Seminario de C++98-C++0x

1 Genericidad en C++ basados en templates.Tipos de argumentos

La genericidad es una propiedad que permite definir una clase o una función sin tener que especificar el tipo de todos o alguno de sus miembros.
La utilidad principal de este tipo de clases o funciones (genéricas) es la de agrupar variables cuyo tipo no esté predeterminado. De esta forma el funcionamiento de una pila, una cola, una lista, un conjunto, un diccionario o un array es el mismo independientemente del tipo de datos que almacene (int, long, double, char, u object de una clase definida por el usuario). Esta propiedad no es imprescindible de un lenguaje de programación orientado a objetos. En C++ la genericidad se alcanza con los templates.
Un template implementa el concepto de tipo parametrizado permitiendo el concepto de genericidad en el lenguaje C++. Le dice al compilador que la definición de clase o de función que le sigue manipulará uno o más tipos de datos no especificados. En el momento en que el código de la clase o de la función actual es generado, los tipos deben ser especificados para que el compilador pueda sustituirlos.

1.1 Tipos de argumentos

Las plantillas de clase y función pueden tener argumentos predeterminados. Cuando una plantilla tiene un argumento predeterminado, se puede dejar sin especificar al usarlo. Por ejemplo, la plantilla std::vector tiene un argumento predeterminado para el asignador:

```
template <class T, class Allocator = allocator<T>> class vector;
```

En la mayoría de los casos, la clase std::allocator predeterminada es aceptable, por lo que se usa un vector similar al siguiente:

```
vector<int> myInts;
```

Pero si es necesario, se puede especificar un asignador personalizado de la siguiente manera:

```
vector<int, MyAllocator> ints;
```

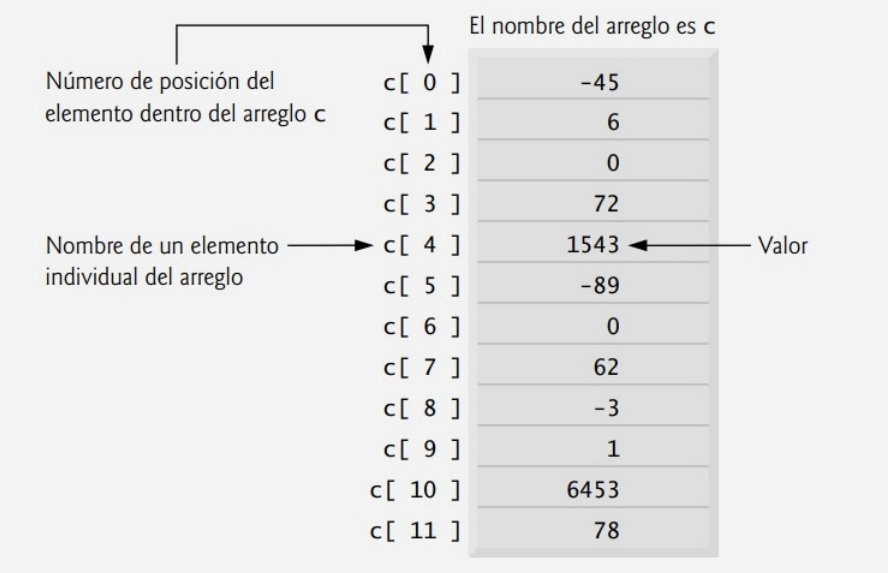
Para varios argumentos de plantilla, todos los argumentos después del primer argumento predeterminado deben tener argumentos predeterminados. Cuando se usa una plantilla cuyos parámetros están todos predeterminados, se usan corchetes angulares vacíos:

```
template<typename A = int, typename B = double>
class Bar
{
    //...
};
...
int main()
{
    Bar<> bar; // use all default type arguments
}
```

2 Definición de arrays en C++

2.1 Definición

Un arreglo es un grupo de ubicaciones de memoria consecutivas, todas ellas del mismo tipo. Para hacer referencia a una ubicación o elemento específico en el arreglo, especificamos su nombre y el número de posición del elemento específico en el arreglo.



En la imagen se muestra un arreglo de enteros llamado c.Este arreglo contiene 12 elementos. Para hacer referencia a cualquiera de estos elementos en un programa, se proporciona el nombre del arreglo seguido del número

de posición del elemento específico entre corchetes ([]). Los nombres de los arreglos siguen las mismas convenciones que los demás nombres de variables; es decir, deben ser identificadores. Un subíndice debe ser un entero o una expresión entera (usando cualquier tipo integral). Si un programa utiliza una expresión como un subíndice, entonces el programa evalúa la expresión para determinar el subíndice.
Ejemplo

```
c[ a + b ] += 2;
```

2.2 Declaración y creación de arreglos

Los objetos arreglo ocupan espacio en memoria. Para especificar el tipo de los elementos y el número de elementos requerido por un arreglo, use una declaracón de la forma:
tipo nombreArreglo[tamañoArreglo];

El compilador reserva la cantidad apropiada de memoria. (una declaración que reserva memoria se conoce en forma más apropiada como definición.) El tamañoArreglo debe ser una constante entera mayor que cero. Por ejemplo, para indicar al compilador que debe reservar 12 elementos para el arreglo c de enteros, use la siguiente declaración:

```
int c[ 12 ]; // c es un arreglo de 12 enteros
```

Se puede reservar memoria para varios arreglos con una sola declaración. La siguiente declaración reserva 100 elementos para el arreglo b de enteros y 27 elementos para el arreglo x de enteros.

```
int b[ 100 ]; // b es un arreglo de 100 enteros\\
x[ 27 ]; // x es un arreglo de 27 enteros\\
```

2.3 Inicialización

Los elementos de un arreglo también se pueden inicializar en la declaración del arreglo, para lo cual colocamos después del nombre del arreglo un signo igual y una lista entre llaves, separada por comas, de inicializadores. Si hay menos inicializadores que elementos en el arreglo, el resto de los elementos del arreglo se inicializan con cero.
Por ejemplo:

```
int n[ 10 ] = {}; // inicializa los elementos del arreglo n con 0
```

Si el tamaño del arreglo se omite en una declaración con una lista inicializadora, el compilador determina el número de elementos en el arreglo

mediante un conteo del número de elementos en la lista inicializadora. Por ejemplo

```
int n[] = { 1, 2, 3, 4, 5 };\\
```

crea un arreglo de cinco elementos.
Si se especifican el tamaño del arreglo y una lista inicializadora en la declaración de un arreglo, el número de inicializadores debe ser menor o igual que el tamaño del arreglo. La declaración del arreglo

```
int n[ 5 ] = { 32, 27, 64, 18, 95, 14 };\\
```

produce un error de compilación, ya que hay 6 inicializadores y sólo 5 elementos en el arreglo.

3 Distintos tipos de constructores en C++(defecto y copia)

Un constructor es una función que es automáticamente llamada cuando se necesita inicializar variables o asignar memoria dinámica durante el proceso de creación de un objeto y así evitar que se obtengan valores inesperados.Los constructores no tiene asignado ningún tipo de retorno, ni siquiera void y deben tener el mismo nombre de la clase.Los constructores se pueden sobrecargar, o sea definir varios constructores (con el mismo nombre) pero deben diferir al menos una vez en el tipo o la cantidad de parámetros.

3.1 ¿Qué hace cada uno de ellos?

- CONSTRUCTOR POR DEFECTO Y CONSTRUCTOR CON PARÁMETROS CON VALOR POR DEFECTO
  Se llama constructor por defecto a un constructor que no necesita que se le pasen parámetros o argumentos para inicializar las variables miembro de la clase. Un constructor por defecto es pues un constructor que no tiene argumentos o que, si los tiene, todos sus argumentos tienen asignados un valor por defecto en la declaración del constructor. En cualquier caso, puede ser llamado sin tenerle que pasar ningún argumento.

- CONSTRUCTOR DE OFICIO

  - Es un constructor generado automáticamente por C++ que no recibe parámetros.

  - No se crea en el caso que se haya definido otro constructor.
  - Inicia todas las variables miembro a 0. Aunque esto no es razonable en todos los contextos.

- CONSTRUCTOR DE COPIA
  En C++ se obliga a inicializar las variables miembro de una clase llamando a un constructor, cada vez que se crea un objeto de dicha clase. Se ha comentado también que el constructor puede recibir como parámetros los valores que tiene que asignar a las variables miembro, o puede asignar valores por defecto.Por definición, el constructor de copia tiene un único argumento que es una referencia constante a un objeto de la clase. Su declaración sería:

```
C_Cuenta(const C_Cuenta&);
```

  El constructor de copia de oficio se limita a realizar una copia bit a bit de las variables miembro del objeto original al objeto copia. Hay casos en lo que la copia bit a bit no da los resultados esperados. En este caso el programador debe preparar su propio constructor de copia e incluirlo en la clase como un constructor sobrecargado más.Este surge si no se crea un constructor de copia.

- CONSTRUCTOR CON ARGUMENTOS

  - Representa un constructor que recibe uno o varios argumentos como parámetros.
  - Debe llamarse pasándole al menos 1 argumento. En este caso todos los demas serán argumentos que tengan algun valor por defecto.

3.2 ¿Cuándo se llaman?

Los constructores se llaman para crear una instancia del objeto en cuestión. Se pueden llamar utilizando el operador new que reserva memoria dinámicamente y retorna un puntero al inicio del bloque de la misma.

- CONSTRUCTOR POR DEFECTO Y CONSTRUCTOR CON PARÁMETROS CON VALOR POR DEFECTO El constructor por defecto es necesario si se quiere hacer una declaración en la forma:

```
C_Cuenta c1;
```

  y también cuando se quiere crear un vector de objetos, por ejemplo en la forma:

```
C_Cuenta cuentas[100];
```

ya que en este caso se crean e inicializan múltiples objetos sin poderles pasar argumentos personalizados o propios para cada uno de ellos.

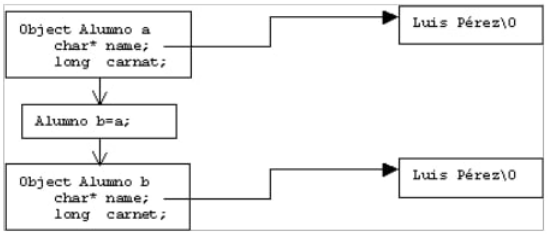
El constructor puede tener definidos unos valores por defecto para los parámetros, que se asignen a las variables miembro de la clase. Esto es especialmente útil en el caso de que una variable miembro repita su valor para todos o casi todos los objetos de esa clase que se creen. Considérese el ejemplo siguiente:

```
class C_Cuenta {
// Variables miembro
private:
double Saldo;            // Saldo Actual de la cuenta
double Interes;          // Interés aplicado

public:
C_Cuenta(double unSaldo=0.0, double unInteres=0.0)  //
    Constructor
{
      SetSaldo(unSaldo);
      SetInteres(unInteres);
}
};
```

- CONSTRUCTOR DE COPIA El constructor de copia se llama cuando se necesita crear un objeto nuevo a partir de otro objeto.
  Surge la necesidad de escribir un constructor de copia distinto del que proporciona el compilador en casos en los que la copia bit a bit no resulta.

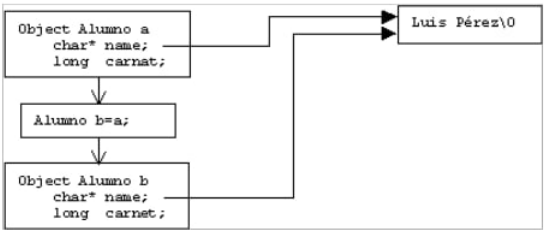
  Fig:



La situación mostrada en la figura 2 puede tener consecuencias no deseadas. Por ejemplo, si se quiere cambiar el nombre del Alumno a, lo primero que se hace es liberar la memoria a la que apunta a.nombre, reservar memoria para el nuevo nombre haciendo que a.nombre apunte al comienzo de dicha memoria, y almacenar allí el nuevo nombre de a.

Como el objeto b no se ha tocado, su variable miembro b.nombre se ha quedado apuntado a una posición de memoria que ha sido liberada en el proceso de cambio de nombre de a.La consecuencia es que b ha perdido información y los más probable es que el programa falle.
Se llega a una situación parecida cuando se destruye uno de los dos objetos a o b. Al destruir uno de los objetos se libera la memoria que comparten, con el consiguiente perjuicio para el objeto que queda, puesto que su puntero contiene la dirección de una zona de memoria liberada, disponible para almacenar otra información.

Fig:



La figura muestra la situación a la que se llega con un constructor de copia correctamente programado por el usuario. En este caso, el constructor no copia bit a bit la dirección contenida en a.nombre, sino que reserva memoria, copia a esa memoria el contenido apuntado por a.nombre, y guarda en b.nombre la dirección de esa nueva memoria reservada. Ninguno de los problemas anteriores sucede ahora.

3.3 Inicialización de campos

Como sabemos C++ no permite crear objetos sin dar un valor inicial apropiado a todas sus variables miembro. Esto se hace por medio de unas funciones llamadas constructores, que se llaman automáticamente siempre que se crea un objeto de una clase.
La idea del constructor es inicializar variables, y una sentencia de asignación no es la única ni la mejor forma de inicializar una variable. C++ permite inicializar variables miembro fuera del cuerpo del constructor, de la siguiente forma:

```
C_Cuenta::C_Cuenta(double unSaldo, double unInteres) :
                    Saldo(unSaldo), Interes(unInteres)    //
                        inicializadores
{// En este caso el cuerpo del constructor está vacío }
```

donde se ve que los inicializadores se introducen, tras el carácter dos puntos (:), separados por comas, justo antes de abrir las llaves del cuerpo del con-

structor. Constan del nombre de la variable miembro seguido, entre paréntesis, del argumento que le da valor. Los inicializadores son más eficientes que las sentencias de asignación, y además permiten definir variables miembro const, que pueden ser inicializadas pero no asignadas.

3.4 ¿Cómo funciona el paso de parámetros cuando se llama a una función?

Los parámetros se comportan como cualquier otra variable dentro de la función. El paso de parámetros en C++ se puede hacer de dos formas:

- Paso de parámetros por valor.
- Paso de parámetros por referencia.

3.5 ¿Cuándo se deben utilizar parámetros por valor, por puntero o por referencia?

- Paso de parámetros por valor.
  Pasar parámetros por valor significa que a la función se le pasa una copia del valor que contiene el parámetro actual. Los valores de los parámetros de la llamada se copian en los parámetros de la cabecera de la función. La función trabaja con una copia de los valores por lo que cualquier modificación en estos valores no afecta al valor de las variables utilizadas en la llamada. Aunque los parámetros actuales (los que aparecen en la llamada a la función) y los parámetros formales (los que aparecen en la cabecera de la función) tengan el mismo nombre son variables distintas que ocupan posiciones distintas de memoria. Por defecto, todos los argumentos salvo los arrays se pasan por valor.

- Paso de parámetros por referencia.
  Hay dos formas de pasar los parámetros por referencia:
  - Paso de parámetros por referencia basado en punteros
  - Paso de parámetros por referencia usando referencias al estilo C++.

- Paso de parámetros por referencia basado en punteros
  Cuando se pasan parámetros por referencia, se le envía a la función la dirección de memoria del parámetro actual y no su valor. La función realmente está trabajando con el dato original y cualquier modificación del valor que se realice dentro de la función se estará realizando con el parámetro actual. Para recibir la dirección del parámetro actual, el parámetro formal debe ser un puntero.
  Ejemplo 1:

Programa c++ que lee dos números por teclado y los envía a una función para que intercambie sus valores.

```
#include <iostream>
using namespace std;
void intercambio(int *, int *);
int main( )
{
    int a, b;
    cout << "Introduce primer numero: ";
    cin >> a;
    cout << "Introduce segundo numero: ";
    cin >> b;
    cout << endl;
    cout << "valor de a: " << a << " valor de b: " << b <<
        endl;
    intercambio(&a, &b);
    cout << endl << "Despues del intercambio: " << endl <<
        endl;
    cout << "valor de a: " << a << " valor de b: " << b <<
        endl;
    system("pause");
}
void intercambio(int *x, int *y)
{
    int z;
    z = *x;
    *x = *y;
    *y = z;
}
```

En la llamada, a la función se le envía la dirección de los parámetros. El operador que obtiene la dirección de una variable es &.

```
intercambio(&a, &b);
```

En la cabecera de la función, los parámetros formales que reciben las direcciones deben ser punteros. Esto se indica mediante el operador *

```
void intercambio(int *x, int *y)
```

Los punteros x e y reciben las direcciones de memoria de las variables a y b. Al modificar el contenido de las direcciones x e y, indirectamente estamos modificando los valores a y b. Por tanto, pasar parámetros por referencia a una función es hacer que la función acceda indirectamente a las variables pasadas.

- Paso de parámetros por referencia usando referencias al estilo C++ Una referencia es un nombre alternativo (un alias, un sinónimo) para un objeto. Una referencia no es una copia de la variable referenciada,

sino que es la misma variable con un nombre diferente. Utilizando referencias, las funciones trabajan con la misma variable utilizada en la llamada. Si se modifican los valores en la función, realmente se están modificando los valores de la variable original.
Ejemplo:
El primer ejemplo del punto anterior utilizando referencias lo podemos escribir así:

```
#include <iostream>
using namespace std;
void intercambio(int &, int &);
int main( )
{    int a, b;
     cout << "Introduce primer numero: ";
     cin >> a;
     cout << "Introduce segundo numero: ";
     cin >> b;
     cout << endl;
     cout << "valor de a: " << a << " valor de b: " << b <<
         endl;
     intercambio(a, b);
     cout << endl << "Despues del intercambio: " << endl <<
         endl;
     cout << "valor de a: " << a << " valor de b: " << b <<
         endl;
     system("pause");
}
void intercambio(int &x, int &y)
{
     int z;
     z = x;
     x = y;
     y = z;
}
```

En la declaración de la función y en la definición se coloca el operador referencia a & en aquellos parámetros formales que son referencias de los parámetros actuales:

```
void intercambio(int &, int &);    //declaración de la
    función
void intercambio(int &x, int &y)  //definición de la función
```

Cuando se llama a la función:

```
intercambio(a, b);
```

se crean dos referencias (x e y) a las variables a y b de la llamada. Lo que se haga dentro de la función con x e y se está haciendo realmente

con a y b.

3.6 Constructores con un solo argumento

Converting constructor o Constructores con un solo argumento:
Representa un constructor que recibe un solo argumento como parámetro Un constructor declarado sin el especificador de función explicit que se puede llamar con un solo parámetro especifica una conversión del tipo de su primer parámetro al tipo de su clase. Tal constructor se llama constructor de conversión.
Ejemplo:

```
class MyClass
{
  public:
     int a, b;
     MyClass( int i ) {}
     MyClass( const char* n, int k = 0 ) {}
     MyClass( MyClass& obj ) {}
}
```

Para ser un constructor de conversión, el constructor debe tener un solo argumento (en el segundo, el segundo argumento tiene un valor predeterminado) y ser declarado sin la palabra clave explicit.

3.7 Constructores explicit

Fuerza al programador a que se haga un cast explicito sobre el objeto que recibe el constructor de un solo argumento de la clase al tipo del objeto que se esta creando, evitando así que se efectúen operaciones de cast implícitas.
Ejemplo:

```
class MyClass
{
  public:
     int a, b;
     explicit MyClass( int i ) {}
}
```

Entonces, el compilador no aceptaría

```
     int main()
  {
     MyClass M = 1 ;
  }
```

Ya que esto es conversión implícita. En su lugar, tienen que escribir

```
    int main()
{
    MyClass M(1) ;
    MyClass M = MyClass(1) ;
    MyClass* M = new MyClass(1) ;
    MyClass M = (MyClass)1;
    MyClass M = static_cast<MyClass>(1);
}
```

explicit la palabra clave siempre se usa para evitar la conversión implícita de un constructor y se aplica al constructor en una declaración de clase.

4 Funciones inline y const

Funcion inline:
En la mayoría de los casos la programación orientada a objetos obliga a utilizar funciones para poder acceder a las variables miembro de una clase. Por esta razón un programa orientado a objetos contiene muchas llamadas a funciones. Por otra parte, muchas de las funciones que se utilizan contienen sólo unas pocas sentencias o incluso una sola, por ejemplo las funciones de lectura y asignación de valores de una variable.
Cada llamada y retorno de una función tiene un cierto costo computacional, porque es necesario reservar una zona de memoria para los argumentos de las funciones llamadas, que a veces, además tienen que ser copiados. En la mayoría de los casos el tiempo empleado en la transmisión de datos es despreciable frente al empleado en los cálculos. En el caso de que la función sea muy sencilla, sin embargo, no se puede despreciar ese tiempo y el uso frecuente de funciones muy sencillas y breves se revela muy poco eficiente.
La expansión inline ofrece la solución a este problema sustituyendo en el programa la llamada a la función por el código de la misma, modificado para simular el paso de argumentos y valor de retorno. Las funciones inline eliminan la necesidad de utilizar las macros de C.
La ventajas de las funciones inline vienen dadas porque su utilización puede suponer una reducción del tiempo de ejecución de un programa. El inconveniente de usar funciones inline es que la introducción de una copia del código en cada llamada a una función puede hacer que el tamaño del programa aumente considerablemente. En definitiva, el uso de este tipo de funciones está recomendado en el caso de que sean funciones muy sencillas cuyo tiempo de llamada es comparable al tiempo de cálculo. Por el contrario, en el caso de funciones más grandes, la mejora en el tiempo de ejecución es despreciable y el tamaño del programa puede crecer innecesariamente.

Funcion const:

En C++ el especificador const se puede utilizar con variables y con punteros. Las variables definidas como const no son lo mismo que las constantes simbólicas, aunque evidentemente hay una cierta similitud en las áreas de aplicación. Si una variable se define como const se tiene la garantía de que su valor no va a cambiar durante toda la ejecución del programa. Si en alguna sentencia del programa se intenta variar el valor de una variable definida como const, el compilador produce un mensaje de error. Esta precaución permite detectar errores durante la compilación del programa, lo cual siempre es más sencillo que detectarlos en tiempo de ejecución.
Las variables de este tipo pueden ser inicializadas pero no pueden estar a la izquierda de una sentencia de asignación.
Las variables declaradas como const tienen importantes diferencias con las constantes simbólicas definidas con la directiva define del preprocesador. Aunque ambas representan valores que no se puede modificar, las variables const están sometidas a las mismas reglas de visibilidad y duración que las demás variables del lenguaje.
Las variables const de C++ pueden ser utilizadas para definir el tamaño de un vector en la declaración de éste, cosa que no está permitida en C.
Es muy frecuente que las funciones a las que por motivos de eficiencia (para no tener que sacar copias de los mismos) se les pasan los argumentos por referencia, éstos serán declarados como const en la definición y en el prototipo de la función, con objeto de hacer imposible una modificación accidental de dichos datos. Esto sucede por ejemplo con las funciones de manejo de cadenas de caracteres. El prototipo de la función strcpy() puede ser como sigue:

```
char *strcpy(char *s1, const char *s2);
```

4.1 Implementación de las funciones inline

C++ permite 2 maneras distintas de implementar funciones inline
En el caso de los punteros hay que distinguir entre dos formas de aplicar el cualificador const:

1. La primera de ellas consiste en colocar la palabra inline precediendo a la definición de la función:

   ```
   inline double funcion (double uno)
   {return uno;}
   ```

   Es importante saber que esta indicación puede ser ignorada por el compilador en el caso de que la función en cuestión sea tan larga o tan complicada que su expansión inline resulte desaconsejable. De todos modos, el que se produzca o no la expansión de la función no hace variar nada la definición de la misma.

La definición de las funciones inline debe hacerse en los ficheros de encabezamiento (extensión *.h), y no en los ficheros fuente (extensión *.cpp). Cada vez que se utiliza una función inline su llamada es sustituida por el código de la misma, por lo que oíste debe ser accesible para cualquier fichero fuente, y eso se consigue incluyendola en el fichero header.

2. El segundo método de declaración de funciones inline consiste en colocar la definición completa de la misma en la declaración de la función:

```
class una {
 double tal;
 public:
 double Expandida( )
 { return tal;}
};
```

En el caso de que se incluya la definición de una función inline en una clase, esta misma definición sirve también como declaración. Estas definiciones pueden incluir argumentos por defecto. Tal como se han definido las funciones, incluyendo su definición en la declaración, estas funciones ya eran inline. En el siguiente ejemplo se añade la palabra inline, aunque no es necesaria, para subrayar esta característica de las funciones:

```
class C_Cuenta {
 // Variables miembro
private: // La palabra private no es necesaria
char *Nombre; // Nombre de la persona
double Saldo;
double Interes; // Interés aplicado
public:
// Métodos
inline char *GetNombre() // La palabra inline no es
    necesaria
{ return Nombre; }
inline double GetSaldo()
 { return Saldo; }
 inline double GetInteres()
 { return Interes; }
 inline void SetSaldo(double unSaldo)
 { Saldo = unSaldo; }
 inline void SetInteres(double unInteres)
 { Interes = unInteres; }
 void Ingreso(double unaCantidad)
 { SetSaldo( GetSaldo() + unaCantidad ); }
```

4.2 Diferencia entre const T x; T const x; y const T const x;

No existe diferencia entre const T x y T const x, es decir el calificador const se aplica a lo que está inmediatamente a su izquierda, si no hay nada a su izquierda, se aplica a lo que esta inmediatamente a su derecha. Por tanto para estar en armonía con la regla 'Derecha Izquierda' se recomienda que es mejor escribir T const x.
Sin embargo cambia cuando son punteros es decir const T *x y T* const x, son diferentes, al igual que const T *const x.

4.3 Función const para punteros

Existen dos formas de aplicar el cualificador const:

1. un puntero variable apuntando a una variable constante
2. un puntero constante apuntando a una variable cualquiera

Un puntero a una variable const no puede modificar el valor de esa variable (si se intentase el compilador lo detectaría e imprimiría un mensaje de error), pero ese puntero no tiene por qué apuntar siempre a la misma variable. En el caso de un puntero const, éste apunta siempre a la misma dirección de memoria pero el valor de la variable almacenada en esa dirección puede cambiar sin ninguna dificultad. Un puntero a variable const se declara anteponiendo la palabra const:

```
const char *nombre1 "Ramón" // no se puede modificar el valor de
    la variable
```

Por otra parte, un puntero const a variable cualquiera se declara interponiendo la palabra const entre el tipo y el nombre de la variable:

```
char* const nombre2 "Ramón" // no se puede modificar la dirección
     a la que apunta el puntero, pero sí el valor.
```

En ANSI C una variable declarada como const puede ser modificada a través de un puntero a dicha variable. Por ejemplo, el siguiente programa compila y produce una salida i=3 con el compilador de C, pero da un mensaje de error con el compilador de C++:

```
#include <stdio.h>
void main(void)
{
 const int i = 2;
 int *p;
 p = &i;
 *p = 3;
```

```
printf("i=%d", i);
}
```

5 Redefinición de operadores [] y +.Tipos de traspasos en C++

Los operadores de C++, al igual que las funciones, pueden ser sobrecargados (overloaded). Este es uno de los aspectos más característicos de este lenguaje. La sobrecarga de operadores quiere decir que se pueden redefinir algunos de los operadores existentes en C++ para que actúen de una determinada manera, definida por el programador, con los objetos de una clase determinada. Los programadores pueden usar operadores con tipos definidos por el usuario también. Aunque C++ no permite crear nuevos operadores, sí permite sobrecargar la mayoría de los operadores existentes para que, cuando éstos se utilicen con objetos, tengan un significado apropiado. Es una poderosa herramienta.

Para sobrecargar un operador, se escribe la definición de una función miembro no static o la definición de una función global como se hace normalmente, excepto que el nombre de la función se convierte ahora en la palabra clave operator, seguida del símbolo del operador que se va a sobrecargar.

En el caso de la funcion operador + se utilizaría para sobrecargar el operador de suma. Cuando los operadores se sobrecargan como funciones miembro, deben ser no static, debido a que se deben llamar en un objeto de la clase y deben operar en ese objeto.

El significado de la forma en que trabaja un operador con objetos de tipos fundamentales no se puede modificar mediante la sobrecarga de operadores. Por ejemplo, no podemos modificar el significado de la forma en como + suma dos enteros. La sobrecarga de operadores funciona sólo con objetos de los tipos definidos por el usuario, o con una mezcla de un objeto de un tipo definido por el usuario y un objeto de un tipo fundamental.

Al sobrecargar [] operador para indexar o cualquiera de los operadores de asignación,la función de sobrecarga de operadores debe declararse como una clase miembro. Para los otros operadores, las funciones de sobrecarga de operadores pueden ser clases miembro o funciones globales.

Cuando una función de operador se implementa como función miembro, el operando de más a la izquierda (o el único) debe ser un objeto (o una referencia a un objeto) de la clase del operador. Si el operando izquierdo debe ser un objeto de una clase distinta o de un tipo fundamental, esta función de operador se debe implementar como una función global.

Ejemplo

```
// operador de sub\'indice sobrecargado para objetos Array no
   const;
// la devoluci\'on de una referencia crea un lvalue modificable
int &Array::operator[ ]( int subindice )
{
// comprueba error de subíndice fuera de rango
        if ( subindice < 0 || subindice >= tamano )
         {
          cerr << "\nError: subindice " << subindice
          << "fuera_de_rango" << endl;
          exit( 1 ); // termina el programa; subindice fuera de
              rango
          } // fin de if

          return ptr[ subindice ]; // devuelve una referencia
  } // fin de la función operator[]


  int main()
  {
        Array enteros1[7]={1,2,3,4,5,6,7}; // objeto Array de 7
            elementos
    Array enteros2; // objeto Array de 10 elementos de manera
        predeterminada

    // usa el operador de subíndice sobrecargado para crear
        rvalue
    cout << "\nenteros1[5] es " << enteros1[ 5 ];

    // usa el operador de subíndice sobrecargado para crear
        lvalue
    cout << "\n\nAsignando 1000 a enteros1[5]" << endl;
    enteros1[5]=1000;

    // trata de usar un subíndice fuera de rango
    cout << "\nTrata de asignar 1000 a enteros1[15]" << endl;
    enteros1[ 15 ] = 1000; // ERROR: fuera de rango
    return 0;
  }
```

Se utiliza el operador de subíndice sobrecargado para hacer referencia a enteros1[5]: un elemento de enteros1 dentro del rango. Este nombre con subíndice se utiliza como rvalue para imprimir el valor asignado a enteros1[5]. Mas abajo se utiliza enteros1[5] como un lvalue modificable del lado izquierdo de una instrucción de asignación para asignar un nuevo valor, 1000, al elemento 5 de enteros1.
En la línea se trata de asignar el valor 1000 a enteros1[15]; un elemento fuera de rango. En este ejemplo, operator[ ] determina que el subíndice está fuera de rango, imprime un mensaje y termina el programa. Observe que resaltamos la ls últimas lineas del programa para enfatizar que es un error acceder a un elemento que está fuera de rango. Éste es un error lógico en

tiempo de ejecución.
Se puede observar que el operador de subíndice de arreglo [] no está restringido para usarlo sólo con arreglos; por ejemplo, también se puede utilizar para seleccionar elementos de otros tipos de clases contenedoras, como listas enlazadas, cadenas y diccionarios. Además, cuando se definen las funciones operator[ ], los subíndices ya no tienen que ser enteros; también se pueden usar caracteres, cadenas, números de punto flotante o incluso objetos de clases definidas por el usuario.

6 Especialización de los templates

En algunos casos, no es posible o deseable que una plantilla defina exactamente el mismo código para cualquier tipo. Por ejemplo, es posible que se desee definir una ruta de acceso de código que se va a ejecutar solo si el argumento de tipo es un puntero, o un tipo derivado de una clase base determinada. En tales casos, se puede definir una especialización de la plantilla para ese tipo determinado. Cuando un usuario crea una instancia de la plantilla con ese tipo, el compilador usa la especialización para generar la clase y para todos los demás tipos, el compilador elige la plantilla más general. Las especializaciones en las que todos los parámetros están especializados son especializaciones completas. Si solo algunos de los parámetros están especializados, se denomina especialización parcial.

```
template <typename K, typename V>
class MyMap{/*...*/};

// partial specialization for string keys
template<typename V>
class MyMap<string, V> {/*...*/};
...
MyMap<int, MyClass> classes; // uses original template
MyMap<string, MyClass> classes2; // uses the partial
    specialization
```

Una plantilla puede tener cualquier número de especializaciones siempre que cada parámetro de tipo especializado sea único. Solo las plantillas de clase pueden estar parcialmente especializadas. Todas las especializaciones completas y parciales de una plantilla deben declararse en el mismo espacio de nombres que la plantilla original.

La generalidad es una propiedad que permite definir una clase o una función sin tener que especificar el tipo de todos o alguno de sus miembros.La utilidad principal de este tipo de clases o funciones es la de agrupar variables cuyo tipo no esté predeterminado. Así el funcionamiento de una pila, una cola, una lista, un conjunto, un diccionario o un array es el mismo independi-

entemente del tipo de datos que almacene (int, long, double, char, u objetos de una clase definida por el usuario). En definitiva estas clases se definen independientemente del tipo de variables que vayan a contener y es el usuario de la clase el que debe indicar ese tipo en el momento de crear un objeto de esa clase.

6.1 Templates de funciones

Si se quiere crear una función que devolviese el mínimo entre dos valores independientemente de su tipo (se supone que ambos tienen el mismo tipo). Se podría pensar en definir la función tantas veces como tipos de datos se puedan presentar (int, long, float, double, etc.). Aunque esto es posible, éste es un caso ideal para aplicar plantillas de funciones. Esto se puede hacer de la siguiente manera:

```
// Declaración de la plantilla de función
template <class T> T minimo( T a, T b);
```

En ese caso con <classT> se está indicando que se trata de una plantilla cuyo parámetro va a ser el tipo T y que tanto el valor de retorno como cada uno de los dos argumentos va a ser de este tipo de dato T. En la definición y declaración de la plantilla puede ser que se necesite utilizar mas de un tipo de dato e incluido algún otro parámetro constante que pueda ser utilizado en las declaraciones. Por ejemplo, si hubiera que pasar dos tipos a la plantilla, se podría escribir:

```
// Declaración de la plantilla de función con dos tipos de datos
template <class T1, class T2> void combinar(T1 a, T2 b);
```

Podría darse el caso también de que alguno de los argumentos o el valor de retorno fuese de un tipo de dato constante y conocido. En ese caso se indicaría explícitamente como en una función convencional.

```
// Definición de la plantilla de función
template <class T> T minimo(T a, T b)
{
 if(a <= b)
 return a;
 else
 return b;
}
```

Ejemplo donde se utiliza la plantilla de función recién definida

```
#include <iostream.h>
template <class T> T minimo(T a, T b);
void main(void)
{
 int euno=1;
```

```
 int edos=5;
 cout << minimo(euno, edos) << endl;
 long luno=1;
 long ldos=5;
 cout << minimo(luno, ldos) << endl;
 char cuno='a';
 char cdos='d';
 cout << minimo(cuno, cdos) << endl;
 double duno=1.8;
 double ddos=1.9;
 cout << minimo(duno, ddos) << endl;
}
```

La ejecución del programa anterior demuestra que el tipo de los argumentos y el valor de retorno de la función minimo() se particularizan en cada caso a los de la llamada.
De forma general como se puede apreciar un template de funcion sería:

```
template <class identifier> function_declaration;
```

6.2 Templates de clases

De una manera semejante a como se hace para las funciones se puede generalizar para el caso de las clases por medio de plantillas de clases.

```
template <class identifier> class_declaration;
```

Ejemplo:

```
// fichero Pila.h
template <class T>
// declaración de la clase
class Pila
{
public:
 Pila(int nelem=10); // constructor
 void Poner(T);
 void Imprimir();
private:
 int nelementos;
 T* cadena;
 int limite;
};
// definición del constructor
template <class T> Pila<T>::Pila(int nelem)
{
 nelementos = nelem;
 cadena = new T(nelementos);
 limite = 0;
};
// definición de las funciones miembro
```

```
template <class T> void Pila<T>::Poner(T elem)
{
 if (limite < nelementos)
 cadena[limite++] = elem;
};
template <class T> void Pila<T>::Imprimir()
{
 int i;
 for (i=0; i<limite; i++)
 cout << cadena[i] << endl;
};
```

El programa principal sería:

```
#include <iostream.h>
#include "Pila.h"
void main()
{
 Pila <int> p1(6);
 p1.Poner(2);
 p1.Poner(4);
 p1.Imprimir();
 Pila <char> p2(6);
 p2.Poner('a');
 p2.Poner('b');
 p2.Imprimir();
}
```

En este programa principal se definen dos objetos p1 y p2 de la clase Pila. En p1 el parámetro T vale int y en p2 ese parámetro vale char. El funcionamiento de todas las variables y funciones miembro se particulariza en cada caso para esos tipos de variable.

7 Tipos union

Las uniones C++ son un tipo especial de clase; un tipo de variable con cierta similitud con las estructuras. Pueden albergar diferentes tipos de datos, pero solo uno, de entre todos los posibles, al mismo tiempo. Se dice que solo uno puede estar "activo" en cada momento, y se corresponden con los registros de tipo variable de Pascal y Modula-2. El tamaño de una unión es el del mayor elemento que puede albergar. El valor exacto depende de la implementación y de las alineaciones internas.
Desde el punto de vista de su organización interna son como estructuras en las que todos sus miembros tuviesen el mismo desplazamiento respecto al origen. Desde un punto de vista funcional pueden ser considerados almacenamientos u objetos multi-uso.

7.1 Declaración

La sintaxis de declaración es similar a las estructuras con las siguientes diferencias:

1. Las uniones pueden contener campos de bits, pero solo uno puede estar activo. Todos comienzan en el principio de la unión y como consecuencia, dado que los campos de bits son dependientes de la implementación, pueden presentar problemas para escribir código portable.

2. Un objeto que tenga constructor o destructor no puede ser utilizado como miembro de una unión.

3. A diferencia de las estructuras, en las uniones C++ no se permiten los especificadores de acceso public, private y protected de las clases. Todos sus campos son públicos.

4. Las uniones solo pueden ser inicializadas en su declaración mediante su primer miembro. Ejemplo:

```
union local87 {
  int i;
  double d;
} a = { 20 };
```

```
union miUnion {        // definición de unión de nombre miUnion
   int i;
   double d;
   char ch;
   char *ptr;
} mu, *muptr=&mu;      // declara un objeto y un puntero al objeto
```

Aquí se ha definido un tipo nuevo de la clase unión, identificado como miUnion; se ha instanciado un objeto de dicha clase, denominado mu, que puede utilizarse para almacenar un int, (4 bytes), un double (8 bytes), o un char (1 byte), pero solo uno cada vez.
El uso debe ser consistente con los valores almacenados en cada caso, cuestión esta que queda bajo responsabilidad del programador. Si se lee un dato de tipo distinto al que se escribió, el resultado obtenido es dependiente de la implementación. Una unión no puede participar en la jerarquóa de clases; no puede ser derivada de ninguna clase, ni ser una clase base. Aunque sí pueden tener un constructor y ser miembros de clases.