UNIDAD 11

Class Template en C++

Clase Plantilla.

1.- Introducción.

Una clase en C++ se diseña con la finalidad de manejar un conjunto de datos de varios tipos como una unidad. Frecuentemente la funcionalidad de una clase se puede aplicar conceptualmente para otros tipos de datos. Entonces si se construye una clase al estilo de un framework en la cual sea posible cambiar el tipo y realizar varias operaciones sobre ese tipo, acorde a las necesidades de programación. Una solución es aislar el tipo de dato dentro de la clase, y diseñar una clase la cual sea capaz de manejar un tipo de dato genérico, llamemosle (**T**). Una clase con estas características no se puede considerar realmente una clase, pero si podemos manejarla como una descripción de clase, y llamarla (**class template**) o clase plantilla. Una clase plantilla es usada por el compilador para crear una clase real en tiempo de compilación, usando un tipo de dato específico.

Las Plantillas nos permiten declarar familias de clases. La creación de una Plataforma es fácil; ya que podemos declarar una clase sin especificar el tipo de dato a manipular, (simplemente le llamamos tipo **T**).

2.- Sintaxis.

La declaración de una clase usando este método es:

```
template<class T>
class tipo_clase {
    ...
    ...
};
```

Ejemplo:

```
template<class T>
class arreglo {
    T *a;
    int tam;
public:
    arreglo(int i);
    ..
    ..
};
```

Cuando se define una función miembro fuera de la declaración de la clase, se le deberá indicar al compilador el mismo encabezado que se usó al declarar la clase.

Sintaxis.

```
template<class T>  
tipo_regresado nom_class<T>::nom_miem(list_arg)
```

Ejemplo:

```
template<class T>  
arreglo<T>::arreglo(int x) : a(new T[tam = x]) {  
    for(int i = 0; i < tam; i++) a[i] = 0;  
}
```

En este caso como se trata de un constructor, no se especifica valor de retorno.

Cuando se define un objeto de una clase template, entonces se deberá especificar el tipo, particular que va a manejar el objeto, esta especificación se hace también usando (<>).

```
nom_class<espe_de_tipo> Objeto(list_arg);
```

Con lo anteriormente espuesto es suficiente para el manejo de templates, de todos los demás detalles se encarga el compilador, creando las definiciones necesarias para el manejo del nuevo tipo y asegurar que no exista una doble definición si se usa el mismo tipo en dos archivos diferentes. De igual manera que cuando definimos un objeto de una clase común, decimos que se esta realizando una **instanciación**, ésto es válido también en el caso de templates.

Si comparamos la creación de una clase común y una clase template, para el compilador es lo mismo, con la diferencia que en el caso de templates, el compilador no realiza un análisis de error hasta el momento en que se instancia un objeto. Por esta razón algunos programadores sugieren crear una clase ordinaria antes de incluirla en un template.

A continuación se presenta un ejemplo simple del manejo de una clase declarada como clase template.

```
// Programa PLCP100.CPP  
// Una clase plataforma.
```

```
#include <iostream.h>
```

```
template<class T>
class arreglo {
    T *a;
    int tam;
public:
    arreglo(int i);
    T &operator[](int indice);
};

template<class T>
arreglo<T>::arreglo(int x) : a(new T[tam = x]) {
    for(int i = 0; i < tam; i++) a[i] = 0;
}

template<class T>
T &arreglo<T>::operator[](int indice) {
    if(indice >= 0 && indice < tam)
        return a[indice];
    clog << "Arreglo fuera de Rango" << endl;
    return *new T;
}

void main()
{
    arreglo<int> I(10);

    for(int i = 0; i < 10; i++) I[i] = 2*i;
    for(i = 0; i < 10; i++) cout << I[i] << ", ";
    cout << endl << endl;

    arreglo<float> X(10);

    for(i = 0; i < 10; i++) X[i] = 0.5*i;
    for(i = 0; i < 10; i++) cout << X[i] << ", ";
}
```

OBSERVACIONES.

La salida nos resulta:

```
0, 2, 4, 6, 8, 10, 12, 14, 16, 18,
0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5,
```

En el ejemplo anterior podemos observar que es posible manejar la clase tanto con datos enteros como con datos del tipo float.

El template para la clase **arreglo** puede manejar cualquier tipo de dato, incluyendo aquellos definidos por el usuario. El constructor pide memoria para un determinado número de apuntadores a **T** y los inicializa a CERO.

El operador sobrecargado **operator[]** regresa una referencia a un tipo arbitrario, por lo tanto es posible usando este operador, leer y escribir en el arreglo. Note que en la definición del operador se realiza una verificación de error, de tal manera de evitar que salga fuera de rango, si esto ocurre se imprime un mensaje de error. Note que después de la sentencia de error existe la siguiente sentencia.

```
*new T;
```

Esto se hace ya que la función debe regresar una referencia, y si regresamos una referencia de un objeto creado dentro de la función, la dirección regresada será inválida una vez terminada la función **operator[]**, pero si regresamos una dirección solicitada al heap, ésta va a seguir siendo válida aun después de haber salida de la función, note que se esta regresando una dirección.

Estamos regresando una referencia a un objeto válido, pero es necesario hacer notar que que el objeto creado no es parte del arreglo, si el usuario continúa con el programa va a asignar o a leer de un objeto que no es parte del arreglo, no es lo adecuado, además el usuario no sabe que este objeto fué creado en el área de almacenamiento libre, y por lo tanto no va a saber que es necesario borrarlo, por lo que este no es el mejor método de hacer las cosas, sin embargo una vez que ha aparecido el mensaje de error, es responsabilidad del usuario el que siga trabajando con el programa, ya que tiene conocimiento que ha ocurrido un problema.

3.- Una clase cadena, simple.

Para ilustrar el ejemplo de manejo de templates definamos una clase simple llamada cadena.

```
// Programa PLCP101.H  
// Una clase cadena  
  
#ifndef _CAD_H_  
#define _CAD_H_  
#include <string.h>  
#include <stdlib.h>
```

```
#include <iostream.h>

class cadena {
    char *s;
public:
    cadena(char *S = "") : s(strdup(S)) {}
    // constructor de copiado.
    cadena(const cadena &rv) : s(strdup(rv.s)) {}
    void operator=(const cadena &rv) {
        free(s);
        s = strdup(rv.s);
    }
    ~cadena() { free(s); }
    operator const char *() const { return s;}
    friend ostream& operator<<(ostream &os, const cadena &S) {
        return os << S.s;
    }
    void imprime(char *msg = "") {
        cout << msg << s << endl;
    }
};

#endif // _CAD_H_
```

El constructor **cadena(char S = "")** pide memoria al heap y copia en él la cadena. Note que la función `strdup()` pide memoria al heap usando la función `malloc()` por lo que es necesario liberar memoria usando la función `free()` en el destructor.

El constructor de copiado `cadena(cadena&)` se usa si una cadena es pasada o regresada por valor, este constructor le dice al compilador que haga una copia de la cadena bit a bit, con lo cual se tienen 2 apuntadores a la misma localidad de memoria en el heap.

El **operator=** es similar al constructor de copiado, excepto que su **lvalue** es un objeto que ya cuenta con un valor y una dirección sobre el heap. Esta memoria deberá ser destruida antes de pedir nueva memoria para hacer una copia del objeto que se encuentra al lado derecho del signo (=).

El **operator char *()** es un conversor automático de tipo, entonces si se usa el tipo `cadena` en una expresión que requiere un tipo `char *`, el compilador automáticamente convierte el objeto llamando mediante este conversor de tipo.

El **operator<<()** no es una función miembro, es una función **friend**,

Implementemos ahora una clase template llamada **cola**, mediante la cual es posible manejar varios tipos de datos, incluyendo a tipos definidos por el usuario.

```
// Programa PLCP101A.CPP
// Una cola usando templates.
#include "PLCP101.H"

template<class T>
class cola {
    enum { tam = 10 };
    T *c[tam];
    int in, fuera; // indices.
public:
    cola() :in(0), fuera(0) {}
    void suma(T *i) {
        c[in++] = i;
        if(in >= tam) in = 0;
    }
    T *obtiene() {
        if(fuera == in) return 0;
        T *result = c[fuera++];
        if(fuera >= tam) fuera = 0;
        return result;
    }
};

void main()
{
    int I[] = {1, 3, 5, 7, 11 ,13};
    cola<int> ic; // Una cola a entero
    for(int i = 0; i < sizeof(I)/sizeof(I[0]); i++)
        ic.suma(&I[i]);
    for(int *p = ic.obtiene(); p; p = ic.obtiene())
        cout << "ic.obtiene() = " << *p << endl;

    cadena S[] = { "Esto", "es", "una", "cola", "también" };
    cola<cadena> sc; // Una cola a cadena
    for(i = 0; i < sizeof(S)/sizeof(S[0]); i++)
        sc.suma(&S[i]);
    for(cadena *sp = sc.obtiene(); sp; sp = sc.obtiene())
        cout << "sc.obtiene() = " << *sp << endl;
}
```

OBSERVACIONES.

La corrida nos resulta:

```
ic.obtiene() = 1
ic.obtiene() = 3
ic.obtiene() = 5
ic.obtiene() = 7
ic.obtiene() = 11
ic.obtiene() = 13

sc.obtiene() = Esto
sc.obtiene() = es
sc.obtiene() = una
sc.obtiene() = cola
sc.obtiene() = también
```

Nos podemos dar cuenta de nuevo que mediante el uso del template podemos crear clases genéricas las cuales son capaces de manejar cualquier tipo de dato.

4.- Class template con argumentos múltiples.

El C++ acepta la declaración de class template con argumentos múltiples. Veamos un ejemplo.

En el siguiente ejemplo vamos a declarar una clase, la cual maneja 3 datos miembro; uno del tipo char *, el otro del tipo int, y el último del tipo T *, esto es, el tipo específico se va a definir en el momento de hacer la instanciación.

```
// Programa PLCP102.CPP
// Manejo de clase plataforma.
#include <iostream.h>
#include <stdlib.h>

template<class T, int i>
class MiClase {

    T* objeto;
    char *titulo;
    int tam, longitud;
public:
    MiClase(char *msg, T f);
    ~MiClase();
    int convierte();
    void muestra();
};
```

```
template<class T, int i>
MiClase<T, i>::MiClase(char *msg, T f)
{
    titulo = msg;
    tam = i;
    longitud = i;
    objeto = new T();
    *objeto = f;
}

template<class T, int a>
MiClase<T, a>::~MiClase() {
    delete objeto;
}

template<class T, int a>
int MiClase<T, a>::convierte()
{
    return atoi(titulo);
}

template<class T, int a>
void MiClase<T, a>::muestra() {
    cout << "Titulo   : " << titulo << endl;
    cout << "Tamaño   : " << tam << endl;
    cout << "Longitud: " << longitud << endl;
    cout << "Objeto   : " << *objeto << endl;
}

void main()
{
    MiClase<int, 5> X("111", 45);
    MiClase<float, 100> Y("222", 3.5F);
    cout << "Valor numérico del dato miembro \"titulo\": = ";
    cout << X.convierte() << endl;
    X.muestra();
    cout << "Valor numérico del dato miembro \"titulo\": = ";
    cout << Y.convierte() << endl;
    Y.muestra();
}
```

OBSERVACIONES.

La salida nos resulta:

```
Valor numérico del dato miembro "titulo": = 111
Titulo   : 111
Tamaño   : 5
Longitud : 5
Objeto    : 45
```

```
Valor numérico del dato miembro "titulo": = 222
Titulo   : 222
Tamaño   : 100
Longitud : 100
Objeto    : 3.5
```

Estamos manejando un argumento en la declaración de class template y 2 argumentos, en la declaración del objeto, el primero es una cadena numérica y el segundo es del tipo **T**. Los 3 argumentos van a ser utilizados para la inicialización de los datos miembro del objeto. Este proceso se lleva a cabo en el constructor, en el cual también antes de asignarle el contenido del parámetro **f** del tipo **T** al dato miembro **objeto** se pide memoria al heap para almacenar este dato.

```
template<class T, int i>
MiClase<T, i>::MiClase(char *msg, T f)
{
    titulo = msg;
    tam = i;
    longitud = i;
    objeto = new T();
    *objeto = f;
}
```

Por último tenemos el destructor, él cual libera la memoria solicitada por new.

Hay que hacer notar que en la declaración de la clase plataforma, al momento de hacer la instanciación, no se admite tener como argumento una cadena a char *, deben ser valores constantes, por ello fué necesario pasar la cadena numérica como argumento del objeto.

En el siguiente ejemplo se muestra la creación de un arreglo del tipo **T**, donde el tamaño y tipo del arreglo son dados como argumento en el momento de instanciar el objeto de la clase plataforma.

Manejemos una clase llamada **buffer**.

```
// Programa PLCP103.CPP
```

```
// Manejo de clase plataforma
#include <stdlib.h>
#include <string.h>
#include <iostream.h>

template<class T, int sz>
class buffer {
    T v[sz];
    void error(char *msg = "") {
        clog << endl << "Error en buffer: " << msg << endl;
        exit(1);
    }
public:
    T &operator[](int indice) {
        if (indice < 0 || indice >= sz)
            error("Indice fuera de rango");
        return v[indice];
    }
    void muestra(char *msg = "");
};

// Se define una función template fuera de clase
template<class T, int sz>
void buffer<T, sz>::muestra(char *msg) {
    if(*msg) cout << msg;
    for(int i = 0; i < sz; i++)
        v[i].imprime();
}

struct Char {
    char c;
    Char(char C = 0) : c(C) {}
    void imprime() { cout << c; }
};

void main()
{
    buffer<Char, 20> A;
    buffer<Char, 20> B;
    buffer<Char, 40> C;
    char a[] = "Hola muchachas";
    size_t lon = strlen(a);

    for(int i = 0; i <= lon; i++)
        A[i] = *(a+i);

    B = A;
}
```

```
    B.muestra();  
}
```

OBSERVACIONES.

La salida nos resulta: Hola muchachas

En la definición de **buffer<T, sz>::muestra()**, note que todos los argumentos deberán ser dados dentro de los (<>), ésto se hace cuando la función se define fuera de la declaración de la clase.

Puesto que **T v[sz]** dentro de la clase **buffer** es un arreglo de objetos, no un arreglo de punteros, el constructor de default (constructor sin argumentos o con argumentos de default) es llamado para cada objeto en el arreglo, lo que significa que todos los objetos de **struct Char** van a ser inicializados a CERO por su constructor cuando **Char** es usado como el primer argumento a la class template **buffer**.

La clase **buffer**, en la aplicación va a manejar una clase llamada **Char**, la cual tiene un dato miembro, para almacenar un valor del tipo **char**, esta clase también cuenta con una función para la asignación del caracter (constructor) y una función para mostrar el caracter.

La clase **Char**, cuenta con un operador de corchetes, si el indice se encuentra fuera de rango, envia un mensaje y aborta el programa, este es un método alternativo, en vez de regresar un valor del heap, aunque tiene la desventaja de que termina el programa, tan pronto ocurre un error. Esta función nos regresa una referencia, entonces por medio de este operador es posible, tanto asignar un valor al arreglo, como regresar un valor determinado de acuerdo al indice pasado como argumento.

Por último en la función **main()**, asignamos el valor del objeto **A** al objeto **B**, ésto si es posible ya que los objetos **A** y **B**, son del mismo tipo (con las mismas características), sin embargo, no sería posible asignar el valor del objeto **A** al objeto **C**. En este caso el compilador marcaría el siguiente error.

```
Coul not find a match for 'buffer<Char, 40>::buffer<Char, 20>'  
in function main()
```

Esto se puede explicar ya que cuando dos instancias de una class template no coinciden exactamente, significa que se tienen 2 tipos completamente diferentes.

En la llamada a la función **B.muestra()** para mostrar el contenido del objeto **B**, no se uso argumento, por lo tanto, esta función usa el de default. Note también que se recorre todo el arreglo, sin embargo como todo fué inicializado con CERO, y no hay una representación ASCII para este valor, no aparece nada en la pantalla.

5.- Clases derivadas y clases template.

Las clases template se pueden usar como clase base y/o como clase derivada, es posible tener las siguientes combinaciones.

- a) Una clase template derivada desde una clase template.
- b) Una clase normal derivada desde una clase template.
- c) Una clase template derivada desde una clase normal.

Estas tres técnicas se deben aprender, ya que nos brindan flexibilidad al programar. Así por ejemplo:

El primer método se puede utilizar para construir una jerarquía de clases template.

La segunda opción nos permite crear clases específicas desde clases template.

La tercera técnica nos permite crear una clase genérica a partir de una clase particular, lo que significa la creación de clases template a partir de clases normales de la librería.

a) Una clase template derivada desde una clase template.

Esto nos va a permitir que una clase template sea una clase base de una clase también template.

Para ver la sintaxis, usemos un ejemplo.

```
template <class T>
class derivadaA : public Tcoord<T> {
    T z;
public:
    derivadaA(T a, T b, T c) : z(a), Tcoord<T>(b, c) {}
};
```

La clase derivadaA hereda de la clase Tcoord<T>. El constructor de la clase derivadaA requiere de 3 parámetros del tipo T, el parámetro **a** se usa para inicializar el dato miembro **z** del tipo T, los otros dos parámetros son pasados al constructor de la clase base referida por el nombre Tcoord<T>.

Cuando el programa usa la clase derivadaA, el compilador genera 2 clases distintas -una clase base Tcoord<T> (donde T es remplazado por el tipo apropiado) y una clase derivadaA<T>-.

Se puede instanciar un objeto de esta clase mediante la sentencia:

```
derivadaA<int> A(1, 2, 3>;
```

b) Una clase normal derivada desde una clase template.

Se usa esta técnica cuando se cuenta con una clase template y se desea que el compilador la sobre-escriba y genere una clase normal, y además se puedan agregar algunos elementos en la clase derivada.

Para ver la sintaxis, usemos un ejemplo.

```
class derivadaB : public Tcoord<int> {  
    int z;  
public:  
    derivadaB(int a, int b, int c)  
        : z(a), Tcoord<int>(b, c) {}  
};
```

De la misma forma que en una clase normal, es posible reemplazar la palabra `public` por `private` o `protected`, también se puede construir herencia múltiple. Note que la clase base template cuenta ya con un tipo específico `Tcoord<int>`. La clase derivada no es una clase template, por lo que, tanto sus datos miembros como los parámetros de sus funciones miembro, deberán tener tipos definidos.

Se puede instanciar un objeto de esta clase mediante la sentencia:

```
derivadaB B(4, 5, 6);
```

c) Una clase template derivada desde una clase normal.

Es posible generar una clase template desde una clase normal, este método nos permite convertir clases normales de la librería en clases template.

Para ejemplificar la sintaxis, usemos la clase anteriormente declarada **derivadaB**, para generar una nueva clase **derivadaC<T>**, la cual hereda de **derivadaB**.

```
template <class T>  
class derivadaC : public derivadaB {  
    T q;  
public:  
    derivadaC(int a, int b, int c, T d)  
        : q(d), derivadaB(a, b, c) {}  
};
```

Note que cuando se declara una clase template debe ir al inicio de la declaración la frase **template <class T>**.

Su dato miembro es del tipo genérico T, la llamada al constructor de la clase base se hace con tipos específicos, en este caso **int**.

Podemos concluir, diciendo que una clase template, no necesariamente tiene que ser la raíz del árbol de herencia, sin embargo, mediante el uso de herencia, cualquier clase puede convertirse en una clase template, y cualquier clase template puede transformarse en una clase normal.

A continuación se presenta un programa ejemplo completo en donde usamos las clases definidas con anterioridad.

Primero tenemos el archivo PLCP103A.H, el cual contiene la declaración de la clase template Tcoord y la definición de sus funciones miembro.

```
// Archivo de definición de la clase Tcoord
#ifndef _PLCP103A_H
#define _PLCP103A_H

#include <iostream.h>

template<class T>
class Tcoord {
    T x, y;
public:
    Tcoord();
    Tcoord(T a, T b);
    ~Tcoord();
    T getx();
    T gety();
    friend ostream& operator<<(ostream &a, Tcoord<T> &b);
    friend istream& operator>>(istream &a, Tcoord<T> &b);
};

// Funciones friend de la clase Tcoord
template <class T>
ostream& operator<<(ostream &a, Tcoord<T> &b)
{
    a << '(' << t.x ' , ' << t.y << ')' ' << endl;
    return a;
}

template <class T>
istream& operator>>(istream &a, Tcoord<T> &b)
{
    cout << "x = ";
```

```
        a >> t.x;
        cout << "y = ";
        a >> t.y;
        return a;
    }

template <class T>
Tcoord<T>::Tcoord()
{
    x = 0;
    y = 0;
}

template <class T>
Tcoord<T>::Tcoord(T a, T b) : x(a), y(b) {}

template <class T>
Tcoord<T>::~~Tcoord()
{
    cout << "Se ejecuta destructor clase \"Tcoord\"" << endl;
}

template <class T>
T Tcoord<T>::getx() { return x; }

template <class T>
T Tcoord<T>::gety() { return y; }

#endif // _PLCP101A_H
```

Note que en el constructor de Tcoord, sin argumentos, se asigna a **x** y a **y** el valor de CERO, ésto nos lleva a limitar el tipo de **T** a valores numéricos.

A continuación definimos las clases derivadas, en el archivo PLCP101A.CPP.

```
// Programa PLCP103A.CPP
// Clases derivadas y derivación de clases template

#include <iostream.h>
#include "plcp103a.h"

// Clase template derivada de una clase template
template <class T>
class derivadaA : public Tcoord<T> {
    T z;
public:
    derivadaA(T a, T b, T c) : z(a), Tcoord<T>(b, c) {}
    void muestra(char *m = "") {
        cout << m << endl;
    }
};
```

```
        cout << "z = " << z << endl;
        cout << "x = " << getx() << endl;
        cout << "y = " << gety() << endl;
    }
};

// Clase no template derivada de una clase template
class derivadaB : public Tcoord<int> {
protected:
    int z;
public:
    derivadaB(int a, int b, int c)
        : z(a), Tcoord<int>(b, c) {}
    virtual void muestra(char *m = "");
};

// Clase template derivada de una clase no template
template <class T>
class derivadaC : public derivadaB {
    T q;
public:
    derivadaC(int a, int b, int c, T d)
        : q(d), derivadaB(a, b, c) {}
    virtual void muestra(char *m = "") {
        cout << m << endl;
        cout << "z = " << z << endl;
        cout << "x = " << getx() << endl;
        cout << "y = " << gety() << endl;
        cout << "q = " << q << endl;
    }
};

void derivadaB::muestra(char *m) {
    cout << m << endl;
    cout << "z = " << z << endl;
    cout << "x = " << getx() << endl;
    cout << "y = " << gety() << endl;
}

void imprime(derivadaB &X, char *cad)
{
    X.muestra(cad);
}

void main()
{
    derivadaA<int> A(1, 2, 3);
    derivadaB B(4, 5, 6);
}
```

```
derivadaC<double> C(7, 8, 9, 10.55);  
A.muestra("Clase derivadaA");  
imprime(B, "Clase derivadaB");  
imprime(C, "Clase derivadaC");  
}
```

OBSERVACIONES.

Se agregó a cada clase una función muestra(), de tal forma de poder presentar en la pantalla los valores almacenados en cada clase.

Como la clase derivadaB es base de derivadaC, en este caso la función muestra() se definió como virtual. En el caso de la clase derivadaA, no se hizo, ya que solo hereda de la clase Tcoord.

Veamos la salida:

```
Clase derivadaA  
z = 1  
x = 2  
y = 3  
Clase derivadaB  
z = 4  
x = 5  
y = 6  
Clase derivadaC  
z = 7  
x = 8  
y = 9  
q = 10.55  
Se ejecuta destructor clase "Tcoord"  
Se ejecuta destructor clase "Tcoord"  
Se ejecuta destructor clase "Tcoord"
```

Para mostrar los valores de **x** y de **y** se pudo haber utilizado el operador sobrecargado (<<), sin embargo la otra forma era utilizar las funciones **getx()** y **gety()**, miembros de la clase Tcoord, y ésto último fue lo que se hizo.

El constructor de la clase Tcoord fue llamado 3 veces, una vez por cada objeto A, B y C por lo tanto, se observa que el destructor es llamado también 3 veces al salir del ámbito de main().

Se deja como ejercicio la construcción del árbol de herencia.

6.- Funciones templates.

Las Plataformas no solo pueden ser utilizadas con clases, también se pueden utilizar con funciones. Nos podemos encontrar situaciones en las cuales la misma función deberá ser utilizada pero con diferentes tipos de argumento. Los programadores

en C típicamente resuelven este problema mediante el uso de macros, pero esta solución no es ideal. Por ejemplo, considere la necesidad de determinar la mayor de 2 cantidades. La siguiente macro, puede manejar todos los tipos de datos.

```
#define max(a, b) ((x > y) ? x : y)
```

Esta macro fué utilizada por años, pero involucra algunos problemas:

La macro trabaja pero no hay verificación por parte del compilador, de los tipos de datos usados en la comparación, si por ejemplo, se invoca a la macro con un entero y un puntero a char (char *), el compilador no va a generar ninguna advertencia o error.

Para que el compilador pueda verificar tipos, es necesario escribir una función que realice la comparación.

```
int max(int a, int b)
{
    return a > b ? a : b;
}
```

Pero la función solo va a manejar valores del tipo entero, o datos que pueden ser procesados como del tipo entero. La función no es capaz de manejar otro tipo de dato; tales como float o puntero a char (char *).

Podemos escribir también la siguiente función:

```
double max(double a, double b)
{
    return a > b ? a : b;
}
```

Esta función deberá manejar todos los tipos de datos, desde char hasta double, teniendo el compilador que realizar una promoción entera en los argumentos. Existen dos limitaciones sobresalientes en la función.

- 1.- El valor de retorno es siempre un double, independientemente del tipo de los argumentos pasados a la función. Esto nos puede causar un problema, considere el siguiente código.

```
// Programa PLCP104.CPP
// funciones plataforma.
#include <iostream.h>

double max(double a, double b)
{
    return a > b ? a : b;
}
```

```
void main()
{
    cout << "max('3', '5') es = " << max('3', '5')
        << endl;
}
```

OBSERVACIONES.

La salida debería de resultar: max('3', '5') es = 5

La salida resulta: max('3', '5') es = 53

El problema estriba en que el operador de inserción asume que esta trabajando con un double e imprime su valor numérico 53, en vez del caracter 5. Una solución a este problema sería la inclusión de un cast.

- 2.- Solo tipos de datos que pueden tener promoción a double pueden ser manejados, esta función no será capaz de manejar tipos como estructuras, punteros, etc.

6.a) La función template.

C++ usa una nueva manera de declaración para implementar funciones como max(...). A esta forma se le conoce como **función template**, mediante este método se crea una generalización de la función max(...).

La declaración se hace como sigue:

```
template<class T> T max( T a, T b)
{
    return a > b ? a : b;
}
```

La declaración inicia con la palabra **template**, la cual es una palabra reservada de C++. Luego continúa con la expresión <class T> la cual le indica al compilador, que alguna variable de nombre genérico **T** (deberá usar cualquier identificador que desee poner en lugar de **T**) va a ser usada en la función. Con esto la definición de la función resulta de la siguiente manera.

```
template<class uno>
uno max(uno a, uno b)
{
```

```
    return a > b ? a : b;  
}
```

Una función de este tipo, no esta restringida a manejar variables de una clase, se puede utilizar cualquier tipo de datos semejante como, estructuras, punteros, enteros, etc.

A continuación se muestra el ejemplo completo:

```
// Programa PLCP105.CPP  
// funciones plataforma.  
#include <iostream.h>  
  
template<class uno>  
uno max(uno a, uno b)  
{  
    return a > b ? a : b;  
}  
  
void main()  
{  
    cout << "max('3', '5') es = " << max('3', '5')  
        << endl;  
    cout << "max(3, 5) es = " << max(3, 5)  
        << endl;  
    cout << "max(3.0, 5.0) es = " << max(3.0, 5.0)  
        << endl;  
}
```

OBSERVACIONES.

La salida nos resulta:

```
max('3', '5') es = 5  
max(3, 5) es = 5  
max(3.0, 5.0) es = 5
```

Note que los tres valores son iguales.

Cuando el compilador se encuentra con una llamada a una función de este tipo, crea una función para manipular los tipos de datos usados como argumento en la llamada a la función. No se intenta la realización de ningún tipo de conversión implícito,

tal como una promoción entera. Si el primer argumento es un entero, el segundo argumento deberá también ser entero, si el primer argumento es double, el segundo también deberá ser double. El primer argumento determina el tipo de función generada.

Mediante el uso de este tipo de funciones, se libera al programador, de tener que definir una función para cada determinado tipo de argumentos.

6.b) Sobrecarga de funciones template.

Es posible también sobrecargar una función template de la misma manera que se hace con otro tipo de funciones.

Analicemos el siguiente código.

```
// Programa PLCP106.CPP
// funciones plataforma.

#include <stdlib.h>
#include <iostream.h>

template<class uno>
uno max(uno a, uno b)
{
    return a > b ? a : b;
}

int max(char *a, char *b)
{
    int x, y;
    x = atoi(a);
    y = atoi(b);
    return x > y ? x : y;
}

void main()
{
    cout << "max('3', '5')    es    = " << max('3', '5')
        << endl;
    cout << "max(3, 5)        es    = " << max(3, 5)
        << endl;
    cout << "max(3.0, 5.0)    es    = " << max(3.0, 5.0)
        << endl;
    cout << "max(\"33\", \"55\") es    = " << max("33", "55")
}
```

```
    << endl;  
}
```

OBSERVACIONES.

La salida nos resulta:

```
max('3', '5')   es   = 5  
max(3, 5)       es   = 5  
max(3.0, 5.0)   es   = 5  
max("33", "55") es   = 55
```

Cuando se llama a la función `max(char *, char *)` en el programa, el compilador no intenta generar una función template. para este caso, va a usar de manera explicita la notación `max(char *, char *,` cuya definición se encuentra en el código fuente.

6.c) Un ejemplo práctico de uso de funciones template.

Cuantas veces se escribe y vuelve a escribirse la misma función de ordenamiento de acuerdo al tipo de dato que se quiere ordenar. Mediante el uso de funciones template es posible escribir una sola función genérica de ordenamiento y utilizarla dependiendo del tipo de dato a ordenar.

A continuación se presenta un ejemplo.

```
// Programa # 107, Uso de funciones template  
// Función genérica de ordenamiento  
#include <iostream.h>  
#include <iomanip.h>  
#include <stdlib.h>  
#include <time.h>  
  
// Una clase simple  
class dato {  
    int x, y;  
public:  
    dato(): x(0), y(0) { }  
    dato(int n): x(n), y(rand()) { }  
    friend ostream & operator<< (ostream &, dato &);  
    friend int operator< (dato &, dato &);  
};
```

```
// Sobrecarga del operador << para la clase dato
ostream & operator<< (ostream &os, dato &t)
{
    cout << '(' << t.x << ", " << t.y << ')';
    return os;
}

// Regresa True si objeto es "menor" que objeto
// Ordena el elemento 'x'
operator< (dato &a, dato &b)
{
    return (a.x < b.x);
}

// Llena el arreglo con valores aleatorios
template<class T>
void llena_arre(T *arreglo, int len)
{
    srand((unsigned)time(0)); // valor aleatorio
    for (int i = 0; i < len; i++) // Llena arreglo
        arreglo[i] = (T)rand();
}

// Muestra el contenido del arreglo
template<class T>
void muestra(T *arreglo, int len)
{
    cout << endl;
    for (int i = 0; i < len; i++)
        cout << arreglo[i] << endl;
}

// Algoritmo Quicksort por C. A. R. Hoare
template<class T>
void Quicksort(T *array, int left, int right)
{
    int i = left;
    int j = right;
    T test = array[(left + right) / 2];
    T swap;

    do {
        while (array[i] < test) i++;
        while (test < array[j]) j--;

```

```
        if (i <= j) {
            swap = array[i];
            array[i] = array[j];
            array[j] = swap;
            i++;
            j--;
        }
    } while (i <= j);
    if (left < j) Quicksort(array, left, j);
    if (i < right) Quicksort(array, i, right);
}

// Ordena un arreglo de n elementos
template<class T>
void ordena(T *arreglo, int len)
{
    if (len > 1) Quicksort(arreglo, 0, len - 1);
}

#define TAM 10

void main()
{
    // Para probar el programa habilite una de las siguientes
    // declaraciones de arreglo[]
    // y ordene un arreglo de int, double, char, u objetos tipo dato

    // int arreglo[TAM];
    // double arreglo[TAM];
    // dato arreglo[TAM]; // Dato activo.

    // Actualiza banderas
    cout.setf(ios::showpoint);
    cout.precision(12);

    // Llena, muestra, ordena y muestra
    llena_arre(arreglo, TAM);
    muestra(arreglo, TAM);
    ordena(arreglo, TAM);
    muestra(arreglo, TAM);
}
```

OBSERVACIONES.

En el ejemplo se manejan 3 arreglos; uno de enteros, uno de double, y uno de objetos del tipo **dato**.

Por simplicidad del programa se se siguen los siguientes pasos:

- 1) Llenar de manera aleatoria cada uno de los arreglos con 10 elementos.
- 2) Mostrar el arreglo como fue llenado.
- 3) Ordenar el arreglo.
- 4) Mostrarlo ahora de manera ordenada.

La salida del programa para elementos del tipo **dato** es como sigue:

No ordenado	Ordenado
(10179, 30187)	(6906, 19356)
(26912, 13006)	(8036, 20257)
(6906, 19356)	(10179, 30187)
(10369, 10357)	(10369, 10357)
(31526, 16991)	(13840, 20151)
(13840, 20151)	(14621, 11923)
(19977, 28087)	(19977, 28087)
(14621, 11923)	(26912, 13006)
(8036, 20257)	(29353, 22025)
(29353, 22025)	(31526, 16991)

NOTA: El acomodo se hizo para ahorrar espacio.

La clase **dato** es perfectamente aceptada como un remplazo del parámetro **T**. Cuando se pasa una objeto de una clase como argumento de una función template, se deberá considerar la inclusión de ciertas sentencias, por ejemplo, para nuestro cas:.

- 1) Un constructor de default para iniciar el arreglo.
- 2) Un constructor sobrecargado con un solo parámetro de tal forma de poder ser utilizado como conversor de tipo en la sentencia (T)rand();, en este caso, el constructor asigna a 'x', el valor regresado por rand(), y este valor nos sirve para generar y.

- 3) Un operador (<<) sobrecargado para mandar a pantalla los elementos del arreglo del tipo **dato**.
- 4) Una operador (<) sobrecargado para poder comparar dos objetos.

Se pueden requerir otras sentencias, como por ejemplo la sobrecarga del operador (==), para ver si dos objetos son iguales, etc., sin embargo en este caso, como muestra solo se consideró el operador de comparación (<.

7.- Donde almacenar las funciones templates.

En la mayoría de los compiladores actuales, este tipo de funciones deben ser almacenadas en archivos de cabecera, y para usarlos solo basta con incluir dichos archivos, ésto debe hacerse así ya que el compilador necesita "ver" la función template antes de usarla.

Este tipo de funciones no puede ser almacenada, por lo general, en librerías o módulos separados, como sucede con las funciones reales.

Tal vez con el avance de los compiladores, en un tiempo no muy lejano ésto sea posible.