio de un enlace dinámico del objeto con su función específica sobrecargada mediante la definición de **funciones virtuales** que se describen en el capítulo de **Herencia**.

VII.1 Sobrecarga de funciones

Consiste en asignar un significado adicional al nombre de la función, de forma que la función es referenciada de diferentes modos (ya que tienen distintas firmas) y proporciona diferentes ejecuciones (ya que hay distintas implementaciones, una para cada sobrecarga).

Existen tres tipos de funciones sobrecargadas:

a) Funciones normales (Funciones C++)

```
void print (char i) {cout << i;}
void print (float I) {cout << i;}
void main( ) {
    print ('a');      //referencia a print(char)
    print(3.14f);     //referencia a print(float).
};
```

Las funciones sobrecargadas pueden tener el mismo nombre pero debe tener distintos prototipos. El sistema selecciona, en tiempo de compilación, la función apropiadas por las características de sus argumentos (número, tipo o ambos).

b) Funciones miembros de clases

```
class X
{
public:
        void print(char);
        void print (float);
};
void print(char);
void main( )
{
        X a;            //crea un objeto a de la clase X
        a.print('a');
        a.print(3.14f);
        print('a')
};
```

En éste caso el sistema referencia y selecciona una de las funciones miembro asociadas a ese objeto. Las funciones miembros sobrecargadas deben cumplir, entre ellas, con los mismos requisitos señalados anteriormente para funciones normales. Sin embargo, una función normal y una función miembro pueden tener el mismo prototipo ya que la forma de referenciarlas es diferente: Para el caso de una función miembros, se referencia usando un objeto ej.: a.print('a'); y una función normal: ej. print('a');.
El tipo más común de función miembro sobrecargada es el constructor de la clase.

c) Funciones amigas

```
class X
{
        friend f(X);
        friend f(X,int);
        friend Y :: f(X);       //f es miembro de la clase Y
};
void main( )
{
        X a;
        Y b;
        f(a);           //referencia a f(X)
        b.f(a)          //referencia a Y::f(X)
        f(a,10) ;       //referencia a f(X,int)
};
```

Las funciones amigas presentan la problemática conjunta de las funciones normales y de las funciones miembros. Es decir, las funciones amigas que no son miembros de otras clases se comportan en la sobrecarga como funciones normales y las que son miembros se comportan como las funciones miembros sobrecargadas.

VII.2 Operadores sobrecargados

La programación en C++ es un proceso sensible a los tipos y enfocado a los tipos.

Los programadores pueden usar tipos de datos incorporados y pueden definir nuevos tipos de datos. Los tipos incorporados pueden ser usados aprovechando la extensa colección de operadores de C++, que proporcionan una notación concisa para expresar manipulaciones de objetos.

Aunque C++ no permite que se creen operadores nuevos, al permitir la **homonimia** de operadores, permite que los operadores existentes sean usados con objetos de clases, con un significado apropiado a los nuevos tipos.

La **hominimia de operadores contribuye a la extensibilidad de C++**.

La homonimia de operadores se usa, de manera inadvertida, regularmente: Ej. El operador + suma enteros, flota, double, etc.

- La sintaxis de la declaración de un operador es:

  <tipo-retorno> **operator** <símbolo-operador>(<parámetros>)

  donde:
  <tipo-retorno> y <parámetros> se ajustan a los especificados para funciones.
  <símbolo-operador> es uno de los símbolos de la tabla.(Pag. siguiente).

- Para definir un **operador homónimo** se escribe la definición de la función, con un encabezado y cuerpo, como se hacer normalmente, pero el nombre de la función se reeemplaza por la palabra **operator** seguida del **símbolo** correspondiente al operador homónimo. (Ejemplo: operator +).

- Para poder usar un operador sobre objetos de clase, dicho operador deberá ser un homónimo, con dos excepciones:

  - **El operador de asignación (=)**: Puede ser usado sin homónimo en cualquier clase. Cuando no existe el homónimo, su comportamiento por omisión es hacer una copia miembro a miembro de los datos miembros de la clase. Esto puede ser peligroso cuando se trata de clases con miembros que apunten a almacenamiento asignado dinámicamente. En estos casos se hace la homonimia del operador de asignación.
  - **El operador de dirección(&)**: Que sólo devuelve la dirección del objeto en memoria. También se puede hacer la homonimia de este operador.

- El propósito de la homonimia de operadores es proporcionar las mismas expresiones concisas para tipos definidos por el usuario que las que C++ proporciona en relación a los tipos incorporados.

- La homonimia de operadores **no es automática** ya que el programador deberá escribir las funciones de homonimia de operadores ya sea como funciones miembros, como funciones amigas o como ninguna de las dos.

VII.2.1 Restricciones sobre la homonimia de operadores

- **Operadores que pueden tener homónimos**

+	-	*	/	%	∧	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	∧=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	→*	,	→	[ ]	( )	new	delete

- **Operadores que NO pueden tener homónimos**

.	**.***	**::**	**?:**	#	# #	**sizeof**

- La **precedencia de operadores no puede ser modificada** por la homonimia, aunque sí se pueden usar paréntesis dentro de las expresión para obligar el orden de evaluación de los operadores homónimos.
- La **asociatividad** de un operador tampoco puede ser modificada para la homonimia.
- Para hacer la homonimia de un operador **tampoco pueden usarse argumentos por omisión**.

- **No es posible modificar el número de operados que toma un operador**. Los operadores unarios homónimos son unarios y los operadores binarios homónimos son binarios.
- **No es posible crear operadores nuevos**.

- El **significado** de un operador sobre objetos de tipos incorporados no puede ser modificado por la homonimia.
- Para hacer la homonimia de los operadores **( )**, **[ ]**, **→** ó **=**, la función de homonimia de operadores debnerá ser declarada como miembro de la clase. Para los demás operadores, puede ser una función friend o amiga.

VII.2.2 Compración entre funciones operador como miembros de clase o amigos

Las funciones operador pueden ser:

- Funciones miembros
- Funciones no miembros (generalmente, funciones friend)

- Las **funciones miembro** usan el puntero this, en forma implícita, para obtener uno de sus argumentos, el que es objeto de la clase.
  El operando de más a la izquierda (o único) deberá ser un objeto de la clase (o referencia a un objeto de la clase) del operador. Las funciones miembros operador son llamadas solamente cuando el operando izquierdo de un operador binario, o el único operando de un operador unario, es de forma específica un objeto de dicha clase.

- En el caso de las funciones no miembro, el argumento de clase debe estar listado explícitamente.
  Si el operando izquierdo tiene que ser un objeto de una clase diferente o de tipo incporporado, la función operador deberá ser implementada como función no miembro.
  Si, además, la función operador debe tener acceso directo a miembros privados o protegidos de una clase, necesita ser implementada como un amigo de dicha clase.
  Más aún, otra razón para elegir una función no miembro para hacer la homonimia de un operador, es permitir que el operador sea conmutativo, ya que en un caso el operando es miembro de la clase y en otro no.

VII.2.3 Hominimia de los operadores de inserción y extracción de flujo

Para procesar cada tipo de datos estándar, incluyendo cadenas y direcciones, y para ejecutar entradas y salidas para tipos definidos por el usuario, se hace la homonimia del **operador de inserción de flujo (<<)** y del **operador de extracción de flujo (>>)**.

Ejemplo: Para probar la homonimia de operadores de inserción y extracción de flujo.

```
//PRUEBA.CPP
//Sobrecarga de operadores >> y <<
#include <iostream.h>
class numtele {
        friend ostream &operator << (ostream&,const numtele&);
        friend istream &operator >> (istream&, numtele&);
private:
        char codarea[4];      //codigo de area de 3 dígitos o NULL
        char exchange[4];     //intercambio de 3 digitos o NULL
        char line[5];};
```

```
//Operador de inserción sobrecargado
//No es funcion miembro
ostream &operator << (ostream &output, const numtele &num)
{
        output << "(" << num.codarea << ")" << num.exchange
                        << "-" << num.line;
        return output;          //posibilita cout << a << b << c.
}
//Operador de extracción sobrecargado
//No es una función miembro
istream &operator >> (istream &input, numtele &num)
{
        input.ignore( );                    //saltea el (
        input.getline(num.codarea,4);       //obtiene el codigo de area
        input.ignore(2);                    //saltea el ) y el blanco
        input.getline(num.exchange,4);      //obtiene el intercambio
        input.ignore( );                    //saltea el guión (-)
        input.getline(num.line,5);          //obtiene la linea
        return input;                       //Habilita cin >> a >> b >> c
}
main( )
{
        numtele telefono;               //crea un objeto numtele
        cout << "Ingrese un numero de telefono de la forma "
                << "      (123) 456-7890:\n";

        //cin >> telefono invoca a la funcion operator >>
        // emitiendo la llamada operator >> (cin,telefono)

        cin >> telefono;

        //cout << telefono invoca a la funcion operator <<
        //emitiendo la llamada operator << (cout,telefono)

        cout << "El numero de telefono ingresado fue: \n"
                << telefono << endl;
        return 0;
}
```

VII.3 Tipos de operadores sobrecargados

En este apartado se describe la forma de sobrecargar un operador atendiendo a su estructura.

a) **Operadores Unarios:** Se puede hacer la homonimia de un operador unario para una clase como una **función miembro no estática, sin argumentos** o como una **función no miembro con un argumento**, que obligadamente debe ser una objeto de la clase o referencia a un objeto de la clase. En el primer caso se sobrecarga un operador unario prefijo y en segundo, un operador unario postfijo.

Si ***pi*** representa un operador unario y ***a*** un objeto, las operaciones ***pia*** y ***api*** se pueden interpretar como ***a.operator pi( )*** u ***operator pi(a)***, dependiendo de la declaración hecha. Si ambas formas han sido declaradas, se aplica una equiparación estándar de sus argumentos para resolver la ambigüedad.
Es posible distinguir la utilización prefija o postfija de los operadores unarios sobrecargados como ++ y -- porque la sobrecarga se realiza en forma diferente.

Ejemplo con ! (negación lógica)

```
i)    class string {                    //Si s es un objeto de la
      public:                           //clase, cuando el
            int operator !( ) const;    //compilador vea !s
            …                           //generará la llamada
      };                                //s.operator(

iii)  class string                      //Si s es un objeto de la
      {                                 //clase, !s se tratará como
         friend operator !(const string&);//si se hubiera escrito
          …                             //operator !(s)
      };
```

b) Operadores Binarios: Los operadores binarios se pueden sobrecargar con funciones miembros no estáticas que lleven un argumento o con funciones no miembros (lo normal es que sean amigas) con dos argumentos, uno de los cuales debe ser un objeto de la clase o referencia a un objeto de la clase.

Si ***pi*** representa un operador binario, y ***a*** un objeto, entonces ***apib*** se puede interpretar como ***a.operator pi(b)*** u ***operator pi(a,b)*** dependiendo de que la declaración efectuada sea una función miembro o no miembro. Si ambas formas han sido declaradas, se aplica la equiparación de sus argumentos para resolver la ambigüedad.

Ejemplo con +=(Asignación con suma)

```
i)    class string                      //Si a y b son objetos de
      {                                 //la clase, cuando el
      public:                           //compìlador vea
            string &operator +=(cons string&);      //a += b
            …                           //generará la llamada
      };                                //a.operator +=(b);

ii)   class string                      //En este caso, a += b
      {                                 //será tratado como si
      friend string &operator +=(string&, const string&);
      …                                 //se hubiera escrito la
      };                                //llamada operator +=(a,b)
```

VII.4 Conversión entre tipos

A menudo es necesario convertir datos de un tipo a otro tipo, por ejemplo, en asignaciones, cálculos, pasaje de parámetros, etc.

El compilador puede llevar a cabo conversiones entre tipos incorporados y los programadores pueden obligar a estas conversiones mediante **conversiones explícitas** llamadas **casting**.

En el caso de tipos definidos por el usuario, el programador deberá especificar en forma explícita como deberán ocurrir las conversiones. Estas serán ejecutadas mediante **constructores de conversión** (constructores con un único argumento que solamente convierten objetos de otros tipos en objetos de una clase).

Se puede usar el **constructor de conversión explícita** (**cast**) para convertir un objeto de una clase en un objeto de otra clase o en un objeto de un tipo incorporado. Este operador debe ser una función miembro no estática; no puede ser una función amiga.

La siguiente función, que se observa que no especifica tipo de retorno puesto que este es el tipo al que se está convirtiendo el objeto:

```
operator char*( ) const;
```

declara una función operator de conversión explícita cast homónima para crear un objeto temporal char* partiendo de un objeto definido por el usuario.

Cuando el compilador vea la expresión **(char*)s** generará la llamada **s.operator char*( )**. El operando s es el objeto de la clase para la cual se invoca la función miembro operator char*.

Así ocurre que:

operator int( ) const;	Es para convertir un objeto de tipo definido por el usuario a **entero**.
operator otraclase( ) const;	Es para convertir un objeto de tipo definido por el usuario a otro tipo definido por el usuario: otraclase.

Una característica notable de los operadores ***cast*** y de los ***constructores de conversión*** es que, cuando es necesario, el compilador puede llamar a estas funciones de forma automática para crear objetos temporales.

Si en un programa aparece un objeto s de la clase ***string***, definida por el usuario, en una posición donde normalmente se espera char*, como:

```
cout << s
```

el compilador llamará a la función cast homónima, operator char*, para convertir el objeto en un char* y usar el char* resultante en la expresión.

VII.5 Homonimia de ++ y –

Los operadores de incremento y decremento, pre y post, pueden tener homónimos.

El problema que se presenta es que el compilador deberá poder distinguir cuál versión de cada operador, prefija o postfija, se desea usar. Cada función deberá tener una forma distinta.

Suponga que en el objeto ***d1*** de la clase date queremos añadir 1 al día.

Cuando el compilador vea la expresdión preincremental:

++d1

- Si fue implementada como una **función miembro**, el compilador generará la llamada:
  d1.operator ++( );
  cuyo prototipo sería:
  date operator ++( );
- Si fue implementada como una **función no miembro**, generará la llamada:
  operator ++(d1);
  cuyo prototipo deberá ser declarado en la clase date como:
  friend date operator ++(date&);

Para hacer la homonimia del **operador de postincremento**, el problema que se presenta es que el compilador deberá ser capaz de distinguir las firmas de las funciones de operador preincremento y postincremento homónimas.

Por convención de C++, cuando el compilador vea la expresión:

d1++

- Si el postincremento fue implementado como una función miembro, el compilador generará la llamada:
  d1.operator++(0);
  cuyo prototipo sería:
  date operator ++(int);
  Se usa 0 como **parámetro de relleno** para hacerlo distinguible del preincremento.
- Si está implementada como una función no miembro, generará la llamada:
  operator ++(d1,0);
  cuyo prototpo deberá ser declarado en la clase date como:
  friend date operator ++(date&,int);

VIII. Herencia

> **Herencia es el proceso por el cual un objeto adquiere las propiedades de otro u otros objetos que constituyen sus predecesores jerárquicos.**

Este concepto, fundamental en la POO, nace de la necesidad de **reutilizar** el código existente. La herencia es una forma de **reutilización de código** en la que se crean clases nuevas a partir de clases existentes, mediante la absorción de sus atributos y comportamientos e incorporándoles las capacidades que las clases nuevas requieran.

La reutilización de software ahorra tiempo en el desarrollo de programas, fomenta la reutilización de software de alta calidad, probado y depurado, y reduce los problemas en un sistema después de convertirlo a funcional.

La reutilización de código puede realizarse, en C++, de dos maneras diferentes:

a) Por composición: En este caso, un objeto se configura como una colección o agregación de otros objetos. Dentro de una clase se incorporan otras clases a sus miembros (**anidamiento de clases**). Por ejemplo, un edificio está compuesto por puertas, ventanas, habitaciones, etc.
En C++ se puede reutilizar código "por composición" declarando un objeto, o un puntero a un objeto, de una clase para que sea miembro de la nueva clase.
En el caso de tener un objeto de una clase como miembro de otra, hay que tener en cuenta que el constructor de la nueva clase tiene que incorporar, de alguna manera, al constructor del objeto de la clase miembro. La composición de objetos tiene lugar en tiempo de compilación.
En el caso en que se usen punteros a objetos de clase como miembros de otras clases, se puede decidir, en tiempo de ejecución, el tipo de objeto que se agregará para formar el objeto de la clase compuesta.

b) Por herencia o derivación: Esta forma de reutilización consiste en que una clase hereda los datos y métodos de sus clases base. Este concepto de herencia conforma el concepto básico en POO, de **jerarquía de clases**.
Por ejemplo, un rascacielos (objeto derivado) es un edificio (objeto base) con muchos pisos. Un cuadrado deriva de un paralelogramo, y éste de polígono.

El **polimorfismo** permite escribir programas de forma general, con el fin de procesar una amplia variedad de clases existentes o sin especificar.
La herencia y el polimorfismo son técnicas efectivas para enfrentar la complejidad del software.

Al crear una nueva clase, el programador puede determinar que la clase nueva debe heredar los datos y funciones miembros de la **clase base**, en lugar de escribirlos en la **clase derivada**. La clase derivada es un candidato a ser una clase base de una clase derivada futura.

Existen dos tipos de herencia:

- **Herencia (simple)** Una clase es derivada de una única clase base.
- **Herencia múltiple** Una clase derivada hereda de varias clases base que pueden no estar relacionadas.

En general, una clase derivada es **más grande** que su clase base ya que añade miembros datos y funciones miembros propios, pero representa un **conjunto más pequeño de objetos** ya que es más específica que la clase base.

La verdadera fuerza de la herencia proviene de **la capacidad de definir una clase derivada, adiciones, reemplazos o refinamientos de las características heredadas de la clase base**.

- Un objeto de clase derivada también es un objeto de clase base (pero no al revés). Esto permite, entre otras cosas, unir una gran variedad de objetos distintos, relacionados mediante herencia, en una lista enlazada de objetos de clase base.
- Distinguiremos dos tipos de relaciones:

  "**es un**" Es una relación de **herencia**, donde un objeto de clase derivada también puede ser tratado como un objeto de clase base.

  "**tiene un**" Es una **composición**. Un objeto de una clase tiene como miembros uno o más objetos de otras clases.
- Una clase derivada **no tiene acceso** a los miembros privados de su clase base, ya que esto violaría el encapsulado de la clase base. Pero si tiene acceso a los miembros públicos y protegidos de su clase base.
- Una clase derivada puede heredar **funciones miembros públicas** de la clase base, aunque no las requiera. Este problema se soluciona **redefiniendo** es función en la clase derivada con una implementación adecuada.
- Las clases nuevas pueden heredar **bibliotecas de clases** existentes. La perspectiva es que el software será elaborado a partir de **CIS** (**Componentes Integrados de Software**).
- C++ **no admite la herencia de constructores, destructores ni el operador sobrecargado =**. En estos casos se aplica el propio de la clase, ya sea definido por el usuario o proporcionado por el sistema.

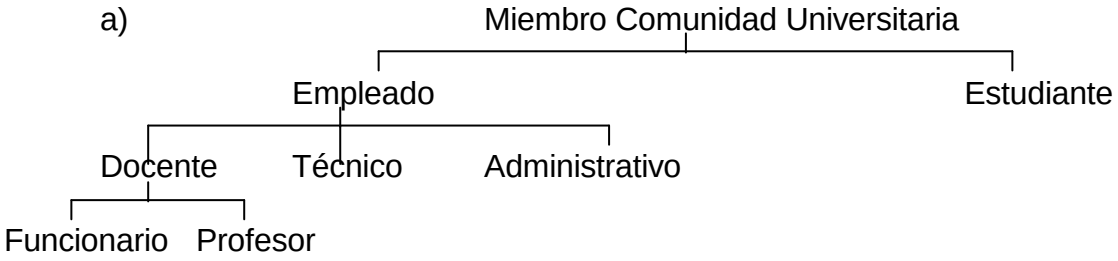
VIII.1 Clases base y clases derivadas

La herencia forma **estructuras jerárquicas de apariencia arborescente**.

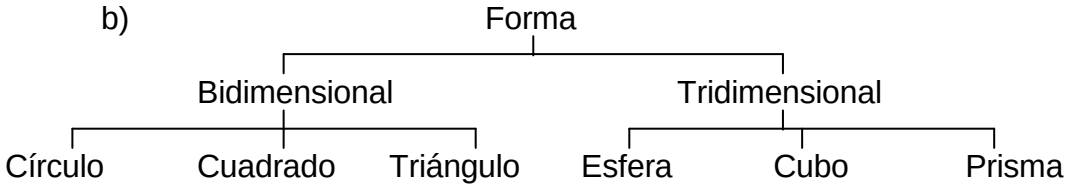
Una clase base existe en una relación jerárquica con sus clases derivadas. Es obvio que una clase puede existir por sí misma. Pero cuando es usada mediante el mecanismo de herencia, se convierte en una clase base proveyendo atributos y comportamientos a otras clases o en una clase derivada que hereda de otras clases esos atributos y comportamientos.

Algunos ejemplos de jerarquía simple de herencia son:

a)



b)



La herencia puede ser:

- **Public** Los miembros públicos y protegidos de la clase base son heredados como miembros públicos y protegidos, respectivamente, en la clase derivada:

```
class empleadoacomision : public empleado {
...
};
```

- **Private** Los miembros públicos y protegidos de la clase base son heredados como miembros privados en la clase derivada:

```
class X : private Y {
...
};
```

- **Protected** Los miembros públicos y protegidos de la clase base son heredados como miembros protegidos en la clase derivada:

```
class Z : protected T {
...
};
```

Los miembros privados de una clase base, en ningún caso, so accesibles en forma directa desde una clase derivada.

VIII.1.1 Miembros protegidos

- Los **miembros públicos** de una clase base son accesibles a todas las funciones del programa.
- Los **miembros privados** de una clase base son accesibles sólo a funciones miembros y amigos de la clase, aunque nunca de forma directa.
- Los **miembros protegidos** de una clase base son accesibles sólo por funciones miembros y amigos de la clase base y por funciones miembros y amigos de las clases derivadas. (Representa un nivel intermedio de protección).

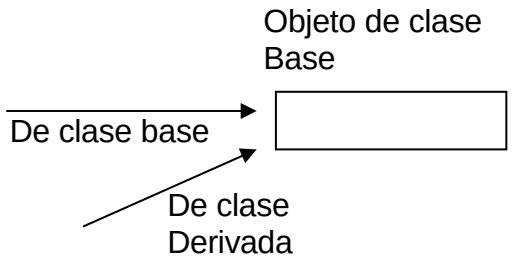
Cuando un miembro de una clase derivada hace referencia a un miembro público o protegido de su clase base, usa su nombre (no necesita el operador de resolución de alcance). Se supone entonces, el objeto actual.

VIII.2 Cast de punteros de clase base a punteros de clase derivada

Un objeto de clase derivada pública puede ser tratado como un objeto de clase base. Esto permite, por ejemplo, crear una lista enlazada de una gran variedad de objetos diferentes de clases derivadas distintas, siempre y cuando los tratemos como objetos de la clase base.

La interpretación en el sentido opuesto no es válida ya que un objeto de clase base no es, automáticamente, un objeto de clase derivada.

Sin embargo, se puede usar un ***cast*** para convertir un puntero de clase base a puntero de clase derivada. Hay que tener en cuenta que si un puntero es desreferenciado, primero se deberá hacer que señale a un objeto de clase derivada.



No se pueden referenciar miembros de la clase derivada que no existen en la clase base.

Ejemplo: Para mostrar el casting de punteros de clase base a punteros de clase derivada y viceversa.

```
//POINT.H
//Declaración de la clase POINT
#ifndef POINT_H
#define POINT_H

class point {
        friend ostream &operator << (ostream&,const point&);
public:
        point(float = 0.0,float = 0.0);      //Constructor default
        void setpoint(float,float);          //Fijar coordenadas
        float getX( ) const {return x;}      //get coordenada x
        float getY( ) const {return y; }     //get coordenada y
protected:
        float x,y;                           //x e y coordenadas de point
                                             //accesibles solo por clase derivadas
};
#endif
//POINT.CPP
//Definicion de funciones miembros de la clase POINT
#include <iostream.h>
#include "point.h"
//Constructor de la clase point
point :: point(float a, float b)
{
        x = a;
        y = b;
}
//Fijar coordenadas de point
void point :: setpoint(float a, float b)
{
        x = a;
        y = b;
}
//Imprimir point con operador sobrecargado
ostream &operator << (ostream &output, const point &p)
{
        output << "[" << p.x << "," << p.y << "]";
        return output;                  //Habilita llamadas concatendas
}
//CIRCLE.H
//Declaracion de la clase circle
#ifndef CIRCLE_H
#define CIRCLE_H

# include <iostream.h>
#include <iomanip.h>
#include "point.h"
```

```
class circle : public point {
//Circle hereda de point en forma publica
        friend ostream &operator << (ostream&, const circle&);
public:
        circle(float r = 0.0, float x = 0.0, float y = 0.0);     //constructor default
        void setradio(float);                                    //fija el radio
        void getradio( ) const;                                  //get del radio
        float area( ) const;                                     //calcula el area
protected:
        float radio;
};
#endif
//CIRCLE.CPP
//Defincion de las funciones miembros de circle
#include "circle.h"
//El constructor de circle llama al constructor de point con un inicializador de
//miembro, despues inicializa el radio
circle :: circle(float r, float a, float b)
        : point(a,b)            // llamada al constructor de point
{radio = r; }
//fijar el radio del circulo
void circle :: setradio(float r) {radio = r; }
//obtener el radio
float circle :: getradio( ) const {return radio; }
//calcular el area del circulo
float circle :: area( ) const
{return 3.14159 * radio * radio;}
//Imprimir circulo en el formato centro = [x,y]; radio = #.##
ostream &operator << (ostream &output, const circle &c)
{
        output << "centro = [" << c.x << "," << c.y << "]; Radio = "
                << setiosflag(ios:showpoit) << setprecision(2) << c.radio;
        return output;          //habilita llamadas concatenadas
}

//PRUEBA.CPP
//Prueba de cast de punteros de clase base a punteros de clase derivada
#include <iostream.h>
#include <iomanip.h>
#include "point.h"
#include "circle.h"
main( )
{
        point *pointptr, p(3.5, 5.3);
        circle *circleptr, c(2.7,1.2,8.9);

        cout << "Punto p: " << p << "\n Circulo c: " << c << endl;
```

```
        //Tratar un circulo como un punto (con algun casting)
        pointptr = &c;                  //asignarle al puntero pointptr la
                                        //dirección de un circle
        circleptr = (circle*)poitptr;   //cast de clase base a clase derivada

        cout << "\nArea de c (via circleptr): "
             << circleptr → area( ) << endl;

        //PELIGROSO: Tratar un punto como un circulo
        pointptr = &p;              //asignarle al puntero pointptr la
                                    //dirección de un punto
        circleptr = (circle*) pointptr; //cast de clase base a clase derivada

        cout << "\nRadio del objeto circleptr apunta a: "
             << circleptr → getradio( ) << endl;
        return 0;
}
```

La salida sería:

```
Punto p: [3.5,5.3]
Circulo c:[1.2,8.9]; Radio = 2.70

Area de c (via circleptr): 22.9

Radio del objeto circleptr apunta a: 4.02e-38
```

VIII.3 Cómo usar funciones miembros

- A veces es necesario que las funciones miembros de una clase derivada contengan ciertos miembros de la clase base. Pero hay que tener en cuenta que una clase derivada no tiene acceso a los miembros privados de la clase base.

- Una clase derivada puede **redefinir** una función miembro de su clase base. Cuando, en la clase derivada, la función es mencionada por su nombre, de manera automática, se selecciona la versión de la clase derivada. Se puede acceder a la versión de la clase base usando el operador de resolución de alcance **::**.

VIII.3.1 Clases base públicas, protegidas y privadas

- Al derivar una clase de una clase base, la clase base puede ser heredada por la clase derivada como public, private o protected.

- La herencia privada y la herencia protegida son raras y deben usarse cuidadosamente.

VIII.3.2 Clases base directas y clases base indirectas

- Una **clase base directa** de una clase derivada es listada en forma explícita en el encabezado de la clase derivada, cuando ésta se define.

- Una **clase base indirecta** se hereda desde varios niveles arriba en la jerarquía de herencia de la clase. No se lista en forma explícita en el encabezado de la clase derivada.

VIII.4 Cómo usar constructores y destructores en clases derivadas

Ya que una clase derivada hereda los miembros de su clase base, cuando es producido un objeto de una clase derivada, **el constructor de la clase base deberá ser llamado para inicializar los miembros de la clase base del objeto de la clase derivada**.

Este llamado se puede hacer de dos maneras diferentes:

- El constructor de la clase derivada llama en **forma implícita** al constructor de la clase base.

- Se provee al constructor de la clase derivada de un **inicializador de clase base**.

Los constructores y los operadores de asignación de la clase base no son heredados por la clase derivada, pero los constructores y los operadores de asignación de la clase derivada pueden llamarlos.

- Un **constructor** de clase derivada siempre **llamará primero** al constructor correspondiente de su clase base para inicializar los miembros de la clase base en la clase derivada.

- Los **destructores**, en general, son llamados en orden inverso a las llamadas a constructor, por lo que el destructor de la clase derivada será llamado antes que el destructor de la clase base.

- El estándar de C++ establece que **el orden en que son construidos los objetos miembros es el orden en el que han sido declarados en la definición de la clase**, sin que afecte a la construcción el orden en que se listaron en los inicializadores de miembros.

- En la herencia, los constructores de clase base son llamados en el orden en el que se ha especificado la herencia en la definición de la clase derivada. El orden en que se hayan especificado los constructores de clase base en el constructor de la clase derivada no afecta la construcción.

Ejemplo: Para mostrar cuando son llamados los constructores y destructores de clase base y clase derivada.

```
//POINT2.H
//Declaración de la clase POINT
#ifndef POINT2_H
#define POINT2_H

class point
{
public:
        point(float = 0.0,float = 0.0);     //Constructor default
        ~point( );                          //Destructor
protected:
        float x,y;                  //x e y coordenadas de point
                                    //accesibles solo por clase derivadas
};
#endif
//POINT2.CPP
//Definicion de funciones miembros de la clase POINT
#include <iostream.h>
#include "point2.h"
//Constructor de la clase point
point :: point(float a, float b)
{
    x = a;
    y = b;

    cout << "Constructor de point: " << '[' << x
        << "," << y << ']' << endl;
}
//Destructor de la clase point
point :: ~point( )
{
    cout << "Destructor de point: " << '[' << x
        << "," << y << ']' << endl;

}
//CIRCLE2.H
//Declaracion de la clase circle
#ifndef CIRCLE2_H
#define CIRCLE2_H

# include <iostream.h>
#include <iomanip.h>
#include "point2.h"
```

```
class circle : public point {
//Circle hereda de point en forma publica
public:
        circle(float r = 0.0, float x = 0.0, float y = 0.0);      //constructor default
        ~circle( );                                               //destructor
protected:
        float radio;
};
#endif
//CIRCLE2.CPP
//Defincion de las funciones miembros de circle
#include “circle2.h”
//El constructor de circle llama al constructor de point con un inicializador de
//miembro, despues inicializa el radio
circle :: circle(float r, float a, float b)
        : point(a,b)            // llamada al constructor de point
        {radio = r;
        cout << “Constructor de circle: radio es “ << radio
                << “ [ “ << a << “,” << b << “]” << endl;
}
//Destructor de la clase circle
circle :: ~circle( )
{
        cout << “Destructor de circle: radio es “ << radio
                << “ [ “ << a << “,” << b << “]” << endl;
 }

//PRUEBA.CPP
//Mostrar cuando son llamados los constructores y destructores
#include <iostream.h>
#include <iomanip.h>
#include “point2.h”
#include “circle2.h”
main( )
{
        {point  p(1.1, 2.2);}           //define un bloque y crea un objeto point
        cout << endl;

        circle c1(4.5,7.2,2.9);
        cout << endl;

        circle c2(10,5,5);
        cout << endl;
        return 0;
}
```

La salida sería:

```
Constructor de point: [1.1,2.2]
Destructor de point: [1.1,2.2]

Constructor de point: [7.2,2.9]
Constructor de circle: radio es 4.5 [7.2,2.9]
Constructor de point: [5,5]
Constructor de circle: radio es 10 [5,5]

Destructor de circle: radio es 10 [5,5]
Destructor de point: [5,5]
Destructor de circle: radio es 4.5 [7.2,2.9]
Destructor de point: [7.2,2.9]
```

VIII.5 Conversión implícita de objetos de clase derivada a objetos de clase Base

Un objeto de clase derivada también “es un” objeto de clase base, pero los **tipos** de ambos son **diferentes**.

Los objetos de clase derivada también pueden tratarse como objetos de clase base. Esta asignación tiene sentido porque la clase derivada tiene miembros que se corresponden con cada uno de los miembros de clase base, y algunos más.

En el otro sentido, la asignación no está permitida, ya que dejaría sin definición a los miembros adicionales de la clase derivada. Eventualmente, se podría haber proporcionado un operador de asignación y/o constructor de conversión correctamente homónimos.

Un apuntador a un objeto de clase derivada puede ser convertido, en forma implícita, a un puntero a un objeto de clase base (ya que un objeto de clase derivada es un objeto de clase base).

Existen cuatro formas de mezclar y hacer coincidir punteros de clase base y punteros de clase derivada con objetos de clase base y objetos de clase derivada:

- Haciendo referencia a un objeto de clase base con un puntero de clase base (simple y trivial).

- Haciendo referencia a un objeto de clase derivada con un puntero de clase derivada (simple y trivial).

- Haciendo referencia a un objeto de clase derivada con un puntero de clase base (ya que el objeto también es de clase base). Se debe tener cuidado de solamente referenciar miembros que estén en la clase base puesto que el puntero no referencia a los que no están y en caso contrario, el compilador informará un error de sintaxis.

- Salvo que el puntero de clase derivada sea convertido previa y explícitamente a puntero de clase base, es un error de sintaxis hacer referencia a un objeto de clase base mediante un apuntador de clase derivada.

De cualquier manera, tratar objeto de una clase derivada como si fueran de clase base y hacerlo manipulando estos objetos con punteros de clase base, presenta muchos problemas que se resolverían usando **funciones virtuales y polimorfismo**.

VIII.6 Composición en comparación con herencia

Las relaciones entre objetos que se manejarán serán:

"es un"	Relación de herencia.
"tiene un"	Composición
"usa un"	Una función usa un objeto al emitir una llamada a una función miembro de ese objeto. (Ej. Una persona usa un automóvil).
"conoce un"	Un objeto tiene una relación de conciencia con otro objeto cuando contiene un puntero o referencia a éste último. En ese caso, el objeto conoce o está consciente del otro objeto.

VIII.7 Herencia múltiple

Una clase puede ser derivada de más de una clase base. Es decir, hereda los miembros de varias clases base. Esta poderosa capacidad fomenta la reutilización de software pero puede causar **ambigüedad**. (Ej. Tipo A "es un" tipo B y "es un" tipo C).

Ejemplo: Para mostrar la relación de herencia múltiple

```
//BASE1.H
//Declaración de la clase BASE1

#ifndef BASE1_H
#define BASE1_H
```

```
class base1 {
public:
        base1(int x) {value = x;}
        int getdata( ) const  {return value; }
protected:
        int value;
};
#endif
```
//BASE2.H
//Declaracion de la clase base2
```
#ifndef BASE2_H
#define BASE2_H

class base2 {
public:
        base2(char c) {letra = c;}
        char getdata( ) const {return letra; }
protected:
        char letra;
};
#endif
```
//DERIVADA.H
//Declaración de clase derivada con herencia múltiple de base1 y base2
```
#ifndef DERIVADA_H
#define DERIVADA_H

#include <iostream.h>
#include “base1.h”
#include “base2.h”
```
//Herencia multiple
```
class derivada : public base1, public base2
{
        friend ostream &operator << (ostream&, const derivada&);
public:
        derivada(int,char,float);
        float getreal( ) const;
private:
        float real;
};
#endif
```
//DERIVADA.CPP
//Defincion de las funciones miembros de la clase derivada
```
#include “derivada.h”
```
//Constructor de la clase derivada
```
derivada :: derivada(int i, char c, float f)
                : base1(i), base2(c)
{real = f;}
```
//llama a ambos constructores

```
//Devuelve el valor real
float derivada :: getreal( ) const {return real;}

//Muestra todos los datos miembros de la derivada
ostream &operator << (ostream &output, const derivada &d)
{
        output << "       Entero: " << d.value
                << "\n Caracter: " << d.letra
                << "\n       Real: " << d.real;
        return output;//Habilita llamadas concatenadas
}
//PRUEBA.CPP
//Driver para el ejemplo de herencia multiple
#include <iostream.h>
#include "base1.h"
#include "base2.h"
#include "derivada.h"
main( )
{
        base1 b1(10), *base1ptr;          //crea objetos de clase base1
        base2 b2('Z'), *base2ptr;         //crea objetos de clase base2
        derivada d(7,'A',3.5);            //crea un objeto de clase derivada

        //Escribe los objetos de clase base
        cout << "Objeto b1 contiene entero " << b1.getdata( )
                << "\nObjeto b2 contiene carácter " << b2.getdata( )
                << "\nObjeto d contiene: \n" << d;

        //Escribe los datos del objeto de clase derivada
        //El operador de resolución de alcance resuelve la ambigüedad
        cout << "\nLos miembros dato de la derivada pueden ser"
                << " accedidos individualmente: \n"
                << "       Entero: " << d.base1 :: getdata( )
                << "\n Carácter: " << d.base2 :: getdata( )
                << "\n       Real: " << d.getreal( ) << "\n\n";

        cout << "Un objeto de la clase derivada puede ser tratado"
                << " como un objeto de cualquiera de las dos clases base: \n";

        //Tratar la derivada como un objeto de base1
        base1ptr = &d;
        cout << "base1ptr → getdata( ) devuelve: " << base1ptr → getdata( );

        //Tratar la derivada como un objeto de base2
        base2ptr = &d;
        cout << "base2ptr → getdata( ) devuelve: " << base2ptr → getdata( );

        return 0; }
```

La salida sería:

Objeto b1 contiene entero 10
Objeto b2 contiene carácter Z
Objeto d contiene
Entero: 7
Caracter: A
Real: 3.5

Los miembros dato de derivada pueden ser accedidos individualmente:
Entero: 7
Caracter: A
Real: 3.5

Un objeto de la clase derivada puede ser tratado como un objeto de cualquiera de las dos clases base:
base1ptr → getdata( ) devuelve 7
base2ptr → getdata( ) devuelve A

IX. Polimorfismo, funciones virtuales y enlace dinámico
IX.1 Ideas sobre el Polimorfismo

El término "**polimórfico**" proviene del griego y quiere decir "**de muchas formas**". En nuestro caso, el polimorfismo se puede asociar con funciones y objetos:

- Una **función polimórfica** tiene el mismo nombre para diferentes clases de la misma familia, pero tiene diferente implementación para las distintas clases.

- Un **objeto polimórfico** tiene el mismo nombre que otros objetos de una jerarquía de clases, de manera tal que, cada objeto, aunque se relaciona a través de una clase común, puede tener distinto comportamiento.

Así, el polimorfismo permite a las funciones y objetos del mismo nombre tener una comportamiento diferente dentro de una jerarquía de clases.

El polimorfismo le permite a los objetos ser más independientes, aún cuando sean miembros de la misma familia de clases. Más aún, se pueden **añadir** nuevas clases a la familia, sin cambiar las existentes. Esto permite que los sistemas se desarrollen con el tiempo, satisfaciendo las necesidades de una aplicación cambiante.

El polimorfismo se lleva a cabo usando **funciones sobrecargadas** o **funciones virtuales**. Las funciones sobrecargadas se **enlazan de manera estática** y las funciones virtuales se **enlazan dinámicamente**.

IX.2 Enlace dinámico versus enlace estático

> **Enlace se refiere al tiempo real cuando se anexa o une el código de una función con su llamada.**

Entonces:

> **El *enlace dinámico o tardío* ocurre cuando se define una función polimórfica para diversas clases en una familia, pero no se une ni se anexa el código real para la función hasta el momento de ejecución. Una función polimórfica que se une a su código en tiempo de ejecución se llama una *función virtual*.**

El enlace dinámico está instrumentado en C++ a través de las funciones virtuales. Con el enlace dinámico, cuando se llama a una función virtual, la selección del código correspondiente se retrasa hasta el momento de la ejecución.

Esto significa que cuando se llama a una función virtual, el código ejecutable determina, en el momento de la ejecución, a qué versión de la función se está llamando. Recuerde que las funciones virtuales son polimórficas, y por lo tanto, tienen diferentes implementaciones para distintas clases de la familia de clases.

El *enlace estático* ocurre cuando se define una función polimórfica para diversas clases en una familia y el código real para la función se anexa en el momento de la compilación. Las *funciones sobrecargadas* se unen de manera estática.

Dicho de otra manera, el enlace estático ocurre cuando el código de función es "ligado" en el momento de la compilación. Esto significa que cuando se llama a una función no virtual, sobrecargada, el compilador determina, al compilar, a qué versión de la función llamará.

En el caso de las **funciones sobrecargadas**, el compilador puede determinar a qué función llamará basándose en el número y tipo de los parámetros de la función. Por ello es necesario que todas las funciones sobrecargadas tengan distinta firma. Pero las **funciones virtuales tienen la misma interfaz** dentro de la familia de clases dada. Así, en el momento de ejecución se deben usar apuntadores de objetos para determinar a qué función llamar. (Recuerde que cada función puede ser rastreada a su clase correcta por medio de puntero ***this*** del objeto llamador).

Por fortuna, el enlace es realizado de manera automática por el link-editor.

Tenga presente que las funciones virtuales son más a menudo enunciadas en una clase base de C++, usando la palabra reservada **virtual**. Cuando se enuncia una función como virtual en una clase base, el compilador sabe que la definición de la función de clase base debe ser sobrescrita o redefinida en la clase derivada, dando una implementación diferente para la misma función. Si la definición de clase base no se sobrescribe en la clase derivada particular, entonces la definición de clase base está disponibles en la clase derivada.

En resumen, el uso de funciones virtuales y polimorfismo permite diseñar e implementar sistemas más fácilmente extensibles.

IX.3 Campos de bits y enunciados switch

Una forma alternativa de tratar objetos de muchos tipos diferentes es usando un enunciado **switch** a fin de tomar la acción apropiada sobre cada objeto basándose en su tipo. Pero el uso de la lógica switch implica algunos problemas:

- El programador puede olvidarse de efectuar la prueba de tipo correspondiente cuando sea requerida.

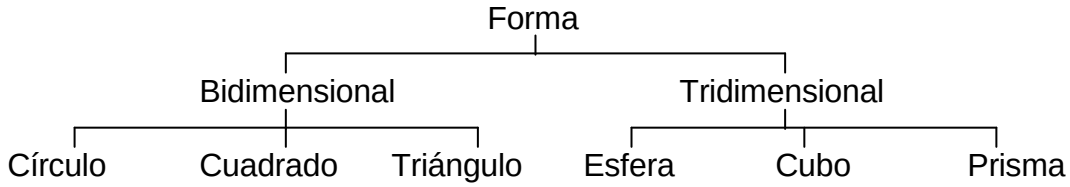
- El programador puede olvidarse alguno de los casos posibles.

- La modificación de enunciados switch es lenta y está sujeta a errores.

Las **funciones virtuales** y la **programación polimórfica** pueden eliminar la necesidad de la lógica switch.

IX.4 Funciones virtuales

En el siguiente esquema:



Supongamos que, en la P.O.O., cada una de estas clases está investida de la capacidad de dibujarse a sí misma con una función ***draw*** muy distinta en cada clase.

Sería interesante tener la capacidad de tratar todas estas formas en una manera genérica, como objeto de la clase forma. Llamaríamos a la función ***draw*** de la clase forma y el programa, dinámicamente, determinaría cuál de las funciones ***draw*** de las clases derivadas usar.

- En la clase base se declara ***draw*** como una **función virtual** y se la redefine en cada una de las clases derivadas. Una función virtual es una función miembro de una clase, declarada con el especificador **virtual** que será redefinida en una o en varias clases derivadas.

- Si en la clase base draw ha sido declarada **virtual**, usamos un puntero de clase base para indicar objetos de clase derivada, y usando ese puntero invocamos a la función ***draw***, el programa seleccionará de manera dinámica la función ***draw*** de la clase derivada. Este tipo de ligadura o enlace recibe el nombre de **ligadura dinámica**.

- A partir del punto en que se declara una función miembro como virtual, **se conserva como virtual** a lo largo de toda la jerarquía de herencia. Si una función está definida como virtual en la clase base no es necesario redefinirla virtual en las clases derivadas. Por el contrario, si se pone el calificador virtual en las clases derivadas, pero no en la clase base, se pierde el mecanismo “virtual” para la clase base, pero se conserva en las clases derivadas.

- Cuando una clase derivada decide no redefinir una función virtual, sólo heredará la función virtual de la clase base superior inmediata.
- Las funciones virtuales redefinidas **deben tener el mismo tipo de retorno y la misma firma** que la función virtual base.

- El especificador virtual se aplica a una **función miembro no estática** o a una **función amiga, miembro de otra clase**, pero en ningún caso se puede aplicar a una función no miembro.

- Si una función virtual es invocada haciendo referencia a un objeto por su nombre y usando el **operador de selección de miembro (.)**, se establece una **ligadura estática** (la referencia se resuelve en tiempo de compilación) y la función virtual llamada es aquella definida para la clase de dicho objeto (o heredada por ella).

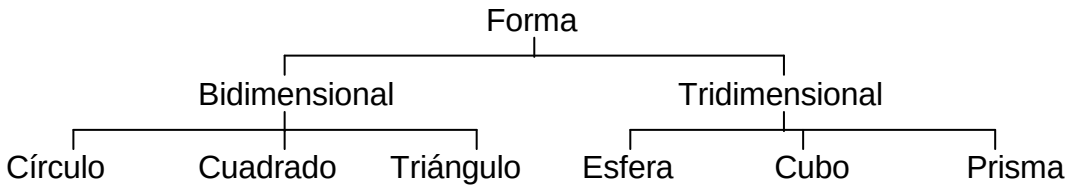
- La **homonimia o sobrecarga** no usa ligadura dinámica, ya que es resuelta en tiempo de compilación al seleccionar la definición de función con la firma coincidente con la llamada.

IX.5 Clases abstractas y clases concretas

Se definen como **clases abstractas** a aquellas clases definidas para las cuales el programador no tiene intenciones de producir objeto. En rigor, **a partir de una clase abstracta no se pueden producir objetos**.

El único fin de la clase abstracta es proporcionar una clase base apropiada, a partir de la cual las clases puedan heredar interfaz y/o implementación.

Las **clases concretas** son aquellas a partir de las cuales sí se pueden generar objetos.



En este ejemplo, las clases **Forma**, **Bidimensional** y **Tridimensional** son clases abstractas, demasiado genéricas como para definir objetos con sentido.

Las clases **Círculo, Cuadrado, Triángulo, Esfera, Cubo** y **Prisma** son clases concretas ya que dan la definición que convierte en razonable la producción de objetos.

- Una **función virtual pura** es aquella que en su declaración contiene un inicializador de igual a cero. Es decir, una función virtual pura no tiene definición relativa en la clase base. Por ejemplo:

  ```
  virtual float ganancias( ) const = 0;
  ```

  Si una clase contiene funciones virtuales puras, una o más, se convierte en una clase abstracta.

- Una **función pura** debe ser declarada como virtual.

- En muchos casos, las clases abstractas forman los **niveles superiores** en la jerarquía de herencia y las clases concretas los niveles inferiores.

- Resulta obvio que, si una clase no redefine las funciones virtuales puras definidas en su clase base, es decir, las hereda como funciones virtuales puras, esta clase derivada también será abstracta.

IX.6 Polimorfismo

<u>Definición</u>: Capacidad de objetos de clases diferentes, relacionadas mediante herencia, de responder de distinta forma a una misma llamada a función miembro.

El polimorfismo se implementa vía **funciones virtuales**. Cuando se hace una solicitud a través de un puntero (o referencia) de clase base para usar una función virtual, C++ escoge la función redefinida correcta, en la clase derivada adecuada, asociada con el objeto.

Un **comportamiento es no polimórfico**, por ejemplo, si una función miembro no virtual es definida en la clase base y redefinida en una derivada. Si esta función es llamada mediante un apuntador de clase base se usará la versión de clase base. Si es invocada mediante un apuntador de clase derivada se usará la versión de clase derivada.

Mediante el uso de funciones virtuales y del polimorfismo, y dependiendo del tipo de objeto que reciba, la llamada a una función miembro puede hacer que ocurran distintas acciones. (**El programador se ocupa de generalidades y, en tiempo de ejecución, el entorno se ocupa de los específico**).

El **polimorfismo fomenta la extensibilidad** ya que el software se escribe independientemente del tipo de los objetos a los cuales se les envía mensajes. Nuevos tipos de objetos que pudieran responder a mensajes existentes se añaden sin modificar el sistema base (y sin necesidad de recompilar, salvo el código cliente que produce nuevos objetos).

Una clase abstracta define una interfaz para distintos miembros de una jerarquía de clases. La clase abstracta contiene funciones virtuales puras que serán definidas en las clases derivadas. Mediante el polimorfismo, todas las funciones en la jerarquía pueden usar esa interfaz.

Ejemplos de aplicación de polimorfismo y funciones virtuales son: Un administrador de pantalla que puede necesitar varias formas y donde cada objeto debe dibujarse a sí mismo; Un sistema Operativo O.O., implementado por capas, donde cada dispositivo físico opera de manera diferente a los demás.

Ejemplo: Programa para probar una jerarquía de clases usando funciones virtuales y polimorfismo.

```
//EMPLE2.H
//Clase base abstracta empleado
#ifndef EMPLE2_H
#define EMPLE2_H

class empleado {
public:
        empleado(const char*, const char*);     //Constructor
        ~empleado( );                           //Destructor
        const char* getnombre( ) const;
        const char* getapellido( ) const;

        //Funciones virtuales puras que hacen que la clase sea abstracta
        virtual float ganancias( ) const = 0;
        virtual void print( ) const = 0;
private:
        char *nombre;
        char *apellido;
};
#endif
//EMPLE2.CPP
//Definiciones de funciones miembros de empleado
//NOTA: No se definen las funciones virtuales puras
#include <iostream.h>
#include <string.h>
#include <assert.h>
#include “emple2.h”
//El constructor reserva espacio dinámicamente para el nombre
//y el apellido y usa strcpy para copiarlos en el objeto
empleado :: empleado (const char *nom, const char *apelli) {
        nombre = new char[strlen(nom)+1];
        assert(nombre != 0);                    //Probar que new funciono
        strcpy(apellido, apelli)
}
//El destructor libera la memoria reservada dinámicamente
empleado :: ~empleado( ) {
        delete [ ] nombre;
        delete [ ] apellido;
}
```

```
//Devuelve un puntero al nombre
const char *empleado :: getnombre( ) const; {
        //const evita que el cliente modifique datos privados
        //El cliente debiera copiar la tira devuelta antes de que
        //el destructor elimine el almacenamiento dinámico para
        //evitar un puntero indefinido

        return nombre;          //El cliente debe borrar la memoria
}
//Devuelve el puntero apellido
const char *empleado :: getapellido( ) const; {
        //const evita que el cliente modifique datos privados
        //El cliente debiera copiar la tira devuelta antes de que
        //el destructor elimine el almacenamiento dinámico para
        //evitar un puntero indefinido

        return apellido;        //El cliente debe borrar la memoria
}
//JEFE1.H
//Clase jefe derivada de la clase empleado
#ifndef JEFE1_H
#define JEFE1_H
#include “emple2.h”

class jefe : public empleado {
public:
        jefe(const char*, const char*, float = 0.0);
        void setsalariosemanal(float);
        virtual float ganancias( ) const;
        virtual void print( ) const;
private:
        float salariosemanal;
};
#endif
//JEFE1.CPP
//Definiciones de funciones miembro para la clase jefe

#include <iostream.h>
#include “jefe1.h”
//Funcion constructor para la clase jefe
jefe :: jefe(const char *nom, const char *apelli, float s)
        : empleado(nom, apelli)             //llama al constructor de la clase base
{salariosemanal = s > 0 ? s : 0;}
//Fijar el salario del jefe
void jefe :: setsalariosemanal(float s)
{salariosemanal = s > 0 ? s : 0;}
//Obtener la paga del jefe
float jefe :: ganancias( ) const {return salariosemanal;}
//Imprimir el nombre del jefe
void jefe :: print( ) const
```

```
{
        cout << "\n     Jefe: " << getnombre( ) << " " << getapellido( );
}
```
//COMISIO.H
//Clase comisionista derivada de la clase empleado
```
#ifndef COMISIO1_H
#define COMISIO1_H
#include "emple2.h"

class comisionista : public empleado {
public:
        comisionista(const char*, const char*, float = 0.0, float = 0.0, int = 0);
        void setsalario(float);
        void setcomision(float);
        void setcantidad(int);
        virtual float ganancias( ) const;
        virtual void print( ) const;
private:
        float salario;          //salario basico semanal
        float comision;         //monto por item vendido
        int cantidad;           //total items vendidos en semana
};
#endif
```
//COMISIO1.CPP
//Definiciones de funciones miembro para la clase comisionista
```
#include <iostream.h>
#include "comisio1.h"
```
//Funcion constructor para la clase comisionista
```
comisionista :: comisionista(const char *nom, const char *apelli, float s,
                float c, int q)
        : empleado(nom, apelli)          //llama al constructor de la clase base
{
        salario = s > 0 ? s : 0;
        comision = c > 0 ? c : 0;
        cantidad = q > 0 ? q : 0;
}
```
//Fijar el salario basico semanal del comisionista
```
void comisionista :: setsalario(float s)
{salario = s > 0 ? s : 0;}
```
//Fijar la comision del comisionista
```
void comisionista :: setcomision(float c)
{comision = c > 0 ? c : 0;}
```
//Fijar la cantidad vendida por el comisionista
```
void comisionista :: setcantidad(int q)
{cantidad = q > 0 ? q : 0;}
```

```
//Determinar las ganancias del comisionista
float comisionista :: ganancias( ) const {return salario +comision * cantidad;}
//Imprimir el nombre del comisionista
void comisionista :: print( ) const
{
        cout << "\n     Comisionista: " << getnombre( ) << " " << getapellido( );
}
//PORPIEZA1.H
//Clase empporpieza derivada de la clase empleado
#ifndef PORPIEZA1_H
#define PORPIEZA1_H
#include "emple2.h"

class empporpieza : public empleado {
public:
        empporpieza(const char*, const char*, float = 0.0, int = 0);
        void setpaga(float);
        void setcantidad(int);
        virtual float ganancias( ) const;
        virtual void print( ) const;
private:
        float pagaporpieza;          //paga por pieza producida
        int cantidad;                //cantidad de piezas producidas
};
#endif
//PORPIEZA1.CPP
//Definiciones de funciones miembro para la clase empporpieza

#include <iostream.h>
#include "porpieza1.h"
//Funcion constructor para la clase empporpieza
empporpieza :: empporpieza(const char *nom, const char *apelli, float w, int q)
        : empleado(nom, apelli)          //llama al constructor de la clase base
{
        pagaporpieza = w > 0 ? w : 0;
        cantidad = q > 0 ? q : 0;
}
//Fijar la paga del empleado por pieza
void empporpieza :: setpaga(float w)
{pagaporpieza = w > 0 ? w : 0;}
//Fijar la cantidad producida por el empleado por pieza
void empporpieza :: setcantidad(int q)
{cantidad = q > 0 ? q : 0;}
//Determinar las ganancias de un empleado por pieza
float empporpieza :: ganancias( ) const
{return cantidad* pagaporpieza;}
```

```
//Imprimir el nombre del empleado por pieza
void empporpieza :: print( ) const
{
        cout << "\nEmpl.p/pieza: " << getnombre( ) << " " << getapellido( );
}
//PORHORA1.H
//Clase empporhora derivada de la clase empleado
#ifndef PORHORA1_H
#define PORHORA1_H
#include "emple2.h"

class empporhora : public empleado {
public:
        empporhora(const char*, const char*, float = 0.0, float = 0.0);
        void setpaga(float);
        void sethoras(float);
        virtual float ganancias( ) const;
        virtual void print( ) const;
private:
        float paga;                 //paga por hora trabajada
        int cantidad;               //cantidad de horas trabajadas
};
#endif
//PORHORA1.CPP
//Definiciones de funciones miembro para la clase empporhora

#include <iostream.h>
#include "porhora1.h"
//Funcion constructor para la clase empporpieza
empporhora :: empporhora(const char *nom, const char *apelli, float w, float h)
        : empleado(nom, apelli)          //llama al constructor de la clase base
{
        paga = w > 0 ? w : 0;
        horas = h >= 0 && h < 168 ? h : 0;
}
//Fijar la paga del empleado por hora
void empporhora :: setpaga(float w)
{paga = w > 0 ? w : 0;}
//Fijar las horas trabajadas por el empleado por hora
void empporhora :: sethoras(float h)
{ horas = h >= 0 && h < 168 ? h : 0;}
//Determinar las ganancias de un empleado por hora
float empporhora :: ganancias( ) const
{return paga* horas;}
//Imprimir el nombre del empleado por hora
void empporhora :: print( ) const {
        cout << "\nEmpl.p/hora: " << getnombre( ) << " " << getapellido( ); }
```

```
//PRUEBA.CPP
//Driver para probar la jerarquía de empleados
#include <iostream.h>
#include <iomanip.h>
#include “emple2.h”
#include “jefe1.h”
#include “comisio1.h”
#include “porpieza1.h”
#include “porhora1.h”
main( ) {
        //Fijar formateo output
        cout << setiosflag(ios:showpoint) << setprecision(2);

        empleado *ptr;                  //crea un objeto puntero de clase base

        jefe b(“Juan”, “Perez”, 800.00)
        ptr = &b;               //puntero de clase base apunta a objeto de jefe
        ptr → print( );                                 //Ligadura dinámica
        cout << “Gano $ “ << ptr → ganacias( );  //Ligadura dinámica
        b.print( );                                     //lLigadura estatica
        cout << “Gano $ “ << b.ganancias( );     //Ligadura estatica

        comisionista c(“Paula”, “Gomez”, 200.00, 3.0, 150)
        ptr = &c;               //puntero de clase base apunta a objeto de
                                //comisionista
        ptr → print( );                                 //Ligadura dinámica
        cout << “Gano $ “ << ptr → ganacias( );  //Ligadura dinámica
        c.print( );                                     //lLigadura estatica
        cout << “Gano $ “ << c.ganancias( );     //Ligadura estatica

        empporpieza p(“Carlos”, “Sanchez”, 2.5, 200)
        ptr = &p;               //puntero de clase base apunta a objeto de
                                //empporpieza
        ptr → print( );                                 //Ligadura dinámica
        cout << “Gano $ “ << ptr → ganacias( );  //Ligadura dinámica
        p.print( );                                     //lLigadura estatica
        cout << “Gano $ “ << p.ganancias( );     //Ligadura estatica}

        empporhora h(“Juana”, “Acosta”, 13.75, 40)
        ptr = &h;               //puntero de clase base apunta a objeto de
                                //empporhora
        ptr → print( );                                 //Ligadura dinámica
        cout << “Gano $ “ << ptr → ganacias( );  //Ligadura dinámica
        h.print( );                                     //lLigadura estatica
        cout << “Gano $ “ << h.ganancias( );     //Ligadura estatica
        cout  << endl;
return 0;
}
```

La salida de este programa sería

```
Jefe: Juan Perez Gano $ 800.00
Jefe: Juan Perez Gano $ 800.00
Comisionista: Paula Gomez Gano $ 650.00
Comisionista: Paula Gomez Gano $ 650.00
Empl.p/pieza: Carlos Sánchez Gano $ 500.00
Empl.p/pieza: Carlos Sánchez Gano $ 500.00
Empl.p/hora: Juana Acosta Gano $ 550.00
Empl.p/hora: Juana Acosta Gano $ 550.00
```

IX.7 Clases nuevas y ligadura dinámica

El polimorfismo y las funciones virtuales funcionan bien, tanto cuando las clases son conocidas con antelación como cuando a los sistemas se les añaden nuevas clases mediante **ligadura dinámica** (o **ligadura tardía**).

Para que una función virtual sea compilada **no es necesario conocer el tipo del objeto**. La llamada a función virtual se hace coincidir, en tiempo de ejecución, con la función miembro del objeto llamado.

Estas ideas facilitan el añadir, con un impacto mínimo, nuevas capacidades a los sistemas y promueve la reutilización de software. Así, los fabricantes independientes de software pueden distribuirlo sin revelar secretos propietarios.

La ligadura dinámica requiere que, en tiempo de ejecución, se encamine la llamada a una función miembro virtual hacia la versión de función virtual apropiada para la clase. Se implementa una **vtable** (**tabla de funciones virtuales**), en forma de un arreglo, que contiene punteros a funciones.

Cada clase contiene que contiene funciones virtuales contiene una **vtable** y para cada función virtual dentro de la clase, la vtable tiene una entrada que contiene un apuntador a la versión de función virtual para uso de un objeto de dicha clase. Esta función virtual puede ser una función virtual definida en la clase o una función heredada, directa o indirectamente, de una clase de más alta jerarquía.

Cuando una clase base proporciona una función miembro y la declara virtual, las clases derivadas pueden o no redefinirla. Por lo tanto, una clase derivada puede usar la versión de clase base de una función virtual, lo que queda indicado en la **vtable**.

Cada objeto de una clase con funciones virtuales **contiene un apuntador a la vtable** de dicha clase. Este puntero no es accesible al programador. Se obtiene el puntero apropiado de función en la vtable y se desreferencia para completar la llamada en tiempo de ejecución. Esta búsqueda y desreferenciación requiere una sobrecarga nominal en tiempo de ejecución.

IX.8 Constructores y destructores virtuales

El uso de polimorfismo para procesar objetos dinámicamente asignados a una jerarquía de clases, puede ocasionar algún problema.

Si el objeto es destruido aplicando el operador **delete** a un puntero de clase base del objeto, se invocará la función destructor de clase base sobre el objeto, independientemente del tipo del objeto y de que el destructor de cada clase tenga nombre distinto.

Los constructores no pueden ser virtuales, pero los destructores si.

Para dar solución a esta posibilidad, se declara virtual al destructor de clase base. Esto hará, automáticamente, virtuales a todos los destructores de las clases derivadas, aunque no tengan el mismo nombre. Si este es el caso, cuando se quiera destruir explícitamente un objeto mediante delete, y el objeto es de clase derivada pero está apuntado por un puntero de clase base, se invocará al destructor apropiado.

El destructor de una clase derivada que tiene en alguna clase base un destructor virtual, es también virtual.

Se aconseja que, si una clase contiene funciones virtuales, se incluya un destructor virtual, aún si la clase no lo requiere.

Ejemplo: Driver de prueba de jerarquía de clases

```
//FORMA.H
//Declaración de la clase FORMA
#ifndef FORMA_H
#define FORMA_H

class forma {
public:
        virtual float area( ) const {return 0.0;}
        virtual float volumen( ) const {return 0.0;}
        virtual void printnomforma( ) const = 0;     //funcion virtual pura
};
#endif
//PUNTO1.H
//Declaración de la clase PUNTO
#ifndef PUNTO1_H
#define PUNTO1_H
#include <iostream.h>
#include “forma.h”
```

```
class punto : public forma {
                friend ostream &operator << (ostream&,const punto&);
public:
                punto(float = 0.0,float = 0.0);         //Constructor default
                void setpunto(float,float);             //Fijar coordenadas
                float getX( ) const {return x;}         //get coordenada x
                float getY( ) const {return y; }        //get coordenada y
                virtual void printnomforma( ) const {cout << "Punto: ";}
protected:
                float x,y;                      //x e y coordenadas de punto
};
#endif
```

//PUNTO1.CPP

//Definicion de funciones miembros de la clase PUNTO

```
#include <iostream.h>
#include "punto1.h"
```

//Constructor de la clase punto

```
punto :: punto(float a, float b)
{
        x = a;
        y = b;
}
```

//Fijar coordenadas de punto

```
void punto :: setpunto(float a, float b)
{
        x = a;
        y = b;
}
```

//Imprimir punto con operador sobrecargado

```
ostream &operator << (ostream &output, const punto &p)
{
        output << "[" << p.x << "," << p.y << "]";
        return output;                  //Habilita llamadas concatendas
}
```

//CIRCULO1.H

//Declaracion de la clase circulo

```
#ifndef CIRCULO1_H
#define CIRCULO1_H
#include "punto1.h"
```

```
class circulo : public punto {
//Circulo hereda de punto en forma publica
        friend ostream &operator << (ostream&, const circulo&);
public:
        circulo(float r = 0.0, float x = 0.0, float y = 0.0);   //constructor default
        void setradio(float);                                   //fija el radio
        void getradio( ) const;                                 //get del radio
        virtual float area( ) const;                            //calcula el area
        virtual void printnomforma( ) const {cout << "Circulo: ";}
protected:
        float radio;
};
#endif
//CIRCULO.CPP
//Defincion de las funciones miembros de circulo
# include <iostream.h>
#include <iomanip.h>
#include "circulo1.h"
//El constructor de circulo llama al constructor de punto
circulo :: circulo(float r, float a, float b)
        : punto(a,b)            // llamada al constructor de punto
{radio = r > 0 ? r : 0; }
//fijar el radio del circulo
void circulo :: setradio(float r) {radio = r > 0 ? r : 0; }
//obtener el radio
float circulo :: getradio( ) const {return radio; }
//calcular el area del circulo
float circulo :: area( ) const
{return 3.14159 * radio * radio;}
//Imprimir circulo en el formato centro = [x,y]; radio = #.##
ostream &operator << (ostream &output, const circulo &c)
{
        output << "centro = [" << c.x << "," << c.y << "]; Radio = "
                << setiosflag(ios:showpoint) << setprecision(2) << c.radio;
        return output;          //habilita llamadas concatenadas
}
//CILINDRO1.H
//Definicion de la clase cilindro
#ifndef CILINDRO1_H
#define CILINDRO1_H
#include "circulo1.h"

class cilindro : public circulo {
//Cilindro hereda de circulo en forma publica
        friend ostream &operator << (ostream&, const cilindro&);
public:
        cilindro(float h = 0.0,float r = 0.0, float x = 0.0, float y = 0.0);
        //constructor default
        void setaltura(float);                                  //fija la altura
```

```
        virtual float area( ) const;                    //calcula el area
        virtual float volumen( ) const;                 //calcula el volumen
        virtual void printnomforma( ) const {cout << "Cilindro: ";}
protected:
        float altura;
};
#endif
```

//CILINDRO1.CPP
//Defincion de las funciones miembros de cilindro

```
# include <iostream.h>
#include <iomanip.h>
#include "cilindro1.h"
```

//El constructor de cilindro llama al constructor de circulo

```
cilindro :: cilindro(float h,float r, float x, float y)
        : circulo(r,x,y)          // llamada al constructor de circulo
{altura = h > 0 ? h : 0; }
```

//fijar la altura del cilindro

```
void cilindro :: setaltura(float h) {altura = h > 0 ? h : 0; }
```

//calcular el area del cilindro

```
float cilindro :: area( ) const
{return 2 * circulo :: area( ) + 2 * 3.14159 * circulo :: getradio( ) * altura;}
```

//calcular el volumen del cilindro

```
float cilindro :: volumen( ) const
{
        float  r = circulo :: getradio( );
        return 3.14159 * r * r * altura;
}
```

//Imprimir cilindro

```
ostream &operator << (ostream &output, const cilindro &c)
{
        output << "[" << c.getX( ) << "," << c.getY( ) << "]; Radio = "
                << setiosflag(ios:showpoint) << setprecision(2) << c.getradio( )
                << ";Altura = " << c.altura;
        return output;          //habilita llamadas concatenadas
}
```

//PRUEBA.CPP
//Driver de prueba para la jerarquia forma

```
#include <iostream.h>
#include <iomanip.h>
#include "forma.h"
#include "punto.h"
#include "circulo.h"
#include "cilindro.h"
```

```
main( )
{
        //crear objetos de distintas formas
        punto punto(7,11);
        circulo circulo(3.5,22,8);
        cilindro cilindro(10,3.3,10,10);

        punto.printnoforma( );        //Ligadura estatica
        cout << punto << endl;
        circulo.printnomforma( );
        cout << circulo << endl;
        cilindro.printnomforma( );
        cout << cilindro << "\n\n";

        cout << setiosflag(ios:showpoint) << setprecision(2);
        forma *ptr;             //crea un puntero de clase base

        //apuntar el puntero de clase base a un objeto de clase punto
        ptr = &punto;
        ptr → printnomforma( );             //Ligadura dinamica
        cout << "x = " << punto.getX( ) << "; y = " << punto.getY( )
                << "\nArea = " << ptr → area( )
                << "\nVolumen = " << ptr → volumen( ) << "\n\n";

        //apuntar el puntero de clase base a un objeto de clase circulo
        ptr = &circulo;
        ptr → printnomforma( );             //Ligadura dinamica
        cout << "x = " << circulo.getX( ) << "; y = " << circulo.getY( )
                << "\nArea = " << ptr → area( )
                << "\nVolumen = " << ptr → volumen( ) << "\n\n";

        //apuntar el puntero de clase base a un objeto de clase cilindro
        ptr = &cilindro;
        ptr → printnomforma( );             //Ligadura dinamica
        cout << "x = " << cilindro.getX( ) << "; y = " << cilindro.getY( )
                << "\nArea = " << ptr → area( )
                << "\nVolumen = " << ptr → volumen( ) << "\n\n";

        return 0;
}
```

La salida sería:

```
Punto: [7,12]
Circulo: [22,8]; Radio = 3.50
Cilindro: [10,10]; Radio = 3.30 Altura 10.00
```

Punto: x = 7.00; y = 11.00
Area = 0.00
Volumen = 0.00

Circulo: x = 22.00; y = 11.00
Area = 38.48
Volumen = 0.00

Cilindro: x = 10.00; y = 10.00
Area = 275.77
Volumen 342.12

IX.9 Clases heredadas virtualmente

Una clase base, o no base, no puede ser declarada como "virtual", pero puede ser heredada virtualmente; en este caso, todos sus miembros, incluidos los destructores son virtuales.

Los constructores de las clases base heredados virtualmente son invocados antes que los de las clases base heredados no virtualmente. Y dentro de los virtuales, se los invoca en el orden en que fueron declarados.

Cabe recordar que los constructores de las clases base heredados no virtualmente tienen más prioridad que cualquiera de sus derivados, sean virtuales o no. Es decir, si una clase virtual deriva de una clase base no virtual, la clase base no virtual será invocada primero, de forma que la clase derivada virtual pueda ser construida adecuadamente.

```
class X; {/*...*/}
class Y; {/*...*/}
class Z : public Y, virtual public X {/*...*/};
class Z1 : public Y, virtual public X {/*...*/};
class Z2 : public Z, virtual public Z1 {/*...*/};
Z2 a;
// orden de llamada de los constructores de a:
// X(), Y(), Z1(), Y(), Z(), Z2()
// los constructores heredados virtualmente sólo se
// invocan una vez.
```

Una clase sólo puede ser especificada una vez en una clase derivada. Sin embargo, puede ser heredada indirectamente más de una vez.

```
class A;
class B : public A,A;          // error. A se especifica 2 veces
class C : public A;
class D : public A;
class E : public C, public D; // correcto.
```

En este ejemplo, cada instancia de la clase E tendrá objetos de la clase A. Para evitar esta duplicidad, basta con añadir la palabra **virtual** al especificador de la clase base A en el momento de la herencia.

```
class B
{
public:
        int i;
        //...
};
class D1 : virtual public B
{
        //...
};
class D2 : virtual public B
{
        //...
};
class D3 : public D1, public D2
{
        //...
}
void main()
{
        D3 d;
        d.i = 10;               // no es ambiguo
        //...
}
```

Nota: Si se quita la palabra virtual, la asignación d.i = 10 sería ambigua, ya que podría corresponder a d.D1:i ó a d.D2: i.

IX.10 Clases contenedores y clases iteradores

Las clases **contenedores** o **de colección** son clases diseñadas para contener colecciones de objetos. Por lo general, proporcionan servicios como inserción, búsqueda, clasificación, prueba de un elemento verificando su membresía dentro de la clase, y otras similares. Ejemplos de clases contenedores son los arreglos y las listas enlazadas.

Es común asociar **iteradores** u **objetos de iterador** con la clase de colección. Un iterador es un objeto que retorna el elemento siguiente de una colección.

Una vez que se ha escrito un iterador para una clase, obtener el siguiente elemento de la clase puede ser expresado de forma simple. Los iteradores se escriben como amigos de la clase, a través de los cuales se hace la iteración.

Una clase contenedor puede tener varios iteradores operando sobre ella simultáneamente.

X. Plantillas (TEMPLATES)

Ya se comentó que una característica básica de la POO es el "**Polimorfismo Paramétrico**";: **Capacidad de definir tipos genéricos** (estructuras paramétricas) **que dependiendo del tipo del mensaje que actúe sobre la estructura genérica, se genere un tipo específico de estructura**.

Así, en el caso de la estructura genérica Pila[X] se podían generar estructuras cuyos componentes fueran valores enteros (Pila[entero]), reales (Pila[real]), etc. usando un único patrón (o clase) para dicha instanciación.

C++, a partir de la versión 3.0, incorpora la palabra reservada **template** para definir e implementar estos tipo genéricos. Las plantillas (o **templates**) proporcionan el mecanismo idóneo para definir los tipos genéricos que necesitan cambiar con cada instancia de clase, y se realiza **parametrizando** los tipos dentro de una definición de clase **template**.

X.1 Plantillas de función

Las plantillas de función, o **función genérica**, proporcionan el mecanismo adecuado para parametrizar un conjunto de **tipos de interfaz** (los argumentos y el tipo de retorno) de la función, mientras que el cuerpo de la misma permanece invariante.

```
// La plantilla de la función absoluto tiene el
// valor absoluto de un número
template <class Tabs>
Tabs absoluto (Tabs a)
{
        return a < 0 ? -a : a;
}
void main ()
{
        cout << abosluto(-40);
        // se llama como: int absoluto(int);
        cout << absoluto(-20.32);
        // se llama como: double absoluto(double);
}
```

- La palabra **template** encabeza tanto la declaración como la definición de la función genérica.

- Encerrada entre los símbolos < y > va una lista, separada por comas, **de tipos de parámetros formales**. Esta lista, que no puede estar vacía, se denomina la "**lista de parámetros formales de la platilla**".

- Cada parámetro formal consta de la palabra **class** seguida de un identificador que no puede estar repetido en la lista de parámetros, aunque puede ser reusado en otras declaraciones de la función genérica.

```
// imprime el valor mínimo
template <class T, class T1>
void impmin (T a, T1 b)
{
        if (a < b)
                cout << a;
        else
                cout << b;
}
void main ()
{
        int i;
        double d;

        impmin (i,i);
        // llama a impmin(int,int)
        impmin(i,d);
        // llama a impmin(int,double)
}
```

- La instanciación específica de una función genérica (o plantilla de función) se denomina **función de plantilla** ó **función plantilla**.

- En tiempo de compilación, el tipo paramétrico T o T1 **es sustituido** por el tipo actual de la instancia que invoca a la función.

- Siguiendo las reglas de sobrecarga de funciones, se puede definir explícitamente una función no plantilla con el mismo nombre. En este caso, esta declaración tiene precedencia sobre la plantilla.

```
// obtiene el valor mayor entre x e y
template <class Tmay>
Tmay mayor (Tmay x, Tmay y)
{
        return (x > y) ? x : y;
}
float mayor (float x, float y)
//función no plantilla sobrecargada
{ if (x > y)
                return x;
        else
                return y; }
void main ( ) {
        int i;
        float f;
        cout << mayor(i,i);
        // función plantilla int mayor(int,int);
        cout << mayor(f,f);
        // función no plantilla float mayor(float,float);
}
```

- La única restricción en el uso de los parámetros formales de una función genérica es que cada uno de ellos debe figurar, al menos una vez, como argumento de la función.

```
template <class X, class Y, class Z>
X fun(Y,Z);
// error. X no figura como argumento
```

X.2 Plantilla de clase

Las clases plantillas a menudo se conoce como **tipos parametrizados** porque requieren uno o más parámetros de tipo para especificar cómo personalizar la especificación de la plantilla genérica.

Una plantilla de clase, también llamada **clase genérica** o **generador de clase** permite definir un modelo para definiciones de clases. Las clases contenedores genéricas, que operan sobre una colección de 0 ó más objetos de un tipo particular, son buenos ejemplos de plantillas de clases.

El programador que desee especificar clases plantillas sólo escribe una definición de clase plantilla genérica. Cada vez que necesite una nueva clase, usa una notación simple y sencilla y el compilador escribe el código fuente para la clase que especifica el programador.

Por ejemplo, la clase plantilla pila podría convertirse en base para crear muchas clases de pilas (pila float, pila int, pila char, etc.) usadas en el programa.

Las declaraciones y definiciones de las **clases genéricas** son semejantes a las descritas para funciones plantillas. La palabra **template** encabeza la declaración/definición de la clase genérica, seguida de una lista de parámetros formales de la plantilla clase encerrada entre < >. La lista no puede estar vacía y los parámetros están separados por comas.

En otras palabras, la definición de una clase plantilla está precedida por el encabezado:

```
template <class T>
```

para indicar que se trata de la definición de una **clase plantilla que recibe un parámetro de tipo T** indicando el tipo de la clase pila que el programador desea crear.

El tipo de elemento a almacenarse en esta pila se menciona sólo genéricamente como T a lo largo del encabezado de la clase y de las definiciones de función miembro.

```
//TSTACK1.H
//Plantilla simple de la clase stak
#indef TSTACK1_H
#define TSTACK1_H

template <class T>
class stack
{
public:
        stack(int=10);                          //Constructor default
        ~stack( ) {delete [ ] stackptr;}        //Destructor
        int push(const T&);                     //Push un elemento en la pila
        int pop(T&);                            //Pop un elemento fuera de la pila
        int isEmpty( ) const {return top == -1;}     //1 si es vacia
        int isFull( ) const {return top == size - 1;} //1 si es llena
private:
        int size;               //Nro. de elementos de la pila
        int top;                //Ubicacion del elemento top
        T*stackptr;             //puntero a la pila
};
// Nótese la sintaxis de la definción de la clase stack.
//Constructor
template <class T>
stack <T> :: stack(int s)
{
                size= s;
                top = -1;                       //Pila vacia
                stackptr = new T[size];
}
//Destructor
template <class T>
stack <T> :: ~stack( )
{
        delete [ ] size;
}
//Push de un elemento en la pila
//devolver 1 si es exitoso, sino 0
template <class T>
int stack <T> :: push (const T &item)
{
        if (!isFull( ))
        {
                stackptr[++top] = item;
                return 1;                       //push exitoso
        }
        return 0;                               //push fallido
}
```

```
//Pop un elemento fuera de la pila
template <class T>
int stack <T> :: pop(T &popvalue)
{
        if (!isEmpty( ))
        {
                popvalue = stackptr[top - -] ;
                return 1;                       //pop exitoso
        }
        return 0;                               //pop fallido

}
#endif
//PRUEBA.CPP
//Driver de prueba para la plantilla stack
#include <iostream.h>
#include “tstack1.h”

void main ()
{
        stack<float> floatstack(5);             //genera un vector de 5 float
        float f = 1.1;
        cout << “Pushing elementos en el floatstack \n”;
        while (floatstack.push(f))
        {                                       //Arma una pila con 1.1, 2.2, 3.3, 4.4, 5.5
                cout << f << “ “;               //El 6.6 no lo puede poner y la pila esta llena
                f += 1.1;                       //Termina el lazo.
        }
        while(floatstack.pop(f))                //Saca todos los elementos de la pila
                cout << f << “ “;               //hasta que queda vacia.
        cout << “\n Pila vacia. No puede pop. “ << endl;

        stack<int> intstack;                    //or default, genera un vector de 10 int
        int i = 1;
        cout << “Pushing elementos en el intstack \n”;
        while (intstack.push(i))
        {
                cout << i << “ “;
                i++;
        }
        while(intstack.pop(i))
                cout << i << “ “;
        return 0;
}
```

- Vemos que en un caso float sustituye a T y luego int sustituye a T.

- Los tipos parametrizados o plantillas pueden tomar más de un parámetro, pero cada uno de los ellos debe estar precedido de la palabra class:

  template <class T, class S, class R>

- Las clases **contenedores genéricas**, que operan sobre una colección de 0 o más objetos de un tipo particular, son buenos ejemplos de plantillas de clases.

- Las declaraciones y definiciones de clases genéricas son similares a las descriptas para funciones.

Veamos un ejemplo de la clase “Vector” parametrizada, que permite crear instancias de vectores con distintos tipos de valores.

```
template <class TVector>
class Vector
{
        int dim;
        TVector *pv;
public:
        vector (int d = 10) {dim = d; pv = new TVector[dim];}
        ~vector( ) {delete [ ] pv;}
        int dimension ( ) {return dim;}
        TVector &operator [ ] (int ind) {return pv[ind];}
};
void main ( )  {
        vector<int> vi;          //genera vector de 10 int
        vector <float> vf(5);  //genera vector de 5 float
        vector <char> vc(8); //genera vector de 8 char
        int i;
        for (i = 0, i < vi.dimension( ) , ++i)
        {       vi[i] = i;
                cout << vi[i] << " "; }
        cout << endl;
        for (i = 0, i < vf.dimension( ) , ++i)
        {       vf[i] = i + 0.5;
                cout << vf[i] << " "; }
        cout << endl;
        for (i = 0, i < vc.dimension( ) , ++i)
        {       vc[i] = i + 97;
                cout << vc[i] << " "; }
        cout << endl;
}
//Imprime
//0 1 2 3 4 5 6 7 8 9
//0.5 1.5 2.5 3.5 4.5
//a b c d e f g h
```

- La plantilla de clase vector no comprueba si un subíndice se sale de los límites del vector definido en la instanciación. Para hacerlo es útil emplear la macro: **void assert(int test);** definida en el archivo de cabecera <assert.h>, que imprime un mensaje de error y aborta el programa si no se cumple la expresión booleana definida por test.
  Así, la función operator de la plantilla clase vector quedaría definida como:

```
Tvector &operator [ ] (int ind)
{
        assert( ind >= 0 && ind < dim);
        return pv[ind];
}
```

- En el caso de que se defina explícitamente una clase plantilla, esta definición anula la definición automática de la clase correspondiente a la clase genérica.

```
//clase genérica pila
template <class T>
class pila
{
        ….
}
```

```
class pila<int>
{
        …
}
```

Esta clase anula a la que se generaría con la clase genérica en la instanciación pila<int> p(10), por ejemplo.

- Una clase genérica también puede contener en su lista de parámetros formales argumentos de tipos predefinidos, cuyos valores deben ser expresiones constantes en las sentencias de llamada a la clase genérica.

```
//clase plantilla pila
template <class T, int dim = 5>
{
        T *tam;
        int nelem;
public:
        pila( );
        ~pila( );
        int esvacia( ) {return nelem == 0;}
        T cabeza( ) {return tam[nelem];}
        void insertar(T);
        void eliminar( ) {if (nelem != 0) –nelem;}
 };
```

```
template <class T, int dim>
pila<T,dim> :: pila( )
{
        tam = new T[dim];
        nelem = 0;
        cout << “Pila de “ << dim << “ elementos ya construida”
                << endl;
}
template <class T, int dim>
pila<T,dim> :: ~pila( )
{
        delete [ ] tam;
        cout << endl; << “Se borro la pila “ << endl;
}
template <class T, int dim>
pila<T,dim> :: insertar(T x )
{
        if (nelem < dim)
        {
                nelem++;
                tam[nelem] = x;
        }
}
//Programa de prueba
int main( )
{
        pila<int> p1;           //crea una pila de tamaño 5 de enteros
        pila<int,4> p2;         //crea una pila de tamaño 4 de enteros
        pila<float,10> p3;      //crea una pila de tamaño 10 de float

        pila <int,3 p>;
        for (int i= 1; i < 11; ++i)
        {
                p.insertar(i);
                cout << p.cabeza( ) << endl;
                //Escribe: 12333333333
        }
        int i;
        cin << i;
        return 0;
}
```

XI Clases: Struct y Unión
XI.1 Clases struct

En C++, una estructura (**struct**) es un "tipo clase" que se caracteriza porque sus miembros son **public**, por defecto, y en la jerarquía de herencia, la transmisión también es **public**, por defecto.

Como este tipo (al igual que el tipo **union**) se incorpora del ANSI C, cabe resaltar las características más relevantes que hereda del C y que, en líneas generales, son aplicables al tipo **class**.

En ANSI C, el tipo **struct** representa una colección de variables que se referencian bajo un único nombre. Es pues, un tipo derivado que agrupa una colección de miembros con nombre, denominados **componentes**.

Los nombres de las estructuras comparten el mismo espacio que el de las uniones y enumeraciones, razón por la que estos tipos deben tener nombres diferentes. Pero el nombre de alguno de sus miembros puede ser el mismo que el de la estructura.

- Los nombres de las componentes de una estructura son únicos dentro de la misma. Sus miembros pueden ser de cualquier tipo, salvo de la misma estructura.

- Un miembro de una estructura no reserva espacio de memoria. Crea un nuevo tipo de datos que es usado para declarar variables.

- Las variables de tipo estructura se declaran de igual manera que las variables de otros tipos.

- Los miembros de una estructura se acceden usando los **operadores de acceso a miembros** que son:

  . operador de acceso directo (vía nombre de la variable)

  ```
  cout << timeobject.hour;
  ```

  → **operador de acceso indirecto** (vía una referencia a la variable)

  ```
  cout << (*timeptr).hour;
  ```

  ó

  ```
  cout << timeptr → hour;
  ```

- Si **alfa** es una estructura miembro de otra **beta**, el acceso a una miembro de **alfa** a través de **beta** requiere dos aplicaciones del selector de componente.

```
struct alfa {
        int j;
        double x;
};
struct beta {
        int i;
        struct alfa a;
        double d;
}s, *spt, s1;
//…
        s.i = 3;
        s.a.j = 4;
        (spt → a).x = 3.14;
```

Problemas: No existe interfaz con el programa para asegurar el estado consistente de los datos.
Las estructuras de C, en general, se manipulan (imprimen y comparan) miembro a miembro y no como una unidad.

XI.2 Uniones

En C++ la union es un tipo de clase en la que todos sus miembros son **public**, no admitiendo explícitamente ningún tipo de especificador de acceso (**public**, **private** o **protected**).

Las uniones no pueden participar del proceso de herencia y sólo admiten un constructor.

En ANSI C, las uniones son de tipos derivados que se corresponden en gran medida con el concepto de los registros variantes de Pascal. Sus miembros comparten la misma zona de memoria y sólo se permite que uno de ellos esté activo.

La dimensión de una unión es pues, la dimensión del miembro más largo.

```
union U
{
        char a;
        int b;
        float c;
        double d;
} x, *px;
```

La variable x del tipo union U tiene una longitud: **sizeof**(U) == 8 bytes, que viene dada por la longitud del miembro mayor (**double** d). Cuando x opera con el miembro a de tipo **char**, hay 7 bytes que no se usan, y cuando opera con el miembro b (entero) no usa 6 bytes.

- El acceso a un miembro de una union es semejante a lo explicado para el caso de la estructuras (**struct**) y usa para ello los operadores . y ->.

```
px = new U;
x.a = 'h';
px -> b = 20;
x.c = 14.28;
px -> d = 43254.732;
```

- Una variable **union** sólo puede ser incializada a través de su primer miembro declarado.

```
union ejem
{
        int a;
        float f;
} x = {5};
```

- En C++ las uniones anónimas no pueden tener funciones miembro.

```
union
{
        //...
        void f(int,float);      // error
} x,y;
```

ANEXO II

Sintaxis del lenguaje de programación C++

Estructura y Procesamiento de un programa en C++

Todos los programas en C++ están formados, principalmente por:

declaraciones
definiciones de funciones
variables
clases

La única función que obligatoriamente debe estar presente es **main ()**, siendo ésta la primera en ser llamada cuando se inicia la ejecución del programa.

Observe en el siguiente ejemplo, los elementos que constituyen un programa en C++:

```
# include <oistream.h>                Directiva de preprocesamiento
class integer {
public:
        int i;
        void set (int ii = 0) {i=ii;}        Especificación de
        integer operator + (int);            declaración de la clase
        integer operator + (integer);        "integer"
};
integer integer :: operator + (int x)
{
         integer result;
         result.set (i + x);
         return result;
}                                             Especificaciones de
integer integer :: operator + (integer x)     definición de los
{                                             métodos +
         integer result;
         result.set (i + x.i);
         return result;
}

// Programa Principal:
void main ()
{
        integer A; A.set (10);
        integer B;
        integer C; C.set (20);                Declaración y
        B = A + 4;                            definición de la
        cout << B.i << "\n";                  función main ().
        B = A + C;
        cout << B.i;
}
```

Es decir:

- Directivas y macros de preprocesamiento que aumentan el alcance del entorno de programación, incrementando su potencia y flexibilidad, aunque no sean realmente sentencias del lenguajes C++.

- Especificaciones de declaraciones de clases, funciones (prototipos) y variables.

- Especificaciones de definición de métodos y funciones (especialmente, la función **main ()**).

Elementos básicos del lenguaje C++

El lenguaje C++ reconoce 6 (seis) clases de unidades léxicas (tokens): identificadores, palabras reservadas, constantes, tiras de caracteres, expresiones constantes, operadores y separadores.

Identificadores

Son nombres usados para referirse a las constantes, variables, tipos, funciones, etiquetas y otros objetos definidos por el usuario. Usando una definición formal BNF (Backus Normal Form)

<identificador> :: <letra>|<identificador><letra>|<identificador><dígito>
<letra> :: a-z|A-Z|_
<dígito> :: 0-9

C++ no impone límite a la cantidad de caracteres de un identificador, pero algunos compiladores como el Borland C++ lo hacen, por omisión, en 32.

C++ distingue entre letras mayúsculas y minúsculas, así *casa* es un identificador distinto a *Casa*.

Se sugiere no usar identificadores que comiencen con un doble subrayado (__) porque son reservados para las implementaciones y bibliotecas estándar de C++, ni identificadores que comiencen con un subrayado (_) pues se suelen reservar para implementaciones de C.

Palabras reservadas

Son palabras usadas por el compilador para acciones específicas y no deben ser usadas por el programador como identificadores fuera de su contexto.

asm	continue	float	new	signed	try
auto	default	for	operator	sizeof	typedef
break	delete	friend	private	static	union
case	do	goto	protected	struct	unsigned
catch	double	if	public	switch	virtual
char	else	inline	register	template	void
class	enum	int	return	this	volatile
const	extern	long	short	throw	while

Existen también un número de palabras reservadas adicionales que son específicas de cada implementación.

Constantes

Son valores fijos de caracteres o números que no pueden ser alterados a lo largo del programa. C++ soporta 4 clases de constantes: enteras, reales, enumeradas o caracter.

a) Constantes enteras: Pueden ser decimales, octales o hexadecimales. Su tipo depende de su forma, valor y sufijo. En ausencia del sufijo-entero (u, U, l o L) el tipo depende del valor.

Decimales:

de 0 a 32.767 (**int**)
de 32.768 a 2.147.483.647 (**long**)
de 2.147.483.647 a 4.294.967.295 (**unsigned long**)

Octales: de 00 a 077777 (**int**)
de 0100000 a 0177777 (**unsigned int**)
de 02000000 a 017777777777 (**long**)
de 020000000000 a 037777777777 (**unsigned long**)

Observe que las constantes octales comienzan con cero.

Hexadecimales: de 0X0000 a 0X7FFF (**int**)
de 0X8000 a 0XFFFF (**unsigned int**)
de 0X10000 a 0X7FFFFFFF (**long**)
de 0X80000000 a 0XFFFFFFFF (**unsigned long**)

Observe que las constantes hexadecimales comienzan con 0X ó 0x.

Las constantes decimales no pueden comenzar con 0 porque serían consideradas octales.

Las **constantes negativas** son constantes si signo, precedidas por el operador unario menos (-).

El **sufijo l ó L** obliga a que la constante sea del tipo **long**.
El **sufijo u ó U** obliga a que la constante sea del tipo **sin signo (unsigned)**.

Ejemplos:

```
const int maxint = 32767;                    // constante entera
const long cil = 416334851                   // cte. entera decimal larga
const unsigned long coul = 0377745ul
                                             // cte. entera octal larga
                                             // sin signo

const unsigned chu = 0X9999u                 // cte. entera hexadecimal
                                             // sin signo
```

b) Constantes reales:

Las constantes reales constan de 6 (seis) partes: parte entera; punto decimal; parte decimal o fraccionaria; e ó E; un exponente entero decimal; y un tipo sufijo: f ó F y l ó L.

Se puede omitir la parte entera o la parte fraccionaria, pero no ambas. También se puede omitir el punto decimal o la letra e ó E y el exponente entero con signo (pero no ambos). Los sufijos f ó F y l ó L son opcionales.

Las constantes reales negativas son consideradas como positivas precedidas del operador unario (-).

Por omisión del sufijo, las constantes reales son consideradas **double**, si llevan f ó F son consideradas de tipo **float**, y con el sufijo l ó L son consideradas del tipo **long double**.

El rango de valores disponible para estos tipos en el Borland C++ es:

float (32 bits): de 3.4×10^{-38} a $3.4 \times 10^{+38}$

double (64 bits): de 1.7×10^{-308} a $1.7 \times 10^{+308}$

long double (80 bits): de 3.4×10^{-4932} a $1.1 \times 10^{+4932}$

Ejemplos:

```
const float crf = 23.45e6f;
// crf == 23.45x10^6 en format real (float) de 32 bits
float ld = .09E20L;
// .09E20L es una cte. long double (80 bits) que se
// almacena en la variable real ld de 32 bits
const float cvd = 3.1416;
// 3.1416 es una cte. double (64 bits) que se almacena
// en la constante real (float) cvd de 32 bits.
```

c) Constantes de enumeración

Son identificadores definidos en declaraciones de tipo **enum**. Las constantes de enumeración son tipos de datos enteros y se pueden usar en cualquier expresión donde sea válida una cte. entera. El identificador usado debe ser único dentro del ámbito de la declaración de enum.

El valor que toman estas ctes. depende del formato de la declaración de enumeración y de la presencia opcional de inicializadores, estando permitidos los inicializadores negativos.

Ejemplos:

```
enum cosas {mesa,silla,escritorio};
// mesa, silla y escritorio son ctes. de enumeración
// que toman los valores: mesa=0, silla=1,escritorio=2.
// cosas es un identificador de tipo enum.

enum cosas
{mesa,silla = 4, ventana = mesa + 1,cubo = 1,casa = cubo - 1};
// las ctes. de enumeración toman los valores: mesa=0,
// silla=4, ventana=1, cubo=1, casa=0.
```

d) Constantes caracter

Una constante caracter es uno o más caracteres, o secuencia de escape, incluidos entre apóstrofes.

En C las ctes. caracter tienen tipo **int** y están formadas por uno o varios caracteres.

En C++, una cte. caracter tiene tipo **char**, mientras que una cte. multicaracter tiene tipo **int**.

El caracter \ se usa para definir una secuencia de escape, permitiendo así la representación de **caracteres no editables**. Algunos ejemplos de secuencias de escape disponibles en C++ son :

\a campana (bell)	\\ barra inversa
\b retroceso de un espacio	\' apóstrofe
\f alimentación de documento	\" comillas
\n nueva línea	\? signo de interrogación
\r retorno de carro	\0 0=número octal (max. 3 dígitos)
\t tabulador horizontal	\xH H = Un número hexadecimal
\v tabulador vertical	\XH H = Un número hexadecimal

El Borland C++ soporta ctes. de dos caracteres, que son valores enteros de 16 bits, ocupando el primer caracter el byte más significativo y el segundo caracter el byte menos significativo.

Ejemplos:

```
const cc='a',ce='\a',co='\l4l',ch='\x6l';
// definen la letra a, la campana, la letra a (en octal)
// y la letra a en hexadecimal
```

Tiras de caracteres

También conocidas como "constantes string", son secuencias de caracteres incluidas entre comillas. Una cadena de caracteres es una constante del tipo ***vector de caracteres***, y se almacena como **static**.

La definición formal de una tira de caracteres es:

<tira_caracteres> :: "<secuencia_caracter>"

<secuencia_caracter> :: <caracter_s>|<secuencia_caracter>
<caracter_s>

<caracter_s> :: cualquier caracter excepto " o nueva línea|
<secuencia_escape>

Ejemplo:

"Esto es una cadena de caracteres"
"" (cadena vacía)

Los caracteres entre comillas pueden incluir secuencias de escape, por ejemplo:

```
"\n\"Resultado\"\\\tTotal\n"
```

produce

```
"Resultado"\  Total
```

habiendo efectuado un salto de línea antes y otro después de la impresión.

Una tira de caracteres se almacena internamente como una secuencia de caracteres finalizando con el caracter nulo ('\0'). Una tira vacía sólo almacena el caracter nulo.

La dimensión de una tira de caracteres es igual al número de caracteres que la forman incluido el caracter nulo.

Cuando una tira no entra en una sola línea se puede continuar en la línea siguiente usando el caracter de continuación (\). Por ejemplo,

```
printf ("%s","esto es una tira\
de caracteres");
// Imprime: esto es una tira de caracteres
```

Por compatibilidad con el lenguaje C que permite modificar las cadenas de caracteres, C++ las define de tipo char[] y no const char[].

Expresión constante

Es una expresión que toma el valor de una constante. Se evalúa como una expresión regular y se puede usar en cualquier lugar donde sea válida una constante.

Su sintaxis es:

<expresión-constante> :: <expresión-condicional>

Una expresión constante no puede contener: los operadores de asignación (=), incremento (++), decremento (--), llamada a una función o la coma (,) salvo que estos operadores estén contenidos dentro del operando del operador **sizeof**.

Ejemplos:

```
const m = 4;
const n = 3;
const r = m > n ? m : n;
// m > n ? m : n es una expresión constante
// m, n y r son constantes enteras
```

Operadores

Los operadores son unidades léxicas que indican al compilador que efectúe determinados cálculos matemáticos y lógicos. C++ es muy rico en operadores: matemáticos, lógicos, de manipulación de bits, de acceso a componentes de estructuras, uniones y clases, y de operaciones con punteros.

En la siguiente tabla se muestra todos los operadores de C++ con su correspondiente orden de precedencia. Todos los operadores procesan la información de izquierda a derecha, excepto los de precedencia 2, 14 y 15 que lo hacen de derecha a izquierda.

p1.	() [] -> . ::
p2.	! ** ++ -- (tipo) *(dirección) & **sizeof** **new delete** + y - (unarios)
p3.	.* ->*
p4.	* / %
p5.	+ -
p6.	<< >>
p7.	< <= > >=
p8.	== !=
p9.	&
p10.	^
p11.	|
p12.	&&
p13.	||
p14.	?: (expresión condicional)
p15.	= += -= .= *= /= %= <<= >>= &= ^= |=
p16.	,

En la siguiente tabla se describe cada operador usado en la tabla de precedencia:

operador	tipo	Descripción
()	--	Operador de paréntesis
[]	binario	Operador de Array
->	binario	Selector de componente indirecto
.	binario	Selector de componente directo
::	binario	Acceso de ámbito para un miembro de una clase
!	unario	Negación lógica (NOT)
**	unario	Negación binario (complemento 1)
++	unario	Pre/post incremento
--	unario	Pre/post decremento
(tipo)	binario	Expresión cast
*	unario	Indirección
&	unario	Dirección
sizeof	unario	De dimensión
new	unario	Adquisición dinámica de memoria
delete	unario	Liberación dinámica de memoria
+	unario	Más unario
-	unario	Menos unario
.*	binario	Puntero a operadores miembros
->*	binario	Puntero a operadores miembros
*	binario	Multiplicación
/	binario	División
%	binario	Resto de un cociente entero
+	binario	Suma
-	binario	Resta
<<	binario	Desplazamiento a la izquierda
>>	binario	Desplazamiento a la derecha
<	binario	Menor que
<=	binario	Menor o igual que
>	binario	Mayor que
>=	binario	Mayor o igual que
==	binario	Igual a
!=	binario	No igual a
&	binario	Producto binario (AND binario)
^	binario	Suma excluyente binaria (XOR binario)
|	binario	Suma binaria (OR binario)
&&	binario	Producto Lógico (AND)
||	binario	Suma lógica (OR)
?:	ternario	Operador condicional
=	binario	Asignación
+=	binario	Asignación con suma
-=	binario	Asignación con resta
*=	binario	Asignación con multiplicación
/=	binario	Asignación con división
%=	binario	Asignación con resto
<<=	binario	Asignación con desplazamiento a la izquierda

>>=	binario	Asignación con desplazamiento a la derecha
&=	binario	Asignación con producto binario
^=	binario	Asignación con suma excluyente binaria
|=	binario	Asignación con suma binaria
,	binario	Operador coma

Ejemplos:	Resultado
int i = 0;	-
while (i++<4) cout <<i;	1 2 3 4
int i = 0;	-
while (++i<4) cout <<i;	1 2 3
int a,b = 20, c=14;	-
a = b % c; cout << a;	6

Delimitadores, puntuadores y comentarios

a) Delimitadores: También llamados separadores, se los conoce como espacios en blanco y sirven para separar identificadores, palabras reservadas y constantes adyacentes. En C++ los delimitadores son: espacio, tabulador, retorno de carro y salto de línea.

b) Puntuadores: Son partes obligatorias de la sintaxis del programa, pero no ejecutan ninguna acción específica. En C++ son: [] () {} , ; : ... * = # que se definen a continuación:

[ ] - Definen Arrays de una o varias dimensiones

( ) - Definen expresiones de grupos, aíslan expresiones condicionales, indican llamadas a funciones y agrupan parámetros de una función.

{} - Delimitan una sentencia compuesta o bloque.

, - Separa los argumentos de una función, y es operador en "expresiones coma".

; - Señala el final de una sentencia C++. Cualquier sentencia seguida de ; es interpretada como una "sentencia expresión".

: - Define una sentencia etiquetada (por ejemplo de un **case**).

... - Declara un número variable de argumentos. Se usa en la lista de argumentos de prototipos o de definición de funciones para indicar un número variable de argumentos o argumentos de tipos variables.

* - Se usa en tres contextos: a) Precediendo a una variable en una sentencia declarativa para crear un puntero a un tipo, b) Como operador de multiplicación y c) Como operador para desreferenciar un puntero.

= - Se aplica en dos contextos: a) Como operador de asignación y b) Como separador de la declaración de una variable de su lista inicializadora.

- Especifica una directiva del Preprocesador cuando aparece como primer caracter no blanco en la línea. También se usa como operador en la fase de preprocesamiento.

c) Comentarios: Es una secuencia de texto que se usa normalmente para explicar el código fuente. Son eliminados del texto en la fase de preprocesamiento. Existen dos estilos de comentarios: a) Estilo C, tira de texto encerrada entre los caracteres /* y */ y b) Estilo C++: Precedidos por // en cualquier posición de la línea y puede extenderse hasta el final de la misma.

Declaraciones

Una ***declaración*** es una sentencia que especifica la existencia de un objeto o una función y los atributos que el objeto o la función deberá tener. Por el contrario, una ***definición*** es una declaración que también hace una reserva de memoria para el objeto, y en el caso de la función, incorpora el código de implementación de la misma.

Un objeto o función puede ser declarado en distintos puntos de un programa fuente, pero una definición sólo puede aparecer una sola vez.

Ejemplos:

```
extern int z;                       // Declaración de z
int x = 4, y;                       // Definición de x y de y
float fun (int,float);              // Declaración de fun
float fun (int x, float y){/*...*/} // Definición de fun
```

Tipos de declaraciones

El tipo de elementos que pueden ser declarados en C++ son:

a) Objetos: variables, punteros, arrays, enumeraciones (**enum**), clases y miembros de una clase (incluyen los tipos **class**, **struct** y **union**).
b) Funciones
c) Etiquetas de sentencias
d) "Templates"
e) Macros del preprocesador

Declaración de objetos

En términos generales, un objeto es una región identificable de memoria que puede mantener un valor fijo o variable. Cada objeto viene definido por su nombre, tipo de dato y clase de almacenamiento.

a) Nombre: Se usa para acceder al objeto y puede ser un identificador o una expresión cuyo resultado sea un "puntero" a ese objeto.

Ejemplos:

```
int va; // va identifica un objeto entero
float a[3];     // a identifica un puntero al inicio
                // de una zona con tres elementos
                // reales
```

A la hora de usar el nombre del objeto se deben tener en cuenta los conceptos de: ámbito, visibilidad, duración y enlazado del identificador.

1) Ámbito: Es la parte del programa en que se puede usar un identificador para acceder a un objeto. Existen cuatro categorías: Local, Función, Archivo y Clase.

- Local (o de Bloque): Comienza y termina el en el bloque donde se declara el identificador.

- Función: Este nivel se usa para las etiquetas de las sentencias y los parámetros de definición de la función.

- Archivo: Son identificadores "globales" que deben ser declarados fuera de las funciones y clases. Su acceso es desde el punto de declaración al final del archivo.

- Clase: Tienen este acceso los datos y funciones (métodos) miembros de una clase (**class**, **struct**, **union**).

Un nombre declarado por una declaración de amistad (**friend**) tiene el mismo ámbito de la clase. Un nombre de clase declarado en un **return** o en un tipo argumento es de ámbito global.

- *Unicidad del nombre*: Es el ámbito dentro del cuál un identificador debe ser único.

Así:

•

Las etiquetas del **goto** debe ser únicas dentro de una función.

- Los nombres de **class**, **struct**, **union** y **enum** debe ser únicos dentro del bloque en que han sido definidos.
- Los nombres de los miembros de **class**, **struct** y **union** deben ser únicos en su estructura.
- Las variables **typedef**, **funciones** y miembros de **enum** deben ser únicos en el ámbito que han sido definidos.

2) Visibilidad: Es la región del programa fuente donde puede ser accedido un identificador. Generalmente, el ámbito y la visibilidad coinciden, pero existen casos de **ocultamiento** por la existencia de un identificador duplicado. El primero existe aún pero no puede ser accedido hasta que no finaliza el identificador duplicado. El nombre oculto de un objeto, función, tipo o enumerador con ámbito archivo se puede usar si se le cualifica con el operador unario **::**. Por otra parte, el nombre de una clase queda oculto por el nombre de un objeto, función o enumerador declarado en el mismo ámbito.

3) Duración: Es el período durante el cual un identificador tiene asignado un objeto físico y real de memoria. Puede ser:

- Estática: Los objetos son asignados a memoria y permanecen con esta ubicación hasta que el programa termina. Todas las funciones, las variables con ámbito de archivo y las variables con especificadores **static** o **extern** tienen esta duración.
Los objetos con duración estática son inicializados a cero (o nulo) en ausencia de una inicialización explícita.

- Local: Son objetos de duración temporal y se asignan a la pila o a un registro cuando se ejecuta el bloque o función donde figuran, desasignándose cuando termina la ejecución del bloque o función. Se llaman "objetos automáticos" y deben ser inicializados explícitamente, ya que sino contienen valores impredecibles.
Los especificadores **auto** y **register** definen objetos locales.

- Dinámica: Son objetos que se crean y destruyen mediante la llamada específica a operadores y funciones que efectúan una gestión dinámica

de memoria. Los operadores **new**, **delete** y las funciones **malloc**, **calloc** y **free** realizan esta función.
A estos objetos se les asigna espacio en la pila o heap del sistema.

4) Enlazado: Es el proceso que permite que cada identificador se asocie correctamente con un objeto o función determinado ya que puede existir el mismo identificador en diferentes ámbitos de una unidad de programa y en diferentes unidades de programa.
Todos los identificadores tienen uno de los siguientes tipos de enlace:

- **Externo**: El identificador se enlaza con el objeto o función durante la ejecución del montador de enlaces ("linkeditor"). Tienen este tipo de enlace: una función sin especificador **static**, y un objeto sin especificador de ámbito de archivo. Un especificador con especificador **extern** tiene el enlace de su variable de ámbito de archivo, y si no existe, el enlace es externo.
- **Interno**: Un identificador se enlaza con el objeto o función durante la compilación. Este tipo de enlace lo tienen los objetos o funciones con especificador **static**.
  Un nombre con ámbito de archivo declarado explícitamente **inline** o **const** (sin especificador **extern**) es local a esa unidad de programa.

- **Sin enlace**: Presentan esta característica: Un identificador con ámbito de bloque declarado sin especificador **extern**, los parámetros de una función y otros tipos de identificadores como los definidos con **typedef**.

b) Tipo: Define la estructura del objeto y su rango de variación. En otras palabras, determina el espacio de memoria asignado al objeto y el formato que se va a aplicar en las operaciones de cálculo y almacenamiento.

Un tipo de datos se puede definir como un conjunto de valores que un objeto puede asumir, junto con el conjunto de operaciones que se identifican como propias de ese tipo.

Los especificadores fundamentales de tipos de datos en C++ son:

char, **double**, **float**, **int**, **long**, **short**, **signed**, **unsigned**.

Ejemplo:

```
char x;         // x ocupa 8 bits, con rango 0 a 255
long int f;     // f ocupa 32 bits, con rango +-2147 10^9
unsigned t;     // t ocupa 16 bits, con rango 0 a 65535
```

c) Clase de almacenamiento: Define la ubicación del objeto (segmento de datos, un registro, el montón, o la pila del sistema) y su duración o tiempo de vida.

Los especificadores de clase de almacenamiento en C++ son:

auto, **extern**, **register**, **static**, **typedef**.

La clase de almacenamiento de un objeto se puede establecer por la sintaxis de la declaración, por el lugar que ocupa el objeto en el programa o por ambos.

Ejemplo:

```
static int x;    // x se ubica en el segmento de datos
register float f;       // f se ubica en un registro
```

En resumen, la declaración de un objeto viene dada, en el siguiente orden, por

- El especificador de almacenamiento
- El especificador de tipo de objeto
- El nombre del objeto

Ejemplos:

```
extern float a[];       // a es un vector float definido externamente
static int i;           // i es un entero ubicado en el segmento de datos
char *p;                // p es un puntero a un tipo char;
```

Por otra parte, los objetos pueden ser de dos tipos:

- **Simples** o **escalares**: variables, enumeraciones, punteros, referencias.

- **Compuestos** o **agregados**: arrays y clases (incluye estructuras y uniones).

En programación orientada a objetos el concepto de objeto se circunscribe fundamentalmente a los del tipo clase (**class**, **struct** y **union**).

Declaración de funciones

Las funciones son los bloques constructores de C++, donde se produce toda la actividad del programa.

Cada programa debe tener una función **main** que es la primera que se ejecuta. Las funciones son piezas de código fuente que se ubican permanentemente en memoria y se ejecutan cuando son requeridas.

Las funciones son declaradas normalmente como prototipos, en archivos de cabecera estándar o definidos por el usuario, o dentro de archivos que configuran el programa. Son declaradas como externas si no existe especificador de almacenamiento y son accesibles desde cualquier archivo de programa, si no se les declara con **static**.

Ejemplo:

```
# include <stdio.h>
int calcula(int);
void main ()
{
        int i;
        //   ...
        i = calcula(432);
        printf("%d,i);
        //   ...
}
// printf está declarada como prototipo en el archivo
// estándar de cabecera stdio.h.
// calcula está declarada como prototipo por el usuario.
```

Especificaciones de clase de almacenamiento

Especifican la ubicación del objeto (segmento de datos, registro, pila del sistema o montón) y su duración o tiempo de vida. Una función siempre se ubica en el segmento de código y su duración es estática.

Los especificadores de almacenamiento en C++ son:

auto, **extern**, **register**, **static** y **typedef**.

a) auto: (Sólo para objetos). Se usa con objetos de ámbito local, por lo que tienen una duración temporal (nacen y mueren en el bloque donde se declaran). Se aplica por defecto en las declaraciones de ámbito local.

Este especificador sólo se puede usar en nombres de objetos declarados dentro de una función o bloque, y por argumentos formales. Un objeto automático se suele crear en la pila, pero puede ser convertido por el compilador en una variable registro.

Ejemplo:

```
if (!x) {auto int i;/* ... */}
cout << i;       //sentencia no válida, i es local a su bloque
```

El declarador **auto** es casi siempre redundante por lo que no se suele usar. Un uso que se le da es para distinguir claramente una declaración de una sentencia expresión.

Ejemplo:

```
float (*pf)[10];
// declaración del puntero pf a un vector de 10 valores
// reales.
auto float (*pf)[10];
//la misma declaración pero hecha más explícita
```

b) register: (sólo para objetos). Indica al compilador que intente almacenar el objeto en un registro. Se usa con objetos del mismo ámbito que los declarados con auto; es decir, dentro de una función o bloque y con los parámetros de funciones. Si el compilador de C++ no puede ubicar ese objeto en un registro, lo tratará como si fuera especificado con **auto**.

Ejemplo:

```
void f(register int i)
// i es un parámetro entero de tipo register
{int y; y = i;/* ... */}
// y es una variable local de tipo auto
```

A diferencia de C, C++ permite asignar la dirección de un objeto declarado con el especificador **register**. En tales casos, el compilador ubica el objeto en un almacenamiento representable por una dirección.

Ejemplo:

```
// es válido en C++
register r;
int *p = &r;
// p apunta al objeto r.
```

c) static: (para objetos y funciones). Indica almacenamiento en el segmento de datos. Se usa con objetos de ámbito de archivo o bloque, pero no se puede usar con los parámetros formales de una función. El tiempo de vida de un objeto estático es el del propio programa; es decir, desde que se inicia su declaración hasta su terminación.

Un objeto declarado como estático tiene visibilidad: a) del archivo (desde el punto de su declaración), si se declara fuera de una función; b) de la función, si se declara dentro de una función y fuera de un bloque; c) del bloque, si se declara dentro de un bloque.

Los objetos así declarados son inicializados a cero (o NULL en el caso de punteros) en ausencia de inicializador.

Ejemplos:

```
static i = 20;  // i tiene visibilidad de archivo
void fun (float f)
{static int j;j++;--f;}
// j tiene visibilidad de función
```

Las funciones declaradas con static no tienen visibilidad externa, sólo en ese archivo, y tienen duración estática (todas las funciones la tienen).

Ejemplos:

```
static void f(int x) {/* ... */}
// f sólo es visible en el archivo en que está declarada
```

d) extern: (para objetos y funciones). Si se usa con objetos globales (con ámbito de archivo) se quiere señalar que esos objetos están declarados en otros archivos. Si se usa con objetos con ámbito de función o bloque, se especifica que esos objetos están declarados en el mismo archivo de la función o bloque, o en otro archivo.

Cuando un objeto con ámbito de archivo no lleva especificador de almacenamiento, se trata de un "objeto global" que es accesible

desde cualquier otro archivo (precisa la declaración **extern** si se usa en otro archivo).

Ejemplo:

```
//fich1.cpp              //fich2.cpp
int v = 64;              extern int v;
// ...                   // ...
// v es una variable global accesible desde cualquier
// archivo.
```

Las variables globales con visibilidad externa tienen duración estática y son inicializadas a cero en ausencia de inicializador.

En el caso de funciones, el especificador extern puede preceder a su declaración. Indica que esa función está definida en otro archivo (aunque puede estar definida en ese archivo) y que su enlazado es externo, siendo visible desde otros archivos.

Ejemplo:

```
//Fich1.cpp
extern void f(int);
// ...
f(20);
// Fich2.cpp
void f(int x) {/x...x/}
// f es accesible desde cualquier archivo
```

Cuando una función no lleva especificador de almacenamiento su enlazado es externo (**extern**) y por lo tanto accesible desde cualquier archivo del programa.

Cabe señalar que una declaración de un objeto es una definición a menos que lleve el especificador **extern** y no se inicialice. Por lo tanto, una declaración no afecta a una definición a menos que genere alguna inconsistencia.

Ejemplo:

```
static float f;
float f; // error: f ya está definido
extern int g;
static int g; // error: el enlace de g como interno
                // es inconsistente con el declarado
                // anterior como externo
static float h;
extern float h; // correcto. Domina la declaración y
                  // se asume para h enlazado interno
```

La siguiente tabla muestra un resumen de la accesibilidad y duración (de un objeto y una función), en función del especificador de almacenamiento y del lugar donde se ubiquen en el archivo.

e) typedef: Sirve para definir un nuevo especificador de tipo de dato más que para declarar un objeto. Se incluye como un especificador de clase de almacenamiento a causa de su sintaxis más que por su similaridad funcional.

Ejemplo:

```
typedef long int LI;
// Permitiría declaraciones del tipo:
LI x,y;          // long int x,y;
extern LI i,j;   // extern long int i,j;
```

Typedef permite que un identificador sea un sinónimo o alias de un tipo arbitrario, y que el identificador pueda ser usado como si fuera un especificador de tipo de dato en las subsiguientes declaraciones. Es especialmente útil para simplificar declaraciones complejas:

```
typedef float (*PF)(int);
PF x,y;
// x e y son punteros a funciones con un argumento entero
// que devuelven un valor real
```

No se puede usar typedef con otro especificador de tipo de datos.

```
signed LI z;   // es un error
```

Tabla: Accesibilidad en función del especificador de almacenamiento

ESPECIFICADOR	NIVEL DE UBICACIÓN	CLASE ELEMENTO	ACCESIBILIDAD	DURACIÓN	UBICACIÓN EN EL SISTEMA
AUTO	ARCHIVO	OBJETO/ FUNCIÓN	NO PERMITIDO	----------	----------
	FUNCIÓN	OBJETO	FUNCIÓN	LOCAL	PILA/REGISTRO
	BLOQUE	OBJETO	BLOQUE	LOCAL	PILA/REGISTRO
	PARÁMETROS FORMALES	OBJETO	FUNCIÓN	LOCAL	PILA/REGISTRO
REGISTER	IGUAL QUE **AUTO**	IGUAL QUE **AUTO**	IGUAL QUE **AUTO**	IGUAL QUE **AUTO**	REGISTRO/PILA
STATIC	ARCHIVO	OBJETO/ FUNCIÓN	ARCHIVO	PERMANENTE	SEG.DATOS/ CÓDIGO
	FUNCIÓN	OBJETO/ FUNCIÓN	FUNCIÓN	PERMANENTE	SEG.DATOS/ CÓDIGO
	BLOQUE	OBJETO/ FUNCIÓN	BLOQUE	PERMANENTE	SEG.DATOS/ CÓDIGO
EXTERN	ARCHIVO	OBJETO	DECLARADO EN OTRO ARCHIVO (EL PROGRAMA)	PERMANENTE	SEGMENTO DATOS
	ARCHIVO	FUNCIÓN	TODO EL PROGRAMA	PERMANENTE	SEGMENTO CÓDIGO
	FUNCIÓN	OBJETO/ FUNCIÓN	DECLARADO EN ESE U OTRO ARCHIVO (DEPENDE)	PERMANENTE	SEGMENTO DATOS/CÓDIGO
	BLOQUE	OBJETO/ FUNCIÓN	DECLARADO EN FUNCION, ARCHIVO U OTRO ARCHIVO (DEPENDE)	DEPENDE DE SU DECLARACIÓN	DEPENDE DE SU DECLARACIÓN
SIN ESPECIFICAR	ARCHIVO	OBJETO GLOBAL	TODO EL PROGRAMA	PERMANENTE	SEGMENTO DATOS
	ARCHIVO	FUNCIÓN	TODO EL PROGRAMA	PERMANENTE	SEGMENTO CÓDIGO
	FUNCIÓN/ BLOQUE	OBJETO	AUTO	LOCAL	PILA/ REGISTRO
	FUNCIÓN/ BLOQUE	FUNCIÓN	ESA FUNCIÓN O ESE BLOQUE	PERMANENTE	SEGMENTO/ CÓDIGO

Especificaciones de tipos de datos

En C++, los tipos pueden ser divididos en fundamentales y derivados:

- Fundamentales: **char**, **double**, **float**, **int**, **short**, **long**, **signed**, **unsigned**, **enum** y **void**.

- Derivados: punteros, arrays, funciones, **class**, **struct**, **union**.

Tipos fundamentales

Los definen los tipos: enteros, reales e incompletos. Los enteros y reales definen el tipo artimético.

a) Tipos enteros: Los especificadores **char**, **short**, **int** y **long**, junto con sus variantes **signed** y **unsigned** configuran los tipos enteros. También se incluye el tipo **enum** dentro de este grupo.

Especificadores	rango mínimo
char	[0,128)
signed char	(-128,128)
unsigned char	[0,256)
short, signed short	$(-2^{15},2^{15})$
short int, signed short int	$(-2^{15},2^{15})$
unsigned short, unsigned short int	$[0,2^{16})$
int, signed, signed int	$(-2^{15},2^{15})$
unsigned int, unsigned	$[0,2^{16})$
long, signed long	$(-2^{31},2^{31})$
long int, signed long int	$(-2^{31},2^{31})$
unsigned long, unsigned long int	$[0,2^{32})$

Estas variaciones son según el ANSI C. En Borland C++, los tipos **short** e **int** son de 16 bits y **long** es de 32.

a.1) Tipo enumeración: Pertenecen al tipo entero y se usa para proporcionar identificadores mnemotécnicos a un conjunto de valores enteros. Los identificadores usados son implícitamente del tipo **unsigned char** o **int**, dependiendo del valor de los enumeradores (un identificador con valor negativo exige tipo **int**).

```
enum elementos {mesa,silla,escritorio} e1;
enum elementos a,b,c;
enum {mesa,silla,escritorio} x,y;
// e1,a,b,c,x,y son variables enum
// mesa, silla, escritorio son constantes de
// enumeración
```

En C++ una variable enum solo puede recibir valores de las lista de enumeración:

```
e1 = mesa;          // correcto
e1 = 1;             // incorrecto
```

En C++ se puede omitir la palabra reservada **enum** si el identificador no crea conflicto con otro objeto del mismo nombre.

```
enum elementos {mesa,silla,escritorio};
elementos e1;
```

La lista de enumeración está constituida por constantes de enumeración, que en ausencia de inicializadores explícitos, asumen los valores enteros 0, 1, 2 ... Si algún enumerador tiene asignado un valor, adquiere dicho valor, en caso contrario adquiere el valor entero sucesor del enumerador anterior.

```
enum dias {lun,mar=5,mier,juev=mar+lun,vier,sab,dom=-4);
```

asigna a las constantes de enumeración, los siguientes valores:

```
lun = 0, mar = 5, mier = 6, juev = 5, vier = 6,
sab = 7, dom = -4
```

Los inicializadores pueden ser valores enteros positivos, negativos y expresiones. Pueden tener valores duplicados.

Cualquier objeto de tipo **enum** puede aparecer donde está permitido un objeto de tipo **int**.

```
enum elementos {mesa,silla,escritorio};
enum elementos e1, *pe1 = new elementos;
e1 = escritorio;
*pe1 = mesa;
int i = silla;
```

Las variables enumeradas comparten el mismo espacio que los identificadores de las variables ordinarias, el identificador del enumeración comparte el espacio de los tipos **struct** y **union**.

```
enum elementos a,b;
struct elementos {int x; float y}; // error
struct s {int elementos; double d;}; // correcto
```

Con excepción de la asignación (=), no se admiten operaciones con variables enumeradas. La razón para esto es que el resultado de tales operaciones puede no estar definido en la lista de enumeración.

***b) Tipos reales*:** El C++ define tres tipos reales: float, double y long double. Su representación y rango de variación dependen de la implementación de C++ usada. Veamos los rangos mínimos y restricciones en la representación especificados por ANSI C.

Tipo	Rango mínimo	Restricciones de la representación
float	$[-10^{38},-10^{-38}]$ y 0 y $[10^{-38},10^{38}]$	Al menos 6 dígitos de precisión
double	igual que **float**	A menos 10 dígitos decimales
long double	igual que **float**	Al menos 10 dígitos decimales

En Borland C++, **float** ocupa 32 bits (3.4×10^{-38} a $3.4 \times 10^{+38}$ y 7 dígitos de precisión), **double** es de 64 bits (1.7×10^{-308} a $1.7 \times 10^{+308}$ y 15 dígitos de precisión) y **long double** ocupa 80 bits (3.4×10^{-4932} a $3.4 \times 10^{+4932}$ y 19 dígitos de precisión).

Ejemplos:

```
float fl = 3.1416;
double d = 14.34678952;
long double ld = 1.48E480;
```

c) Tipos void: corresponde al tipo incompleto e indica la ausencia de cualquier valor. Se usa cuando:

1o.) La lista de parámetros está vacía en una declaración de función.

```
float f (void);          // la función f no tiene argumentos
```

2o.) La función no devuelve ningún valor

```
void f(int i);           // Se la podría llamar con f(20), ejemplo
```

3o.) Se declara un puntero genérico a cualquier cosa.

```
void *ptg;
int *pti;
char ptc;
ptg = ptc;               // correcto
ptg = pti;               // correcto
ptc = pti';              // incorrecto
```

4o.) A una expresión se le descarga de su valor calculado o a una función de su valor de retorno.

```
extern float f (void);   // devuelve un valor real
(void) f();              // Elimina el valor real de retorno
```

Modificadores

En algunos casos es necesario alterar algunas características del identificador de una declaración usando para ello un modificador.

Aunque cada implementación dispone de sus propios modificadores, los más usuales en C++ son los modificadores **const** y **volatile**.

a) const

Este modificador transforma una variable simbólica en una constante simbólica, impidiendo cualquier tipo de asignación al objeto u otros efectos laterales tales como incrementos o decrementos.

Dado que una variable **const** no puede ser modificada, después de definida debe ser inicializada.

En el caso de punteros, el modificador **const** puede preceder al tipo al que apunta, a la variable puntero o a ambos. El elemento a quien preceda no podrá sufrir modificaciones.

```
const int a = 20;                  // a++ es ilegal
const b = 512;                     // const int b = 512
const double = d;                  // error, d no fue inicializada
char *const lit = "ejemplo";       // puntero constante
lit++;                             // error
lit[6] = 'a';                      // correcto
const char *lit1 = "suma"          // puntero a una constante
lit1++;                            // correcto
lit1[3] = 'o';                     // error
char const *lit2 = "valor";        // const char *lit2 = "valor"
```

Como es lógico, un puntero a una constante no puede ser asignado a un puntero a una variable, dado que se podría modificar el contenido de la constante a través del puntero a la variable.

```
char *lit3 = lit1;                 // error
```

Es normal definir los argumentos y fórmulas de las funciones como punteros a objetos constantes cuando se transmite una dirección o referencia y no se quiere que se modifique el contenido del objeto.

b) Volatile

Un objeto es declarado **volatile** cuando es posible que su valor sea modificado fuera del control del compilador, por ejemplo, por una rutina de interrupción o un port de entrada/salida.

Declarando un objeto como **volatile** se notifica al compilador que no tenga en cuenta el valor del objeto mientras evalúa expresiones que lo contienen, dado que ese valor puede cambiar en cualquier momento. Así, el compilador, no

debería optimizar el código que contiene el objeto (por ejemplo, almacenándolo en un registro).

Ejemplo:

```
#include <stdio.h>
#include <dos.h>
volatile int ts = 0;
void interrupt tiempo (...)
{
        ts++;
}
void espera (int intervalo, int inum)
{
        while (ts < intervalo)
        {
                geninterrupt (inum);
                printf ("%d %d\n",ts,intervalo);
        }
}
void install (void interrupt (*faddr)(...),int inum)
{
        setvect (inum,faddr);
}
void main ()
{
        install (tiempo,10);
        espera (4,10);
        printf ("Hola\n");

}
// Salida:
// 1 4
// 2 4
// 3 4
// Hola
```

La función *tiempo* es una rutina de interrupción que, cuando se la llama, incrementa la variable **volatile** ts en una unidad.

La función *espera* genera interrupciones, llamando a la función *tiempo*, mientras ts es menor que intervalo.

La función *install*, instala en el vector de interrupciones no. 10 de la memoria del sistema la función *tiempo*.

Si no se pone **volatile** en ts, un compilador con una alta optimización podría no incorporar el valor de ts dentro de la sentencia **while**, dado que aparentemente esta sentencia no cambia el valor de ts, produciendo una ejecución errónea.

Tipos derivados

a) Punteros

Son variable que contienen la dirección del objeto o función al que apuntan. Es decir, objetos de datos cuyo valor es la dirección de almacenamiento que el programa usa para designar otro objeto de datos o función (se puede apuntar a cualquier elemento, excepto a un "campo de bits" que se define en las clases (**class**, **struct** o **union**)).

a.1) Punteros a objetos

Se declaran igual que los otros objetos de C++, excepto que un objeto puntero va precedido de una asterisco (*).

```
int *pi;              // pi es un puntero a un objeto entero
long double *pd;      // pd es un puntero a un objeto de 10 bytes
```

Un puntero puede ser reasignado para que apunte a un objeto de un tipo diferente al declarado, pero el compilador emitirá un aviso a menos que el puntero sea declarado del tipo **void**.

```
void *pv;
int *pi;
float *pf;
pv = pi;        // correcto
pf = pi;// error
pf = pv;        // error. En C está permitido
pi = 0;         // puntero nulo
pi = NULL;      // puntero nulo
// NULL es una macro definida en el archivo standard stdio.h
// de la librerías del C++
```

. ***Artimética de punteros***. La artimética de punteros está limitada de forma restringida a las operaciones de *suma*, *resta* y *comparación* y tiene en cuenta la dimensión del objeto al que apunta.

```
int i,*pi;
pi = &i;              // pi recibe como valor la dirección de i
pi++;                 // incrementa pi en 2 unidades (el tamaño de un
                      // int)
double d,*pd;
void *pv;
pv = &i;
pv++;                 // no permitido pues es un puntero genérico
((int *)pv)++         // incrementa pv en 2 unidades
pd = &d;
pd++;                 // pd = pd + 8
```

Si un puntero apunta a un array de objetos, un incremento en el puntero significa un incremento equivalente al **tipo** del array, no de todo el array.

```
char ac[5], *pc;        // ac es un vector de 5 caracteres
int ai[10], *pi;        // ai es un vector de 10 enteros
pc = ac;                // pc apunta a la posición ac[0]
pc++;                   // pc = pc + 1 byte
                        // pc apunta a la posición ac[1]
pi = ai;                // pi apunta a la posición ai[0]
pi ++;                  // pi = pi + 2 bytes
                        // pi apunta a la posición ai[1]
```

Un puntero pv que apunta al último elemento de un array, admite la operación pv + 1, pero es indefinido el valor pv + 2 porque se sale del array.

Si p1 es un puntero al inicio de un array y p2 es un puntero al final del array, la operación p2 - p1 proporciona un entero igual a la longitud del array.

La artimética de punteros requiere que un puntero apunte a un tipo de objeto, por lo que si el puntero está declarado como **void** no se puede aplicar ninguna operación artimética. La causa es que para incrementar o decrementar el compilador debe conocer el tamaño del objeto al que se apunta. En el caso de **void**, ese tamaño es deconocido.

Ejemplo:

```
void *pun;
// ...
pun--;// error
```

. ***Punteros constantes***: Un puntero puede ser declarado con el modificador **const**. Un puntero (y cualquier objeto) declarado con **const** debe ser asignado pues no puede ser modificado.

Ejemplo:

```
float a;
float *const pi = &a;
// pi es una constante puntero que apunta a la variable
// real a
pi = &a;                        // Ilegal
const int b = 10;               // b es una constante entera
b++;                            // Ilegal
const float *const pf = &a;
// pf es una constante puntero que apunta a una cte. real
pf++; (*pf)++;                  // Ilegales
```

a.2) Punteros a funciones

Se usan para acceder a funciones y pasar estas funciones como argumentos de otras funciones.

Se declaran como se declara el prototipo de una función con la singularidad de que un puntero a una función va precedido por un asterisco (*).

```
int (*pf)(int,float);
// pf es un puntero a una función con dos argumentos
// (entero y real) que retorna una valor entero

int (*pf) void;          // pf es un puntero a una función sin
                         // argumentos que devuelve un puntero
                         // a un entero.
```

No está permitida una artimética de punteros a funciones. Así pf++ sería ilegal.

Para acceder o llamar a la función a la que apunta, se utiliza también su nombre precedido por un asterisco.

```
double sin(double);          //prototipo de la función seno
double (*ptr)(double);       //puntero a función que toma como
                             // argumento un double y retorna otro
                             // double
double v;
ptr = sin;
v = (*ptr)(2.0);             // equivale a v = sin(2.0)
```

a.3) Conversión de punteros ("cast")

El mecanismo "cast" permite que un objeto (variable puntero, etc.) declarado de un tipo dado pueda ser tratado por el sistema como de otro tipo diferente.

El mecanismo consiste en colocar como prefijo del objeto que se desea convertir, el tipo de dato, entre paréntesis.

Los punteros a un tipo de dato se pueden convertir en punteros a otro tipo de dato mediante el mecanismo "cast".

```
int *pi;
float *pf;
pf = (float*) pi;            // (float*) es el mecanismos de "cast"
// El valor de pi es convertido a estructura puntero a
// float antes de ser asignado a pf
```

Cuando los punteros son de tipos diferentes (y no son de tipo **void**), para su correcta asignación hay que aplicar este mecanismo.

```
int (*pfi)(float,int)
float *pf;
pf = (float*) pfi;
// se asigna un puntero a una función a un puntero a un
// objeto float
```

b) Arrays

Son colecciones de objetos del mismo tipo, contiguos en memoria, que se referencian con un nombre común. A un elemento específico de un array se accede con el nombre del arreglo y uno o más subíndices que dan la posición del elemento, teniendo el primer elemento del arreglo el índice 0.

Cada elemento del array se numera de 0 a dim-1. Los arrays multidimensionales se construyen declarando arrays de tipo array.

Ejemplo:

```
char p[10];          // vector de caracteres: p[0] a p[9]
int b[3 * 4];        // vector de enteros: b[0] a b[11]
float x[6][7];       // matriz de 6 filas por 7 columnas
```

b.1) Características de un array

. Una declaración **extern** de un array no necesita la dimensióne exacta del array (es un tipo incompleto). Pero si el array es multidimensional se necesita conocer las dimensiones de los arrays subordinados. Es decir, se puede omitir la primera dimensión.

```
extern float y[];          // vector y definido en otro ámbito
extern int x[][20];        // matriz x de 20 columnas, definida
                           // en otro ámbito
```

. A un elemento y de un array x se puede acceder por x[y] o por *(x+y).

```
char t[80];          // t==&t[0]
char c = t[10];      // char c = *(t+10)
```

. En el caso de una array multidimensional, en el que su nombre aparece en una expresión, éste es convertido en puntero a la primera fila del array al que se suma tantas filas cuanto sea el valor del primer subíndice del arreglo. El resultado es un puntero a una determinada fila al que se suma tantas columnas como sea el valor del segundo subíndice del array, y así sucesivamente.

```
float a[4][7][8];
// a[3][6][7] == *(*(*(a+3)+6)+7)
```

. Un array se almacena en memoria por filas, o si es un arreglo multidimensional, variando más rápidamente el índice más externo.

. Un puntero a la dirección del primer elemento de un array se puede tratar como el propio arreglo y se puede modificar su dirección. En un arreglo no se puede modificar su dirección.

```
int c[5],x;
int *cp = c;
x = c[2];          // x = cp[2]
c++;               // ilegal
cp++;              // legal
```

b.2) Inicialización de un array

En C++ se puede inicializar un arreglo con cualquier especificador de almacenamiento: **extern**, **auto**, **static** o **register**.

1) Como una lista: Usando una lista de inicializadores para los elementos del arreglo incluida entre llaves {}. Si un elemento no tiene valor de inicialización se rellena con un cero del tipo apropiado.

```
int tab1[3][2] = {{1,2},{3},{4,5}};
// tab[1][1] = 0
int tab2[3][2] = {1,2,3,4,5,6};
// tab[1][1] = 4
char t[3] = {'a','b','c'}
// t[0] = 'a'
```

2) Como un literal: Usando una tira de caracteres, opcionalmente incluida entre llaves. La inicialización incorpora el caracter nulo ('\0') al final de la tira.

```
char t1[] = "hola"          //t1[0]='h',...,t1[4]='\0'
char t2[] = {"hola"}        // igual que t1
char t3[] = {'h','o','l','a'}   // t3[0]='h',...t3[4] no existe
```

. Si no se incializa un arreglo y tiene una duración estática (declarado con **static** o es un objeto global), se inicializará, por defecto, a cero si es de tipo artimético o a NULL si es de tipo puntero.

b.3) Arrays de punteros

Cada elemento del arreglo es un puntero a otro objeto. Este tipo de arreglos son muy útiles, por ejemplo, para definir textos en formato de página con logintud de línea variable.

```
char *ptiras[] = {"primera","segunda","tercera"};
// *(ptiras[0]=='p';*(ptiras[2]+2)=='r'
```

En algunas implementaciones, los arrays de punteros se pueden inicializar de forma que cada puntero apunte al primer elemento de un array del tipo asociado.

```
int i[] = {10,5,4};
int j[] = {1,2,3,20,25};
int *p[] = {i,j};           // p[1][2] == 3
```

c) Funciones

Una función es un bloque con nombre que incorpora un código ejecutable con un único punto de entrada (el nombre de la función) y uno o varios puntos de retorno.

En una función interesa diferenciar su declaración y su definición, aunque como en cualquier objeto, se puedan realizar ambas acciones a la vez.

c.1) Declaración de una función (Prototipado)

Especifica las características de la función en lo que se refiere a su tipo de almacenamiento, tipo de valor de retorno y características de los parámetros formales.

Permite al compilador verificar si la llamada a una función tiene el tipo y el número de argumentos apropiados.

```
int f(void);
// declara una función sin argumentos que devuelve un entero
f f1(float);                    // no permitido
void f2(char*,int,float);
// declara una función con tres argumentos
// (puntero a char, entero y real) que no devuelve nada
```

c.2) Características de la declaración de parámetros

Las características más relevantes son:

- La declaración de parámetros formales sigue una sintaxis similar que la del resto de los objetos C++.

  ```
  int f1 (int i, float f);
  // los identificadores i y f son informativos
  float f2(int*,int,float);
  // f2 tiene 3 parámetros
  ```

- Los parámetros de tipo array de T se convierten a punteros a T, y los de tipo función que devuelven T se convierten a punteros a función que devuelven T.

  int f(**float**,**char**[]); //**int** f(**float**,**char***);
  float f1(**int** f2(**void**); // **float** f1(**int** (*f2)(**void**));

- Cuando el número de parámetros es variable, se usa la elipsis... (función variádica), que se coloca al final, detrás de los parámetros fijos. Estos se comprueban en tiempo de compilación, mientras que los variables se comprueban en tiempo de ejecución.

  int printf (**char*** formato...);
  // printf("%d",i); corresponde a:
  // formato == "%d", elipsis ==i

- El único especificador legal de almacenamiento es **register** (se asigna a **auto** po defecto).

  int f(**register int**,**float**);
  // el primer parámetro es entero, ubicado en un registro,
  // y el segundo es real (**float**) ubicado en la pila

- Los tipos de los parámetros pueden ser: escalares (enteros, reales, punteros), **class**, **struct**, **union**, punteros o referencias a **class**, **struct** y **union**, o punteros a funciones o clases.

  int f(**int***, **struct** pila, **union** elemento*,**int**(*f1)(**int**));
  // puntero a entero, estructura, puntero a union
  // y puntero a una función que recibe un entero y
  // devuelve un entero

- En C++ los parámetros de las funciones pueden tomar *valores por defecto*, que son asignados por el compilador si no se suministran los argumentos actuales.

  void f(**int** i = 2); // f() equivale a f(2)

  La asignación de parámetros por defecto debe figurar al final de la lista de parámetros.

  int f(**int**,**char**,**int** u = 4,**float** v = 5.2);

- Si una función se declara sin parámetros, C++ la considera una función sin argumentos (**void**), mientras que C la considera con un número variable de argumentos.

```
int *(*pf)();
// pf es un puntero a una función sin argumentos
// que devuelve un puntero a un entero (en C++)
// En C, pf es un puntero a una función con argumentos
// que devuelve un puntero a un entero
```

- Los parámetros tienen el ámbito y la duración de la función.

c.3) Definición de una función

Una definición de función declara una función (si no ha sido previamente declarada) y especifica las acciones que realiza cuando se ejecuta.

- Los identificadores son obligatorios si se usan en el cuerpo de la función.

```
int f(int x, float y) {return x + y;}
// x e y son obligatorios en la defínción
void f1(int x, int) {cout << x << end1;}
// solo es obligatorio x
// reserva un espacio en la lista de argumentos para
// futuras necesidades sin modificar la interfaz
```

- Los parámetros deben coincidir en número y tipo con los descriptos en la declaración (si existe).

- Los valores de los *parámetros por defecto* se suelen especificar aquí, en lugar de en el prototipado de la función.

```
char *f (char c, int i = 24, float j = 2,8){/* ... */}
```

- El cuerpo de la función, que especifica las acciones que realiza la función, se delimita con llaves y se configura como un bloque. La función devuelve un resultado con la sentencia **return** que es su valor de retorno. En algunos casos, la función se comporta como un procedimiento que no devuelve ningún valor, siendo el valor de retorno de tipo **void**.

```
// Obtiene el valor más pequeño entre a1 y a2.
int minimo (int a1, int a2)
{
        return (a1 < a2 ? a1 : a2);
}
```

```
// Calcula el máximo común divisor
int mcd(int a, int b)
{
        int temp;
        while (b)
        {
                temp = b;
                b = a % b;
                a = temp;
        }
        return a;
}
// imprime un vector descendentemente
void imprimir (int a[], int n)
{
        int i,j,temp;
        for (i = 0; i < n; i++)
        {
                for (j = i+1;j <= n;j++)
                        if (a[i] < a[j])
                        {
                                temp = a[i];
                                a[i] = a[j];
                                a[j] = temp;
                        }
                cout << a[i] << end1;
        }
}
```

c.4) Llamada a una función

El tercer contexto en el que aparece una función es cuando se la llama. La referencia se hace con un conjunto de parámetros actuales que el sistema hace corresponder con los parámetros formales. Si el tipo de los argumentos actuales no se correponde con el tipo de los argumentos formales, el sistema los convierte al tipo de éstos últimos, siempre y cuando se trate de tipos compatibles. El número de argumentos debe coincidir con los declarados como fijos en la función (con la excepción de los asignados por defecto).

```
void f(int x,float y) {/* ... */}          // definición de f
//.....
void main ()
{
        f(4,3.22);                         // llamada a la función f
}
```

Los nombres de las funciones pueden ser usados con objetos de su mismo tipo de retorno en el cálculo de expresiones, y su valor de retorno puede ser asignado a una variable.

```
int f(int i);              // Declaración de la función f
```

```
int j = f(20) * 48;
```

c.5) Tipos de argumentos de una función

El paso de parámetros, cuando se efectúa una llamada a una función, se puede realizar de tres formas:

a) Por valor: Se hace una copia local a la función de los valores de los parámetros actuales en los formales. Su uso es equivalente a una asignación y se aplica una conversión de tipo si procede. Cuanlquier cambio que se haga en la función de estos valores, no se refleja en los valores de los argumentos actuales.

```
// funv intercambia localmente
// el contenido de dos variables
void funv (int u, int v)
{
        int temp = u;
        u = v;
        v = temp;
}
void main ()
{
        int x = 5, y = 10;
        funv(x,y);
        cout << x << ' ' << y;
        // Imprime 5 y 10
        // Los cambios realizados en u y v
        // no se reflejan en x e y
}
```

b) Por dirección: Se pasa la dirección del objeto mediante un puntero. Es en realidad una variante sobre el anterior, ya que se hace una copia del la dirección del objeto (parámetro actual) en el puntero (parámetro formal). Se usa cuando los objetos a pasar como argumentos son grandes o se desea devolver en los parámetros el resultado de alguna operación.

```
// fund intercambia el contenido de dos
// variables
void fund(int *u, int *v)
{
        int temp = *u;
        *u = *v;
        *v = temp;
}
void main ()
{
        int x = 5, y = 10;
        fund (&x,&y);
        // se envían direcciones
        cout << x << ' ' << y;
        // imprime 10 y 5
        // Realiza el intercambio
}
```

c) Por referencia: Se pasa un "alias" del objeto, de modo que el argumento actual y el parámetro formal referencien a la misma dirección de memoria.

. Una referencia se representa precediendo al parámetro formal con un &.

```
// funr intercambia el contenido de dos
// variables
void funr (int &u, int &v)
{
        int temp = u;
        u = v;
        v = temp;
}
void main ()
{
        int x = 5, y = 10;
        funr (x,y);
        // dirección de x = dirección de u
        // direccion de y = dirección de v
        cout << x << ' ' << y;
        // imprime 10 y 5
}
```

. El uso de referencias se puede hacer tanto con parámetros de funciones como con objetos en general. En este último caso, la referencia debe ser inicializada por un objeto de su mismo tipo, y no se puede cambiar para que referencie a otro objeto.

```
int j = 2;
int &rj = j;      // rj es un alias de j
rj = 4;           // equivalente a j = 4;
```

. Si la referencia se inicializa con una constante o con un tipo diferente al de la referencia, C++ crea un objeto temporal sobre el cual actúa la referencia como un "alias", no reflejándose las variaciones de la referencia en el objeto que la inicializa.

```
float x = 2.4;
int &ri = x;      // Avisa en la compilación
ri = 3;           // ri == 3, pero x == 2.4
```

c.5) Argumentos en la línea de comando

La ejecución de un programa C++ siempre empieza en la función **main**. Esta función puede proporcionar el acceso a parámetros pasados en la línea de comando de ejecución del programa, mediante el uso de dos argumentos, conocidos como *argc* ("*argumento count*") y *argv* ("*argumento value*").

- argc Es un **valor entero**, que especifica el número de argumentos existentes en la línea de comando, incluido el nombre del programa. Su valor es mayor que cero.

- argv Es un **array de punteros** a cada uno de los argumentos de la línea de comandos. argv[0] apunta al nombre del programa.

Ejemplo:

```
#include <stdio.h>
void main (int argc, char *argv[])
{
        while (-- argc > 0)
                printf ("%s",*++argv);
}
```

Si este programa se ejecutase, desde el archivo *main.exe*, con la siguiente orden: main.exe ejemplo de línea de comando, imprimiría: ejemplo de línea de comando.

La **longitud máxima de la línea de comando**, incluido el nombre del programa, es de 128 caracteres en una PC.

Expresiones

Una expresión es una secuencia de operadores, operandos y puntuadores que especifica un cálculo.

Las expresiones son evaluadas conforme con determinadas reglas de precedencia que dependen de los operadores usados, la presencia de paréntesis y el tipo de datos de los operandos.

Según la tabla de precedencia vista anteriormente, las expresiones con operadores de prioridad **1, 3 al 13 y 16 se evalúan de izquierda a derecha**, mientras que las correspondientes a los operadores **2, 14 y 15 se evalúan de derecha a izquierda**.

El orden en que C++ evalúa los operandos de una expresión no viene especificado y es una implementación que depende del compilador, excepto cuando la prioridad de un operador específicamente así lo determine o se pueda aplicar a ese operador las reglas matemáticas asociativa y conmutativa.

```
int a,b,c,x;
// ...
x = a + b + c;
// Se calculará como x = ((a + b) + c) ó x = (a + (b + c));
x = a + b * c;
// Se calculará como x = (a + (b * c))
```

Se debe tener especial cuidado con las expresiones en las que se modifica el valor de una variable más de una vez.

```
int a,b,cont[10];
//...
a = cont[a++]          // a es indefinido
b = (a = b) - (--a)    // a y b son ambiguos
```

Al igual que cualquier lenguaje, se usan los paréntesis para forzar el orden de evaluación de una expresión.

lvalue y rvalue

Las expresiones pueden producir un "lvalue", un "rvalue" o ningún valor, y pueden o no causar un efecto lateral cuando se calcula su valor.

Un ***"lvalue"*** es una expresión de referencia a un objeto (como por ejemplo su nombre) o a una función.

```
float f = 3.14;
int i = 34;
double d = 434.67845;
```

*e , es también una expresión "lvalue", si e es una expresión que proporciona un puntero no nulo.

Existen diferentes **tipos de expresiones "lvalue"**:

- expresión "lvalue" accesible: Sólo permite acceder al objeto:

```
const int x = 28;              // x no puede ser modificado
```

- expresión "lvalue" modificable: El objeto puede ser accedido y modificado. Una expresión "lvalue" es modificable si no es el nombre de una función, array o **const**.

```
int x = 28;
++ x;
```

- expresión "lvalue" array: Cuando el objeto es una array.

```
int x[ ] = {1,8,2,4};
```

Una expresión ***"rvalue"*** referencia el contenido de un objeto. En general, el compilador no puede hallar su valor antes de que el programa se ejecute.

```
int a,b;
// ...
int x = a + b;  // a + b es una expresión "rvalue"
```

No obstante, existen **expresiones "rvalue" que se calculan antes de la ejecución**, tales como:

- expresión "rvalue" de direccionamiento constante:

```
int x;
int *px = &x;  // &x se calcula antes de la ejecución
```

- expresión "rvalue" artimética constante:

```
# define PI 3.1416
int x = PI / 0.301 + 0.5;
// PI/0.301 + 0.5 es una expresión artimética constante
```

- expresión "rvalue" entera constante:

```
int a[20],x [sizeof a/sizeof a[0] ];
// sizeof a / sizeof a[0] es una expresión "rvalue"
// entera constante
```

Los términos "lvalue" y "rvalue" provienen históricamente de la operación de asignación, donde el operando de la izquierda ("left value") debe referenciar un objeto, y el de la derecha ("right value") especifica un valor calculado.

Conversión de tipos en expresiones

Cuando en una expresión, los operandos son de distintos tipos, C++ efectúa una conversión interna antes de evaluar la expresión. Fundamentalmente, esta conversión trata de pasar los tipos de más bajo orden de precedencia a los de más alto a fin de **aumentar la precisión y consistencia del resultado obtenido**.

El mecanismo usado para forzar la conversión recibe el nombre de ***cast***, que presenta dos tipos alternativos:

a) (<nombre-tipo>)<expresión>
b) <nombre-tipo>(<expresión>) // éste es específico del C++

```
int i = 20;
printf ("%f\n",(float)i);
// convierte el valor de la variable i a float
cout << float(i);
// convierte el valor de i a float antes de su impresión
```

El uso del ***cast*** tiene el mismo efecto que si la expresión fuera asignada a una variable temporal del tipo requerido y esa variable se usara en lugar de la expresión original.

La siguiente tabla muestra la conversión estándar de tipos que realiza el Borland C++:

op2/op1	(1) **int**	**float**	**double**	**long**	**long double**	**unsigned long**	**unsigned**
(1) **int**	int	float	double	long	long double	unsigned long	unsigned
float	float	---------	double	float	long double	float	float
double	double	double	----------	double	long double	double	double
long	long	float	double	----------	long double	unsigned long	long
long double	long double	long double	long double	long double	----------	long double	long double
unsigned long	unsigned long	float	double	unsigned long	long double	----------	unsigned long
unsigned	unsigned	float	double	long	long double	unsigned long	----------

(1) **char**, **unsigned char**, **signed char**, **short** y **enum** son convertidos previamente a int.
unsigned short se convierte previamente a **unsigned int**.

Semántica de los operadores

Los operadores de C++ se agrupan como sigue:

a) Operadores prefijo y posfijo: Son [], (), ., ->, ++ y --

. **Operador de subíndice de un array [ ]**: Referencia un elemento de un arreglo.

```
int a[20];              // Aquí es un delimitador
a[2] = b * c;           // Aquí es un operador de subíndice
p = a;
printf("%d",p[2]);      // es un operador de subíndice
```

. **Operador de llamada a una función ( )**: Referencia a una función.

```
void f(int,float,char);       // aquí es un delimitador
f(20,4.53,'C');               // aquí es un operador
```

. **Operador miembro de una clase (class,struct o union) .**: Referencia un miembro de un objeto de tipo clase.

```
struct s {int a;float f; } s = {5,3.14};
int j = s. a;     // j == 5
```

. **Operador puntero de una clase ->**: Referencia un miembro de un objeto puntero a un clase.

```
Class X
{
        int i,j;
public
        void print(int);
        // ...
};
X *px;
// ...
px -> print(40);
// ejecuta el método print del objeto apuntado por px
```

. **Operador prefijo/posfijo incremento ++ (o decremento --)**:

```
int a = 10; b;
b = a++;                // b == 10, a == 11
b = a++;                // b == 12, a == 12;
while (*pa++ = *pb++);
// copia el contenido apuntado por pb en pa hasta que
// detecta el caracter nulo que también lo copia
```

b) Operadores Unarios: Son &, *, +, -, ~, ! (además de ++ y --).

. Operador de dirección &: Referencia la dirección de un objeto o función.

```
int *pt, i = 20;
int f(int,float);
int (*pf)(int,float);
```

```
pt = &i;                    // *pt == 20
pf = f;
```

. Operador de indirección *: Referencia a un objeto o función

```
int i = 20,j;
int *pt = &i;// aquí * es un delimitador
j = *pt; // aquí * es un operador de indirección (j==20)
```

El resultado de la operación de indirección queda indefinido si la <expresión-cast> es un puntero nulo, o es la dirección de una variable automática y ha finalizado la ejecución de su bloque.

. Operadores unarios + y -: Multiplica al operando por +1 o por -1.

```
int i = 20,j;
j = -i;          // j == 20
```

. Operador de complemento a uno ~: Es un *not* a nivel de bit.

```
int i = 2,j;
j = ~i;  // j == -3
```

. Operador de negación lógica !: Es un *not* lógico.

```
int i = 2,j;
j = !i;          // j == 0
```

c) Operadores artiméticos binarios: Son +, -, *, /, %

Estos operadores efectúan las operaciones artiméticas de suma (+), resta (-), multiplicación (*), división (/) y resto (%).

El operador + admite como operandos legales (op1 y op2) ya sea a) ambos de tipo aritmético o bien b) op1 de tipo entero y op2 de tipo puntero a un objeto o viceversa.

El operador - admite como operandos legales a) ambos de tipo artimético, b) ambos de tipo puntero a objetos compatibles y c) op1 de tipo puntero a un objeto y op2 de tipo entero.

Los operadores * y / solamente admiten operandos de tipo artimético.

EL operador % admite solamente ambos operandos de tipo entero.

d) Operadores de manipulación de bits binarios: Son >>, <<, &, |, ^.

. Operador <<: Efectúa un desplazamiento de op2 bits hacia la izquierda en el operando op1, rellenándole con ceros por la derecha. op1 y op2 deben ser enteros. Si op2 es negativo o mayor o igual al ancho en bits de op1, la operación es indefinida.

. Operador >>: Efectúa un desplazamiento de op2 bits hacia la derecha en el operando op1, rellenándole con ceros por la izquierda si op1 es de tipo **unsigned** y con el bit del signo (0 ó 1) si op1 es de tipo **signed**. Si op2 es negativo o mayor o igual al ancho en bits de op1, la operación es indefinida.

. Operador &: Efectúa un "and" de bits (1 & 1 == 1, el resto de las operaciones dan cero como resultado). Ambos operandos deben ser enteros.

. Operador |: Efectúa un "or" de bits (0 | 0 == 0, el resto de las operaciones dan 1 como resultado). Ambos operandos deben ser enteros.

. Operador ∧: Efectúa un "xor" de bits (0 ∧ 1 y 1 ∧ 0 == 1; 0 ∧ 0 y 1 ∧ 1 == 0). Ambos operandos deben ser enteros.

```
int i = 3, j = 2;
i = i << 1;                    // i == 6
i = i & j;                     // i == 2
i = i ∧ j;          // i == 0
```

e) Operadores de relación: Son: <, >, <=, >=, ==, !=

En la operación *op1 operador op2*, los operandos se deben ajustar a las siguientes condiciones:

- op1 y op2 son de tipo aritmético
- op1 y op2 son punteros a tipos de objetos compatibles
- op1 y op2 son punteros a tipos incompletos compatibles

El resultado es de tipo **int** y toma el valor 1 si se cumple la relación y el valor 0 en caso contrario.

Cabe señalar que los operandos == (igual) y != (distinto) tienen más baja prioridad que el resto, y pueden comparar ciertos tipos de punteros que no estarían permitidos para los otros. Por ejemplo, si op1 es un puntero y op2 es una constante puntero NULL.

```
int x[4]={1,2,3,4}, *pint;
int * p = x;
p++;
pint = x + 2;
if (pint-p < 2)
  cout << *pint << *p;          // imprime 3 y 2
```

f) Operador condicional: (op1 **?** op2 **:** op3)

Donde op1 debe ser de tipo escalar (tipo artimético o puntero) y los otros dos operandos deben ajustarse a las siguientes reglas:

- Ambos de tipo artimético, de tipo **void**, de tipo puntero a tipos compatibles o de tipo **class**, **struct** o **union**.

- Un operando de tipo puntero y el otro una constante puntero no nula.

- Un operando de tipo puntero a un objeto o a un tipo incompleto y el otro de tipo puntero a **void**.

El resultado de op1 ? op2 : op3 será el valor de op2 o de op3 dependiendo de que el valor de op1 sea diferente de cero (cierto) o igual a cero (falso).

```
int i = 9, c:
c = (i > 20) ? 3 : (i <= 10) ? 1 : 2;    // c == 1
// Devuelve el valor absoluto de i
int abs(int i)
{
        return (i < 0 ? -i : i);
}
```

g) Operadores de asignación: Son: =, +=, -=, *=, /=, %=, <<=, >>=, &=, ∧=, |=.

Efectúan una operación de asignación, compuesta, y precedida por otra operación: suma, resta, producto, cociente, resto, desplazamiento de bits, and binario, xor binario u or binario.

La asignación compuesta:

op1 op= op2 se interpreta como op1 = op1 op op2

donde op1 debe ser un "lvalue" modificable y el valor final calculado, previa su conversión al tipo de op1, es almacenado en el objeto op1.

Tanto op1 como op2 deben ajustarse a las siguientes reglas:

- op1 y op2 son de tipo artimético, de tipos compatibles **class**, **struct** o **union** o de tipos punteros compatibles.

- Un operando es un puntero a un objeto o a un tipo incompleto, y el otro es un puntero a **void**.

- op1 es un puntero y op2 una constante puntero nula.

```
int a = 20;
float f = 3.4;
a += a - f;                // a == 36
```

h) Operadores lógicos: Son && y ||

Efectúan las operaciones lógicas **and** y **or**.

Ambos operandos deben ser de tipo escalar y el resultado es de tipo int, en el caso de &&, es 1 (cierto) si ambos son distintos de cero y 0 (falso) en cualquier otro caso y en el caso de ||, es 0 (falso) si ambos son iguales a cero y 1 (cierto) en cualquier otro caso.

Ambas expresiones e evalúan de izquierda a derecha.

```
int a = 2, b = 3, c;
a = >> 2;                  // a == 0
c = a && b;                // c == 0
c = a || b;                // c == 1
```

i) Operador coma: Se usa en expresiones del tipo:

op1, op2, ..., opn

donde op1, op2... se evalúan como expresiones **void** y el operando opn da el resultado y el tipo de la expresión coma.

```
f(x,(y = 4, ++y), 2);          // f(x,5,2)
```

j) Operador sizeof: Proporciona la dimensión, en bytes, de su operando que puede ser una expresión o un tipo. El resultado es una constante entera con el valor de esta dimensión.

En un **sizeof**<expresión>, el tipo de la expresión se determina sin evaluarlo, y por lo tanto, sin efectos laterales. También se puede hacer un **sizeof**(nombre-tipo). En cualquier caso, el resultado viene dado por la siguiente tabla de acuerdo al tipo del operando.

TIPO DEL OPERANDO	No. de bytes
char (**signed** o **unsigned**)	1
short (**signed** o **unsigned**)	2
int (**signed** o **unsigned**)	2
long (**signed** o **unsigned**)	4
float	4
double	8
long double	10
enumeración	2
array	no. de bytes del array
array o función (como parámetro de una función)	dimensión del puntero al array o función
class, **struct**, **union**	no. de bytes de la clase

El operador **sizeof** no se puede utilizar con expresiones de tipo función, tipos incompletos, nombres entre paréntesis de tales tipos o con un “lvalue” que designe un objeto que sea un campo de bits.

```
long double ld[100];
cout << sizeof ld;                  // 1000
cout << sizeof (long double);       // 10
```

Sentencias

C++ proporciona un reducido, aunque suficiente, conjunto de construcciones para controlar el orden de ejecución de las instrucciones, es decir el flujo del programa. Las instrucciones se ejecutan en el orden en que aparecen en el programa fuente, salvo que se indique lo contrario mediante un salto o una selección.

Sentencias etiquetadas

Una sentencia etiquetada puede tener una de las siguientes formas:

<identificador-etiqueta> : <sentencia>
case <expresión-constante> : <sentencia>
default : <sentencia>

<identificador-etiqueta> es un identificador que srive como llamada de la sentencia **goto**. El ámbito de esta etiqueta es el de la función donde figura, por lo que no puede ser redeclarada dentro de la función, pero no interfiere con otros identificadores.

Las etiquetas **case** y **default** sólo se pueden usar en la sentencia switch que se decribe más adelante.

<sentencia> es cualquier tipo de sentencia descripta en este capítulo.

```
int i = 1;
a: b: i++;                          // válido
// ...
if (i) goto a; else goto b;    // válido
```

Sentencias de expresión

Son aquellas definidas por una expresión que termina con un punto y coma ;. Su sintaxis es:

<sentencia-expresión> :: [<expresión>];

La expresión se procesa evaluándose su contenido y efectuando todos los efectos laterales antes de ejecutarse la siguiente sentencia.

Existen cuatro tipos de setencias de expresión:

a) De asignación: Su forma general es:

<nombre-objeto> = <expresión>;
o
<nombre-objeto$_1$> = <nombre-objeto$_2$> = ... = <expresión>;

Donde nombre-objeto debe ser un lvalue" modificable y expresión puede ajustarse a cualquiera de las modalidades descriptas anteriormente para expresiones.

El valor de la expresión, después de calculada es convertido al tipo del "nombre-objeto" y almacenado en él.

La asignación puede ser simple o múltiple, y en ambos casos las reglas que gobiernan su ejecución vienen definidas por los operadores de asignación.

```
int i,j = 2, k = 3;
i = j = ++k-j;          // i==j, j==2, k==4
i = j++ + (++k-j);      // i = 2+(5-2)=5,j==3,k==5
```

b) Llamada a un función: Tiene la siguiente sintaxis:

```
<nombre-función> ([<parámetros-actuales>])
```

```
void f(int,float);
int fl(int),
//...
f(4,2.3);                       // llamada a función
int i;
i = fl(10) + 5;         // llamada a función con asignación
```

c) De incremento o decremento pre/posfijo: Su forma general es:

```
++<expresión>; | <expresión>++;
--<expresión>; | <expresión>--;
```

```
int a = 20, *pi;
--a;                    // a == 19
pi = &a;
pi++;                   // pi == &a + 2
```

d) Sentencia nula: No realiza ninguna acción. Es simplemente un ;.

Sentencia compuesta o de bloque

Es una lista de sentencias encerradas entre llaves {}. Se puede usar en cualquier parte del programa en que se pueda usar una sentencia simple. (Una declaración es una sentencia).

No debe aparecer ningún punto y coma después de la llave de cierre, ya que las llaves son puntuadores y no sentencias. La lista de sentencias puede ser vacía.

```
if (a > b)
{
        int c = a;
        a = b;
        b = c;
}
// {} contiene 3 sentencias simples
if (a >= b)
        {;}             // sentencia nula
```

```
else
        cout << "a menor que b";
// {} es una lista de sentencias vacía
```

Sentencias de selección

Las sentencias de selección, o de control de flujo, seleccionan diferentes alternativas de acción dependiendo del valor que tomen ciertos predicados.

a) if/else: Es una sentencia condicional que, según el valor de una condición o expresión condicional, ejecuta una acción entre dos alternativas posibles. Su formato es:

if <exp-cond> <st1> [**else** <st2>]

donde:

exp-cond debe ser un escalar. Si su valor es diferente de cero (predicado cierto) ejecuta <st1>, si su valor es cero (o nulo para punteros) (predicado falso) ejecuta <st2> si existe la rama else.

<st1> y <st2> son sentencias de cualquier tipo. Una sentencia finaliza siempre con un ;.

```
if (i == 2)
        if (j == 3)
                ++j;
        else
                ++i;
// Si i==2 y j==3 entonces i==2 y j==4
// Si i==2 y j != 3 entonces i==3
```

Aunque C++ no tiene definido específicamente el tipo "boolean", cualquier expresión de tipo entero o puntero pueje jugar este papel, dado que el predicado si es cierto toma el valor 1 y si es falso toma el valor entero 0.

```
if (a < b) {int t; /*...*/}
// Si a < b el valor del predicado es 1, sino es 0
```

b) Switch: Es una sentencia de selección que ejecuta 0 o una de varias alternativas posibles de acuerdo al valor de la expresión **switch**. Su forma es:

switch (<exp-sw>) <st-case>

donde:

<exp-sw> :: Debe ser de tipo entero
<st-case> :: **case** <exp-consti>:<st-i>| **default**:<st-default>
<exp-consti> :: Expresión constante con un único valor entero (convertido al tipo de la expresión de control).
<st-i> :: Sentencia de cualquier tipo, hasta nula.

<st-default> :: Sentencia de cualquier tipo, hasta nula.

Dependiendo del valor de <exp-sw> la sentencia switch transfiere el control a una de las instrucciones etiquetadas con **case**. Si el valor coincide con alguno de los especificados por <exp-consti> tranfiere el control a <st-i>, si no coincide y existe la etiqueta **default**, transfiere a <st-default>, y si ésta no existe, no se ejecuta ninguna instrucción.

La búsqueda de equiparaciones en la ejecución de la sentencia switch se debe detener con **break** una vez que se encuentra porque sino sigue idefectiblemente hasta el **default**.

Las etiquetas **case** y **default** pueden estar en cualquier orden, pero no se puede duplicar constantes **case** en el mismo **switch**.

```
//indica si un año es bisiesto o no
int esbisiesto(int a)
{
  return (a % 4 == 0 && a & 100 != 0 || a & 400 == 0);
}
// calcula el número de días de un mes
int dias(int mes, int anno)
{
        int valor;
        switch (mes)
        {
        case 1: case 3: case 5: case 7: case 8: case 10: case 12:
                        valor = 31; break;
        case 4: case 6: case 9: case 11:
                        valor = 30; break;
        case 2:
                        valor = esbisiesto(anno) ? 29 : 28; break;
        default:
                        exit(1);// Error
        }
        return valor;
}
```

Sentencias de Iteración

Estas sentencias permiten repetir un conjunto de instrucciones en función de ciertas condiciones. Existen tres tipos de sentencias iterativas: while, do/while y for.

a) while: Su sintaxis es

while (<exp-cond>) <st>

donde:

<exp-cond> :: Debe ser cualquier escalar (tipo artimético o puntero).
<st> :: Cualquier sentencia, incluso nula.

Esta sentencia ejecuta repetidamente la instrucción <st> mientras las <exp-cond> sea diferente de 0 (o NULL en el caso de expresiones con punteros).

Como siempre, hay que tener en cuenta que, en algún momento, <exp-cond> debe tomar el valor 0 para poder salir de la iteración.

```
char x[] = "programa";
char *px = &x [0];
int con = 0;
while (* px++)
        con++;          // cuenta en con el no. de caracteres de la tira.
```

b) do/while: Su sintaxis es:

```
do <st> while (<exp-cond>);
```

donde:
<exp-cond> y <st> :: Igual que en la sentencia **while**.

Esta sentencia ejecuta repetidamente <st> hasta que <exp-cond> es falsa (0 o NULL en el caso de expresiones punteros).

La instrucción <st> se ejecuta por lo menos una vez antes de finalizar el **do/while**.

```
do
        putchar(' ');
while (col++ % col1);
// imprime blancos mientras el resto de dividir
// col y col1 sea diferente de cero
```

c) for: Su sintaxis es:

```
for ([<exp1>]); [<exp2>]; [<exp3>]) <st>
```

donde:
<exp1>, <exp2> y <exp3> :: Cualquier tipo de expresiónes.
<exp1> también puede ser una declaración, en cuyo caso el ámbito de acceso del nombre declarado es el del bloque que incluye al **for**.
<st> :: Cualquier tipo de sentencia, incluso nula.

Esta sentencia es equivalente a:

```
[<exp1>]
while ([<exp2>])
{<st>; [<exp3>;]}
```

Primero se evalúa <exp1> si existe, y sólo se evalúa una vez al comienzo del **for**. Luego se evalúa <exp2> si existe, y si es cierta (!= 0), se ejecuta la instrucción <st>, seguida por la expresión <exp3> si existe.

Finalizado el ciclo se vuelve a evaluar la <exp2> comenzando un nuevo ciclo, hasta que el valor de <exp2> es falso (cero o NULL).

```
for (;;) {/*...*/} //iteración infinita, que puede ser
                                    //interrumpida por un break o un return.

for (int con = 0, i=40; a[i] != 0 && i > 0; ++con)
                    printf ("%d %d\n",a[i--],con);
// imprime los valores del vector a hasta que se detecta
// un valor cero o se han suprimido 40 datos.
// Contabiliza en con el número de elementos impresos.
```

Sentencias de salto

Estas sentencias transfieren el control incondicionalmente. Son: **break**, **continue**, **goto** y **return**.

a) break: Origina la salida más inmediata del bloque más interno, finalizado la iteración de ese lazo o **switch.** Se puede usar solamente en **while**, **do/while**, **for** o **switch**.

b) continue: Finaliza la iteración actual de ese lazo e inicia la siguiente iteración transfiriendo el control a la <exp-cond> de la sentencia **while** y **do/while**, o a la <exp3> del **for**. Sólo puede ser usada en estas tres.

c) goto: Transfiere el control a la instrucción etiquetada con <etiqueta>. El uso más justificable del goto es la salida de una serie de lazos anidados. Pero no deberá usarse para saltar dentro de un bloque.

En C++ es ilegal saltarse una declaración inicializada explícita o implícitamente, a menos que la declaración esté dentro de un bloque interno que también se salta.

```
if (a > b)
        goto et;        // error; salta la declaración int ii=2
int ii = 2;
et:
```

d) return: Su sintaxis es:

```
return [<expresión>];
```

donde:

<expresión> :: Valor que retorna la función. Debe ser de tipo escalar o compuesto y puede ser omitido si el tipo de retorno de la función es **void**.

La sentencia return se usa para salir de una función y, a menos que el tipo de retorno de la función sea **void**, el cuerpo de la función debe contener por lo menos una sentencia **return**. El valor de <expresión> es el valor retornado por la función.

Cabe señalar que, la llamada a una función, tal como f(x,4.2), es una expresión "rvalue" y no un "lvalue".

```
x = f(arg);          // correcto
f(arg) = x;          // ilegal en C pero legal en C++
                     // si el tipo de retorno de f es una
                     // referencia.
(f(arg))++;          // ídem
```

Sentencias asm

Se usa para pasar información a través del compilador al ensamblador, y su implementación depende de cada compilador.

Sentencias declarativas

En C++ una declaración es considerada como una sentencia que introduce un nuevo identificador en un bloque.

Si un identificador es declarado en un bloque y ya fue declarado en un bloque más externo, la declaración externa queda oculta en el bloque interno.

Cualquier inicialización de variable de tipo **auto** o **register** se efectúa cada vez que se ejecuta su declaración, y cuando se finaliza la ejecución de su bloque estas variables son destruidas.

Las variables de tipo **auto** definidas en sentencias de un lazo son destruidas en cada iteración.

```
for (int j = 1; j <= 10; j++)
{
        float f = 14.3;          // f es creada, inicializada y
        //...                            // destruida en cada iteración
}
```

Es ilegal saltar una declaración con un inicializador explícito o implícito, a menos que se salte todo el bloque en que se encuentra la misma, o se sale desde un punto donde la variable ya ha sido inicializada.

```
goto a;              // error: se salta una inicialización
//...
b : int i = 14;
//...
a:
goto b;              // correcto, se destruye u y se vuelve a
                     // crear y a inicializar
```

La inicialización de un objeto local con almacenamiento **static** se realiza la primera vez que se ejecuta su declaración. Si la variable estática se inicializa con una expresión no constante, se produce su inicialización antes de que se ejecute su bloque.

```
int f(int i)
{
        static int a = f(i + 1);
        cout << a << end1;
        return i;
}
void main()
{
        cout << "valor de f:" << f(1) << end1;
}
```

El resultado es:

```
0
2
valor de f: 1
```

Gestión Dinámica de Memoria

Los objetos se ubican en la memoria de una computadora en cuatro zonas diferentes:

Segmento de datos: Para estos objetos (estáticos), la gestión de memoria se realiza en la fase de compilación (compilación - enlazado), y corresponde a objetos globales y a los declarados con **static**.

Pila (stack): Objetos locales para los que la gestión de memoria se realiza en la fase de ejecución, justamente cuando su bloque o función pasa a ser activo, y se desasignan de esa memoria cuando finaliza la ejecución de ese bloque o función. Son los objetos declarados con **auto** o **register**, implícita o explícitamente.

Registros: Idem Pila.

Heap del sistema: Son objetos dinámicos para los que la gestión de memoria se realiza en tiempo de ejecución de forma dinámica mediante el uso de las funciones **malloc**, **calloc**, **realloc** y **free** o de los operadores **new** y **delete**.

Funciones de gestión dinámica de memoria

a) malloc

Su prototipo, que está definido en el archivo de cabecera stdlib.h es:

void *malloc(size_t<longitud>);

donde:

size_t :: Tipo definido en stdlib.h, normalmente mediante un **typedef unsigned** size_t.

<longitud> :: Es el número de bytes requeridos.

La función *malloc* asigna dinámicamente memoria del heap de la memoria de la computadora en la cantidad especificada por <longitud> y devuelve un puntero **void** al comienzo de esa memoria o NULL si no hay suficiente memoria disponible para satisfacer el pedido.

```
char *pe;
int *pi;
```

```
pi = (int *) malloc (40);
// asigna 40 bytes (20 enteros)
// pi = malloc(49); está permitido en C pero no en C++
pe = (char *) malloc(100);
// asigna 100 bytes
```

b) calloc

Su sintaxis es:

```
# include <stdlib.h>
void *calloc(size_t<nelementos>,size_t<longitud>);
```

La función calloc asigna un bloque de memoria del heap del tamaño <nelementos> * <longitud> bytes. En otras palabras, asigna una bloque de memoria de un objeto de datos array[nelementos], siendo cada elemento del array de tamaño <longitud>.

Devuelve la dirección del primer elemento del array o NULL si no hay suficiente memoria disponible. El bloque de memoria queda borrado a 0.

```
char *c;
c = (char *) calloc(3,10);
// proporciona 3 objetos de 10 bytes.
```

c) realloc

Su sintaxis es:

```
# include <stdlib.h>
void *realloc(void *<mem>,size_t<longitud>);
```

Ajusta la dirección del bloque de memoria del heap cuya dirección es <mem> al tamaño <longitud>. Si ello no es posible copia el contenido apuntado por <mem> a una nueva localización del heap de tamaño <longitud>, liberando el contenido de memoria direccionado por <mem>.

Devuelve la dirección del bloque reasignado, que puede ser diferente de la dirección del bloque original <mem>. Si el bloque no puede ser ajustado o relocalizado o <longitud> == 0, realloc retorna NULL.

```
char *str = (char *) malloc(9);
strcpy (str, "FACULTAD");
str = (char *) realloc(str,20);
// str apunta al literal FACULTAD ubicado en 20 bytes.
```

d) free

Su sintaxis es:

```
# include <stdlib.h>
void free (void *<bloque>);
```

La función free libera (desasigna) un bloque de memoria del heap asignado previamente mediante malloc, calloc o realloc.

```
char *str = (char *) malloc(9);
//...
free (str);                // libera la memoria direccionada por str.
```

BIBLIOGRAFÍA

1. Ingeniería de Software – Un enfoque práctico
   Roger S. Pressman – 3o. Edición
   Mc Graw Hill – 1993
2. Ingeniería de Software – Un enfoque práctico
   Roger S. Pressman – 5º. Edición
   Mc Graw Hill – 2000
3. Ingeniería de Software
   Ian Sommerville – 2º. Edición
   Addison-Wesley Iberoamericana – 1988
4. Ingeniería de Software
   Ian Sommerville – 6º. Edición
   Addison-Wesley Iberoamericana – 2002
5. Análisis Estructurado Moderno
   Edward Yourdon
   Prentice Hall – 1989
6. Análisis y Diseño de Sistemas
   Kendall y Kendall
   Prentice Hall – 1991
7. Análisis y Diseño de sistemas
   Kendall y Kendall – 3º. Edición
   Prentice Hall – 1997
8. Entornos y Metodologías de programación
   F. Alonso Amo y F.J. Segovia Pérez
   Paraninfo – 1993
9. Análisis estructurado de sistemas
   Chris Gane y Trish Sarson
   El Ateneo – 1987
10. Ingeniería de Software
    Shari Lawrence Pgleeger
    Prentice Hall – 2002
11. Object Oriented Analysis
    Peter Coad/ Edward Yourdon – 2º. Edición
    Yourdon Press Computing Series – 1991
12. Object Oriented Design
    Peter Coad/ Edward Yourdon – 2º. Edición
    Yourdon Press Computing Series – 1991
13. Análisis y Diseño Orientado a Objetos
    James Martin y James Odell
    Prentice Hall – 1992
14. Programación Orientada a objetos – Un enfoque evolutivo
    Cox y Novobilski – 2º. Edición
    Addison Wesley – 1993