

Vigésimo tercera Sesión

Metodologías y Técnicas de Programación II

Programación Orientada a Objeto (POO) C++

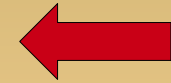
Plantillas II

Estado del Programa

Introducción a la POO

Historia de la Programación
Conceptos de POO

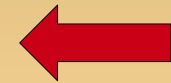
C++
Mi primera Clase



Repaso de Conceptos

Estándares de Programación
Punteros y Memoria

E/S
Control y Operadores



Clases y Objetos en C++

Uso y aplicación
Constructores
Constantes e "inline"

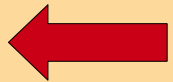
Funciones Amigas
Sobrecarga de Funciones



Sobrecarga

De Operadores

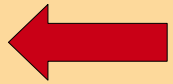
Creación Dinámica de Objetos



Herencia.

Tipos de Visibilidad

Herencia Múltiple



Polimorfismo

Funciones Virtuales

Polimorfismo y Sobrecarga.



Plantillas

Contenedores

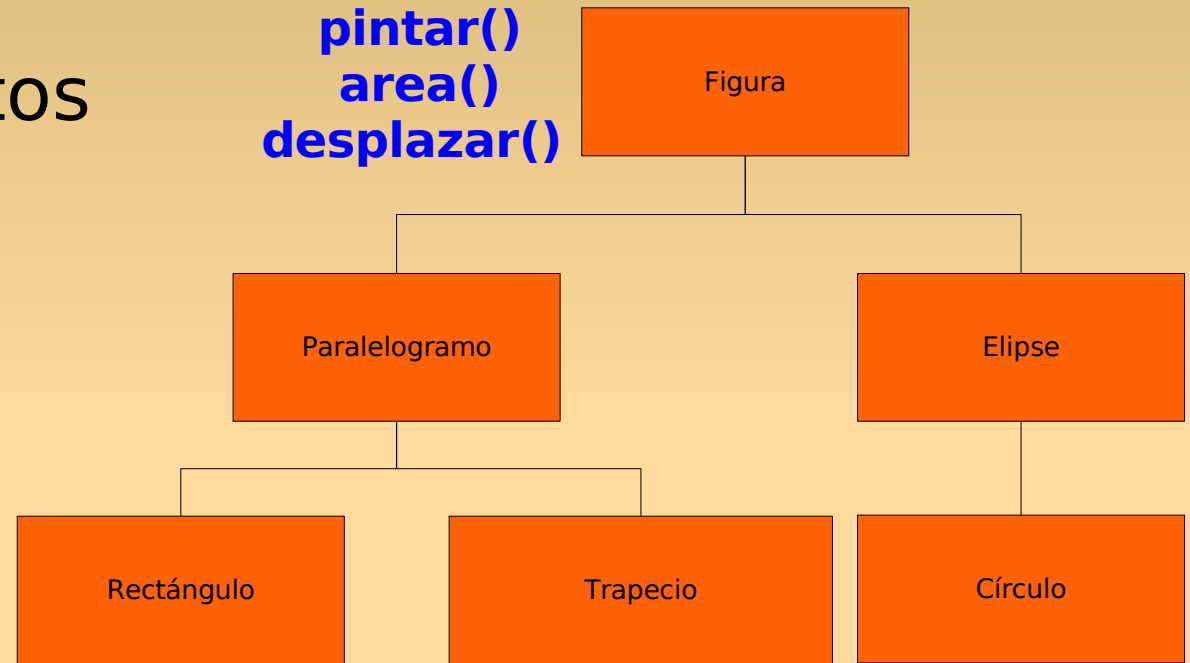
Iteradores



23.1 Repaso

Programación Orientada a Objeto:

- Abstracción de Datos
- Herencia
- Polimorfismo



Sobrecarga (De funciones y de Operadores)

Posibilidad de declarar métodos con el mismo nombre que pueden tener diferentes argumentos dentro de una misma clase.

23.1 Repaso

Plantillas de Funciones – Funciones Genéricas - Templates

Las funciones genéricas son un mecanismo C++ que permite definir una función mediante uno o varios parámetros (tipos genéricos).

A partir de estas plantillas el compilador es capaz de generar código de funciones distintas que comparten ciertas características.

Las funciones así generadas se denominan:

Instancias o **especializaciones** de la plantilla.

La plantilla representa un **número indefinido** de funciones.

Depende de cuántas Instancias creemos o utilizemos el compilador generará más o menos código.

```
template <class T>
T maximo (T a, T b )
{ .....};

//Instancias:
int i1=1, i2=20, i3;
i3 = maximo(i1,i2);

float f1=10.03, f2=2.2, f3;
f3 = maximo(f1,f2);

Alumno a1("Pepe",8), a2("Ana",8);
Alumno a3;
a3 = maximo(a1,a2);
```

23.2 Plantillas

Métodos Genéricos

También podemos utilizar funciones genéricas como métodos de clases.

```
// Declaración o Prototipo (.h)
class MiClase
{
    public:
        template <class T>
        T maximo (T a, int b);
        ....
    private:
        ....
        ....
}
```

```
// Implementación (.cpp)
template <class Tipo>
MyClase::maximo ( Tipo a, Tipo b )
{
    if (a<b)
        return b;
    return a;
}
```

```
// Uso (test.cpp)
MiClase objeto;
int max_int;
float max_f, g = 3.3;

max_int = objeto.maximo(5,5);
max_f   = objeto.maximo(g,88);
```

23.2 Plantillas

Plantillas de Clases o Clases Genéricas

Podemos definir clases mediante uno o varios parámetros que hacen referencia a un tipo de dato.

Podemos generar un montón de clases partiendo de un diseño común.

Partiendo de las mismas líneas de código genero muchas y diferentes clases. Otra vez la reutilización con sus ahorros de esfuerzo y reducción de equivocaciones.

Las clases generadas a partir de una plantilla se denominan **Instancias** o **Especializaciones** de la plantilla.

Como siempre, tratamos de resolver o simplificar un problema.

23.2 Plantillas

Plantillas de Clases o Clases Genéricas

El problema a resolver:

```
class Punto
{
    private:
        int coorx ;
        int coory ;
    public:
        ...
        ...
};
```

Puede ser necesario implementar esta clase de forma que las coordenadas necesiten decimales.

La opción directa – y con muchos inconvenientes – ya la conocemos:

Copiar y Pegar

23.2 Plantillas

Plantillas de Clases o Clases Genéricas

```
class PuntoEntero
{
private:
    int coor
    int coor
public:
    ...
    ...
};
```

```
class PuntoLong
{
private:
    long int coorx ;
    long int coory ;
public:
    ...
    ...
};
```

```
class PuntoFloat
{
private:
    float coorx ;
    float coory ;
public:
    ...
    ...
};
```

```
template class <T>
class Punto
{
private:
    T coorx ;
    T coory ;
public:
    ...
    ...
};
```

Con la Plantillas: Podemos definir una **clase genérica** desde la que crear **Especializaciones** o **Instancias** para determinados tipos de dato o clases.

23.2 Plantillas

Sintaxis

```
template <class|typename <id>[,...]>  
class <identificador_de_plantilla>  
{  
    // Declaración de funciones  
    // y datos miembro de la plantilla  
};
```

```
// Declaración (.h)  
template class <T>  
class Punto  
{  
    private:  
        T coorx ;  
        T coory ;  
    public:  
        Punto(int n);  
        ~Punto();  
};
```

```
// Implementación (.cpp)  
template <class T>  
Punto<T>::Punto(int n)  
{  
    coorx = n;  
    coory = n;  
}
```

```
// Uso (main.cpp)  
Punto <int>    objeto_punto_ent(3);  
Punto <float>  objeto_punto_flo(5);  
.....
```

23.2 Plantillas

Plantillas para Clases Genéricas

Representación UML:



Las funciones miembro son a su vez plantillas que pueden utilizar los mismos parámetros que la plantilla de la clase a la que pertenecen.

```
template class <T>
class Punto
{
    private:
        T coorx_ ;
        T coory_ ;
    public:
        Punto(T a, T b);
        T coorx();
        T coory();
        void coorx(T c);
        void coory(T c);
        void mostrar();
};
```

// Implementación (.cpp)

```
template <class T>
Punto<T>::Punto(T a, T b)
{
    coorx = a;
    coory = b;
}
//-----
template <class T>
T Punto<T>::coorx()
{
    return coorx;
}
```

23.2 Plantillas

Consideraciones para el diseño de plantillas

- Diseñar primero para un tipo de datos en concreto: Podemos pensar primero en clase Punto para enteros.
- Implementar primero la clase para el caso particular: Codificar lo necesario para Punto con enteros.
- Convertirla en una clase genérica: Añadimos la palabra reservada “template” donde haga falta y sustituimos tipos.

Es más sencillo ir de lo particular a lo general

23.3 Ejemplos Plantillas

Ejemplo

Diseñar una clase Ejemplo1 que tenga dos atributos de tipo genérico (a y b) cada uno de un tipo. Debe tener dos métodos:

Un constructor que recibe dos parámetros, cada uno para asignar valor a cada uno de los atributos.

Un método mostrar() que debe sacar por pantalla lo siguiente:

Valor de a: _____

Valor de b: _____

El programa principal para probarlo es este:

```
void main( )
{
    Ejemplo1 <int, char> obj1(5, 'p') ;
    Ejemplo1 <bool, float> obj2(true, 9.0) ;
    obj1.visualizar( );
    obj2.visualizar( );
}
```

Compilar sin errores y mostrar el resultado por pantalla.

23.3 Ejemplos Plantillas

Arrays genéricos

Las plantillas son muy útiles para definir estructuras que albergan objetos. Estas estructuras tienen un comportamiento general que es independiente del tipo de dato que manejan.

Un lista de Objetos encadenados tiene los mismos atributos y tiene un método añadir() y otro eliminar() que debe valer tanto para Alumnos como para Coches.

Los arrays por defecto de C++ no hacen comprobaciones de fuera de rango.

```
int a[100];
int i = 1508;
.....
a[i]= 73;    // Nos hemos salido de la memoria asignada!!
```

Podemos construir un clase Array_de_Enterros que nos ayude a proteger nuestro código de errores.

23.3 Ejemplos Plantillas

Arrays Genéricos

```
class ArrayInt
{
public:
    ArrayInt(int nElem);
    ~ArrayInt();
    int& operator[](int indice)
    {
        return pInt[indice];
    }
private:
    int *pInt;
    int nElementos;
};
// Definición:
ArrayInt::ArrayInt(int nElem) :
    nElementos(nElem)
{
    pInt = new int[nElementos];
}
ArrayInt::~~ArrayInt()
{
    delete[] pInt;
}
```

```
int main()
{
    ArrayInt TablaI(10);

    for(int i = 0; i < 10; i++)
    {
        TablaI[i] = 10-i;
    }
    for(int i = 0; i < 10; i++)
    {
        cout << TablaI[i] << endl;
    }
}

// Podemos hacer cosas como esta
// que no se pueden hacer con los
// arrays por defecto:
int elementos = 700;
ArrayInt arr(elementos);
```

23.3 Ejemplos Plantillas

Arrays Genéricos: Cadenas

```
template <class T>
class Array
{
public:
    Array(int nElem);
    ~Array();
    T& operator[](int indice)
    {
        return pT[indice];
    }
    const int NElementos()
    {
        return nElementos;
    }
private:
    T *pT;
    int nElementos;
};
//-----
template <class T>
Array<T>::Array(int nElem) :
    nElementos(nElem)
{
    pT = new T[nElementos];
}
```

```
template <class T>
Array<T>::~~Array()
{
    delete[] pT;
}
//-----
const int nElementos = 10;
int main()
{
    Array<int> ArrayInt(nElementos);
    Array<float> ArrayFloat(nElementos);
    for(int i = 0; i < nElementos; i++)
        ArrayInt[i] = nElementos-i;
    for(int i = 0; i < nElementos; i++)
        ArrayFloat[i] = 1/(1+i);
    for(int i = 0; i < nElementos; i++)
    {
        cout << "ArrayInt[" << i << "] = "
              << ArrayInt[i] << endl;
        cout << "ArrayFloat[" << i << "] = "
              << ArrayFloat[i] << endl;
    }
    return 0;
}
```

23.3 Ejemplos Plantillas

Arrays Genéricos: Cadenas

```
cont int nElementos = 10;
int main()
{
    Array<char *> ArrayCad(nElementos);
    char cadena[20];
    for(int i = 0; i < nElementos; i++)
    {
        sprintf(cadena, "Numero: %5d", i);
        ArrayCad[i] = cadena;
    }
    strcpy(cadena, "Modificada");
    for(int i = 0; i < nElementos; i++)
        cout << "ArrayCad[" << i << "] = "
            << ArrayCad[i] << endl;
    system("pause");
    return 0;
}
```

¿Cuál es la salida por pantalla?

¿Por qué no “funciona”?

La solución pasa por crear nuestra propia clase cadena con un operador asignación que haga lo correcto y con su constructor copia.

23.3 Ejemplos Plantillas

Arrays Genéricos: Cadenas

```
class Cadena
{
public:
    Cadena(char *cad)
    {
        cadena = new char[strlen(cad)+1];
        strcpy(cadena, cad);
    }
    Cadena() : cadena(0) {}
    Cadena(const Cadena &c) :
        cadena(0) {*this = c;}
    ~Cadena()
    {
        if(cadena)
            delete[] cadena;
    }
    const char* Lee() const
    {
        return cadena;
    }
private:
    char *cadena;
};
```

```
Cadena &operator=(const Cadena &c)
{
    if(this != &c)
    {
        if(cadena)
            delete[] cadena;
        if(c.cadena)
        {
            cadena =
                new char[strlen(c.cadena)+1];
            strcpy(cadena, c.cadena);
        }
        else
            cadena = NULL;
    }
    return *this;
}
ostream& operator<<(ostream &os, const
Cadena& cad)
{
    os << cad.Lee();
    return os;
}
```

23.3 Ejemplos Plantillas

Arrays Genéricos: Cadenas

Ahora podemos crear un Array para cadenas utilizando esta clase y nuestra plantilla.

```
int main()
{
    Tabla<Cadena> TablaCad(nElementos);
    char cadena[20];
    for(int i = 0; i < nElementos; i++)
    {
        sprintf(cadena, "Numero: %2d", i);
        TablaCad[i] = cadena;
    }
    strcpy(cadena, "Modificada");
    for(int i = 0; i < nElementos; i++)
        cout << "TablaCad[" <<i<< "] = "
            << TablaCad[i] << endl;
    system("pause");
    return 0;
}
```

Compilar y mostrar salida por pantalla.