

Vigésimo segunda Sesión

Metodologías y Técnicas de Programación II

Programación Orientada a Objeto (POO) C++

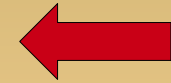
Plantillas I

Estado del Programa

Introducción a la POO

Historia de la Programación
Conceptos de POO

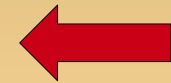
C++
Mi primera Clase



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"

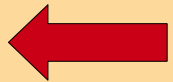
Funciones Amigas
Sobrecarga de Funciones



Sobrecarga

De Operadores

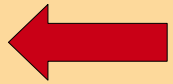
Creación Dinámica de Objetos



Herencia.

Tipos de Visibilidad

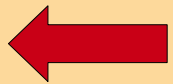
Herencia Múltiple



Polimorfismo

Funciones Virtuales

Polimorfismo y Sobrecarga.



Plantillas

Contenedores

Iteradores



22.1 Repaso

Programación Orientada a Objeto:

Buscamos **resolver problemas**:

- ◆ Tenemos unos datos de partida.
- ◆ Aplicamos unos algoritmos.
- ◆ Obtenemos un resultado.

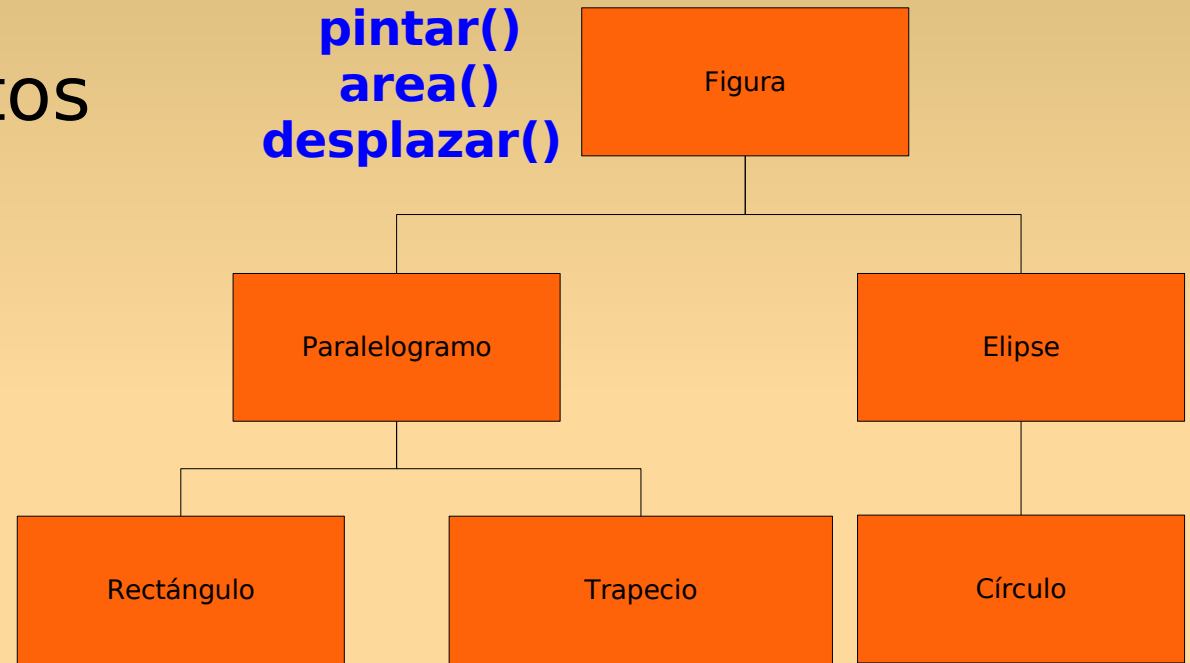
Queremos:

- Optimizar nuestro esfuerzo.
- Utilizar lo que han hecho otros o nosotros mismos.
- Que sea eficiente.
- Hacer nuestro código lo más genérico posible.

22.1 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.

22.2 Plantillas de Funciones (Templates)

Haciendo nuestro código genérico:

Nos hemos encontrado en situaciones donde vemos claramente que lo deseable es que **una operación** (una función, un método) sea válida con **diferentes tipos de datos**.

Sobrecargando funciones y operadores:

- La suma para enteros.

- La suma para números complejos.

- La suma para Alumnos.

Utilizando el polimorfismo:

- Con un puntero a una clase base y una jerarquía podíamos

- “tocar_una_nota()” para:

 - Clarinete....

 - Guitarra....

La sobrecarga tiene un inconveniente:

Tengo que definir tantas funciones como tipos de datos haya

22.2 Plantillas de Funciones (Templates)

Plantillas

Las Plantillas nos permiten definir funciones genéricas.

Mayor reutilización de código.

EL PROBLEMA

```
int maximo(int a, int b )
{
    if (a<b)
        return b;
    return a;
}

float maximo(float a, float b )
{
    if (a<b)
        return b;
    return a;
}
```

Repetimos el código de la función:

Reutilización de “copia-pega”
Aplicamos sobrecarga de funciones.

Problemas:

Posibilidad de errores.
Replicación de problemas.
Código que no está centralizado.

¿Si queremos el máximo para *doubles*?

22.2 Plantillas de Funciones

Plantillas de Funciones

```
int maximo(int a, int b )
{
    if (a<b)
        return b;
    return a;
}
float maximo(float a, float b )
{
    if (a<b)
        return b;
    return a;
}
template <class UnTipo>
UnTipo maximo (UnTipo a, UnTipo b)
{
    if (a<b)
        return b;
    return a;
}
```

Las funciones maximo() tienen el mismo cuerpo.

Se diferencian en los prototipos, en la declaración. Claro eso es porque los tipos de los argumentos son diferentes.

Parece interesante y útil:

“Tener una función genérica capaz de calcular el máximo de dos valores de cualquier tipo.”

C++ nos permite esto mismo definiendo funciones genéricas (para cualquier tipo) mediante el uso de Plantillas - *Templates*.

22.2 Plantillas de Funciones

Funciones genéricas utilizando plantillas.

- Una función genérica define un conjunto de operaciones que se aplicarán a diferentes tipos de datos.
- Una plantilla de funciones **es como un patrón**.
- Una plantilla **especifica un conjunto infinito de funciones** que pueden ser aplicadas a distintos tipos de datos.
- Una plantilla describe las **propiedades genéricas** de una función.

```
template <class UNTIPO>
UNTIPO maximo ( UNTIPO a, UNTIPO b )
{
    if (a<b)
        return b;
    return a;
}
```

```
...
int a=5, int b=8, c;
float d=3.0, e=5.1, f;
c=maximo(a,b);
f=maximo(d,e);
...
```

22.2 Plantillas de Funciones

Plantillas de Funciones:

```
template <class <id>[,...]>  
<tipo_retorno> <identificador>(<lista_de_parámetros>)  
{  
    // Declaración de función  
};
```

La palabra reservada **template** indica que se va a declarar una plantilla. Se declaran en un archivo de cabecera (.h).

Se puede poner cualquier tipo sea clase o no (enteros, char,...)

```
template <class TIPO1>  
TIPO1 funcion ( TIPO1 a)  
{  
    // cuerpo de la función.  
}  
  
template <class TIPO1>  
TIPO2 funcion ( int cont, TIPO1 a)  
{  
    // cuerpo de la función.  
}
```

```
template <class TIPO1>  
TIPO1 funcion ( TIPO1 a, TIPO1 b)  
{  
    // cuerpo de la función.  
}  
  
template <class TIPO1, class TIPO2 >  
TIPO2 funcion ( TIPO1 a, TIPO2 b)  
{  
    // cuerpo de la función.  
}
```

22.2 Plantillas de Funciones

Plantillas de Funciones

- Una función genérica define un conjunto ilimitado de funciones sobrecargadas.
- Las plantillas no generan código directamente. El código lo genera el compilador en el punto que ve que se utiliza una plantilla.
- Se siguen las normas de la sobrecarga. Sólo se mira la lista de paámetros no el retorno.
- Los tipos tienen que aparecer al menos una vez en la lista de argumentos de la función.

```
template <class UNTIPO>
UNTIPO maximo ( int a, float b ) // ERROR ¿POR QUÉ?
{
    .....
}
```

22.2 Plantillas de Funciones

Sobrecarga de Plantillas de Funciones

¿Puedo definir varias veces y con el mismo nombre una función genérica?

Sí, aplico las mismas normas de la sobrecarga de funciones “normales”.
Utilizamos: **DISTINTO NÚMERO Y/O TIPO DE ARGUMENTOS.**

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

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

template <class T>
T maximo (T arr[], int size )
{ .....};

template <class T>
T maximo (int A, int size ) //??
{ .....};
```

```
int main()
{
    char c[5]={5, 6, 3, 2, 0};
    int a; float b;
    double m;
    a = maximo(5,10);
    b = maximo(4.5, 5.0, 9.0);
    m = maximo(c, 5);
}
```

22.2 Plantillas de Funciones

Especializando Funciones Genéricas

```
template <class T>
UnTipo maximo (T a, T b)
{
    if (a<b)
        return b;
    return a;
}

// Funcion maximo() especializada
char * maximo ( char* s1, char* s2 )
{
    if (strcmp(s1,s2)>0)
    {
        return s1;
    }
    return s2;
}
```

Se puede definir una función “normal” que prevalezca sobre la definición de la plantilla.

La plantilla hace lo correcto con tipos de datos tales como enteros, reales,...

No hace lo esperado con cadenas.

¿Por qué?

¿Funciona nuestra plantilla con la clase Alumno?

Le falta algo, ¿no? ¿El qué?

22.2 Plantillas de Funciones

Especialización de Funciones Genéricas

Reglas para encontrar la función sobrecargada correcta:

- Se busca coincidencia exacta de funciones. ¿Hay alguna función específica que coincida en nombre y en número, tipo y orden de los argumentos?
- Se busca alguna plantilla de función que coincida en nombre y argumentos.
- Las reglas utilizadas son las de la sobrecarga “normal”.
- Si no hay coincidencia se genera error.

22.3 Ejemplo de Plantillas de Funciones

Plantilla de Funciones en acción

Supongamos que nos piden que codifiquemos una función que devuelva el valor máximo de los elementos de un array de enteros.

```
int max_array (int arr[ ] ,int tam )
{
    int aux;
    aux = arr[0];
    for (int i =0; i<tam; i++)
    {
        if ( arr[i]>aux )
            aux = arr[i];
    }
    return aux;
}
```

Ahora nos dicen que necesitan otra función para arrays de “long int”.

Usamos la sobrecarga:

- Ellos hacen la llamada igual.
- Nosotros “copiar-pegar”

```
long int max_array (long int arr[ ]
                    ,int tam )
{
    long int aux;
    aux = arr[0];
    for (int i =0; i<tam; i++)
    {
        if ( arr[i]>aux )
            aux = arr[i];
    }
    return aux;
}
```

22.3 Ejemplo de Plantillas de Funciones

Plantilla de Funciones en acción

Sabemos que nos van pedir otra función pero para reales – *float*, porque van a querer notas con decimales.

Además nos damos cuenta de que se puede optimizar (empezamos en la posición 1 del array en lugar de en la 0) y no queremos ir tocando código en quince sitios diferentes.

```
long int max_array (long int arr[ ]
                    ,int tam )
{
    long int aux;
    aux = arr[0];
    for (int i =0; i<tam; i++)
    {
        if ( arr[i]>aux )
            aux = arr[i];
    }
    return aux;
}
```

```
int main()
{
    int arr_i[5]={1,7,3,5,2};
    float arr_f[4]={1.2,2.3,7.5,0.3};
    char arr_c[]={'a','x','z','b'};

    cout << "maximo int:" <<
        max_array(arr_i,5) << endl;
    cout << "maximo float:" <<
        max_array(arr_f,5) << endl;
    cout << "maximo char:" <<
        max_array(arr_c,4) << endl;
}
```

- 1.- Implementar `max_array` usando una función genérica.
- 2.- Hacer que sirva también para cadenas (`char*`)