

Algoritmos y Estructuras de Datos
2º de Ingeniería Informática, Curso 2008/2009

SEMINARIO DE C++

(orientado a programadores java,
pero no es parte del plan piloto, sólo para alumnos de AED)

Sesión 3

Contenidos:

1. Funciones y clases genéricas
 2. Excepciones
 3. Asertos
 4. El puntero `this`
 5. Redefinición de operadores
- Ejemplo

1. Funciones y clases genéricas¹

- **Genericidad (parametrización de tipo):** el significado de una función o de un tipos de datos está definido en base a unos parámetros de tipo que pueden variar.
 - **Ejemplo: funciones genéricas.** En lugar de: `OrdenaInt(int array[])`, `OrdenaCharP(char *array[])`, `OrdenaFloat(float array[])`, ..., tenemos `Ordena<T>(T array[])`.
 - **Ejemplo: clases genéricas.** En lugar de las clases `PilaInt`, `PilaChar`, `PilaFloat`, ..., tenemos una clase genérica `Pila<T>`.
- En C++, las funciones y clases genéricas se definen mediante **sentencias `template` (plantilla)**. La sintaxis es:

`template < parámetros genéricos > función o clase genérica`

- *parámetros genéricos:* Lista con uno o varios tipos genéricos (clases). Por ejemplo: `template <class T>...`, `template <class A, class B>...`
- *función o clase genérica:* Declaración normal de una función o una clase, pudiendo usar los tipos de la lista de parámetros genéricos.

- **Funciones genéricas** -----
 - **Definición.**

```
template < parámetros genéricos >
tipoDevuelto nombreFunción ( parámetros de la función )
{
    cuerpo de la función
}
```

- **Utilización.** Llamar directamente a la función. El compilador, según los tipos de los parámetros de entrada, instanciará la versión adecuada.
- **Ejemplo.** Función genérica `ordena(T array[], int tam)`, que ordena un array de elementos de cualquier tipo `T`.



```
#include <string>           // Contiene la clase string, alternativa
                          // a los (char *) de C

#include <iostream>
using namespace std;

template <class T>
void ordena (T array[], int tam)
{
    for (int i= 0; i<tam-1; i++)
        for (int j= i+1; j<tam; j++)
            if (array[j]<array[i]) {
                T tmp= array[i];
                array[i]= array[j];
                array[j]= tmp;
            }
}
```

¹ desde la versión 1.5 (octubre 2004) java incluye clases genéricas; ¿visto en primero?

```
template <class T>
void escribe (T array[], int tam)
{
    for (int i= 0; i<tam; i++)
        cout << array[i] << (i<tam-1? ", ": "\n");
}

#define numElems(ARRAY) sizeof(ARRAY)/sizeof(*ARRAY)

main ()
{
    int a1[]={65, 23, 12, 87};
    ordena (a1, numElems(a1));
    escribe(a1, numElems(a1));

    string a2[]= {"hola", "esto", "es", "una", "prueba"};
    ordena (a2, numElems(a2));
    escribe(a2, numElems(a2));
}
```

- **Clases genéricas** -----

- **Definición de la clase.**

```
template < parámetros genéricos >
class nombreClase
{
    definición de miembros de la clase
};
```

- **Implementación de los métodos.** Para cada método de la clase tenemos:

```
template < parámetros genéricos >
tipoDevuelto nombreclase<par>::nombreFunción ( parámetros )
{
    cuerpo de la función
}
```

- **Utilización.** La clase genérica se debe instanciar a un tipo concreto antes de usarla. **Instanciación:** *nombreclase*<*parámetros*>.

- **Ejemplo.** Clase genérica **pila<T>**, de las pilas de cualquier tipo **T**.



```
#include <iostream>
#define MAXIMO 100
using namespace std;

template <class T> // Declaración de la clase
class pila
{
private:
    int ultimo;
    T datos[MAXIMO];
public:
    pila () {ultimo= 0;} // Constructor
    void push (T v);
    void pop ();
    T tope();
};
```

```
// Implementación de los métodos
template <class T>
void pila<T>::push (T v)
{
    if (ultimo<MAXIMO)
        datos[ultimo++] = v;
}

template <class T>
void pila<T>::pop ()
{
    if (ultimo>0)
        ultimo--;
}

template <class T>
T pila<T>::tope ()
{
    return datos[ultimo-1];
}

// Programa principal
// Ejemplo de uso de la clase genérica
main ()
{
    pila<int> p1;           // p1 es una pila de enteros
    p1.push(4); p1.push(9);
    p1.pop();
    cout << p1.tope() << '\n';

    pila<char *> p2;      // p2 es una pila de cadenas
    p2.push("Ultimo"); p2.push("en"); p2.push("entrar");
    p2.push("primero"); p2.push("en"); p2.push("salir");
    for (int i= 0; i<6; i++, p2.pop())
        cout << p2.tope() << '\n';

    pila<float> *p3;      // p3 es una pila de float que
    p3= new pila<float>; // se crea dinámicamente
    p3->push(1.3); p3->push(2.19); p3->push(3.14);
    cout << p3->tope() << '\n';
    delete p3;
}
```

- **Probar:**
 1. Probar pilas de otros tipos: `pila<char>`, `pila<pila<char> >`, etc.
 2. Implementar una operación `igual`, que compare dos pilas y devuelva **true** si son iguales.
 3. Probar la función `igual`.
- Existe una "**pequeña pega**" relacionada con las clases genéricas y la compilación separada: el compilador sólo genera código cuando se instancia la clase genérica.

Problema: si separamos la definición de una clase genérica en fichero de cabecera (`pila.hpp`) y de implementación (`pila.cpp`), la compilación (`g++ -c pila.cpp`) no genera código objeto. Si otro módulo usa la clase genérica (`#include "pila.hpp"`), el compilador dirá que no encuentra la implementación de la clase.

Solución: para las clases genéricas, tanto la definición como la implementación de la clase deben ir en el mismo fichero, el **hpp**.

2. Excepciones²

- ¿Qué hacer si se produce un **error irreversible** dentro de un procedimiento? Por ejemplo, hacer una división por cero, aplicar `tope()` sobre una pila vacía, imposibilidad de reservar más memoria o de leer un fichero, etc.
 1. Mostrar un **mensaje de error** por pantalla, o por la salida de error estándar:
`cerr << "Error: la pila está vacía";`
 2. Devolver al procedimiento que hizo la llamada un **valor especial** que signifique "se ha producido un error".
 3. **Interrumpir la ejecución** del programa: `exit(1)`
 4. No hacer **nada** y continuar con la ejecución del programa como mejor se pueda.
 5. Provocar o lanzar una **excepción**.
- **Significado de las excepciones:**
 1. El procedimiento que **lanza la excepción** (lo llamamos *A*), interrumpe su ejecución en ese punto.
 2. El procedimiento que ha llamado al *A* (lo llamamos *B*), debe tener un código para **manejar excepciones**.
 3. Si el procedimiento *B* no maneja la excepción, la excepción pasa al que ha llamado al *B*, y así sucesivamente hasta que alguien trate la excepción (**propagación de la excepción**).
 4. Si la excepción se propaga hasta el `main()` y este no maneja la excepción, se interrumpe la ejecución del programa y se muestra un mensaje de error.
- Conceptualmente, una **excepción es un objeto**³.

```
class ErrorPilaVacía {};  
class DivisionPorCero {};
```

- **Provocar una excepción.** Sentencia: `throw` ⁴ `nombreClase();`

```
double divide (int a, int b)  
{  
    if (b==0)  
        throw DivisionPorCero();  
    return double(a)/double(b);  
}
```

- **Manejar una excepción.** Bloque:

```
try {  
    sentencias1  
}  
catch (nombreClase_1) {  
    sentencias2  
}  
catch (nombreClase_2) {  
    sentencias2  
}  
// no existe la sentencia "finally"
```

² muy parecido a java, ¿qué habéis visto?

³ cualquier objeto, no es necesario que sea un objeto de clase heredera de otra que sea una excepción, es decir, no necesario `extends nombre_excepcion`.

⁴ sin añadir `new`, es decir, no se usa `throw new nombreClase();`

- Funcionamiento del bloque `try-catch`:
 1. Ejecutar las sentencias de *sentencias1* normalmente.
 2. Si se produce una excepción del tipo *nombreClase*, ejecutar *sentencias2*. Estas sentencias contienen el código de manejo de excepciones.

```
try {
    double d1, d2, d3;
    d1= divide(3, 4);
    d2= divide(5, 0);
    d3= divide(1, 2);
}
catch (DivisionPorCero) {
    cerr << "División por cero\n";
};
```

- Al contrario que en java, **no se usa `throws nombre_excepcion`** para indicar que un método lanza excepciones del tipo *nombre_excepcion*.
- Manejar **excepciones de cualquier tipo**⁵:

```
try {
    sentencias1
}
catch (...) {
    sentencias2
}
```

- **Ejemplo.**



```
#include <iostream>
class DivisionPorCero {};
double divide (int a, int b)
{
    if (b==0)
        throw DivisionPorCero();
    return double(a)/double(b);
}
main()
{
    try {
        cout << divide(3, 4) << '\n';
        cout << divide(5, 0) << '\n';
        cout << divide(1, 2) << '\n';
    }
    catch (DivisionPorCero) {
        cerr << "División por cero\n";
    }
    cout << "Acaba\n";
}
```

- **Probar:**
 1. Ejecutar la división por cero fuera del bloque `try ... catch ...`
 2. Cambiar `catch (DivisionPorCero)` por `catch (...)`
 3. Ejecutar una sentencia `throw` dentro del `main()`

⁵ en java, se haría `catch` con la clase `exception`.

3. Asertos

- **Asertos: verificar una condición.** Si se cumple, seguir normalmente. Si no, se muestra la línea de código donde falló el programa y se interrumpe la ejecución.

```
assert ( condición booleana );
```

- La función `assert()` está definida en la librería `<assert.h>`
- Los asertos se pueden utilizar para incluir precondiciones y postcondiciones.

```
funcion (parametros)
{
    assert (precondición);
    ...
    // Cuerpo de la función
    ...
    assert (postcondición);
}
```

- **Ejemplo.** En el ejemplo de la página anterior sustituir la implementación de `divide()` por la siguiente. No olvidar el `#include <assert.h>`

```
double divide (int a, int b)
{
    assert(b!=0);
    return double(a)/double(b);
}
```



- Si se define la macro `NDEBUG`, se **desactiva** la comprobación de todos los asertos definidos. **Probar en el ejemplo anterior.**
- **Ojo:** Los asertos provocan la interrupción del programa.
- **Otra posibilidad:** comprobación de precondición y postcondición generando excepciones. Definirse uno mismo las funciones...

```
class FalloPrecondicion {};  
class FalloPostcondicion {};  
  
void precondicion (bool condicion)  
{  
#ifndef NDEBUG  
    if (!condicion)  
        throw FalloPrecondicion();  
#endif  
}  
  
void postcondicion (bool condicion)  
{  
#ifndef NDEBUG  
    if (!condicion)  
        throw FalloPostcondicion();  
#endif  
}
```

4. El puntero `this`⁶

- Dentro de cualquier método de una clase `T` existe una variable especial: `this`
- El tipo de `this` es un **puntero** a `T`. El valor es un puntero al objeto receptor del mensaje. En java, se trataría de una referencia (indirecta) al mismo objeto.

```
pila<int> p1, p2;
```

```
p1.push(5);      // Aquí dentro this apunta a p1  
p2.pop();       // Y aquí this apunta a p2
```

- **Ejemplo.**



```
#include <iostream>
```

```
class Simple {  
    char c;  
public:  
    Simple *asigna(char valor);  
    Simple *escribe();  
};
```

```
Simple *Simple::asigna(char valor)  
{  
    this->c= valor;    // Equivalente a: c= valor;  
    return this;  
}
```

```
Simple *Simple::escribe()  
{  
    cout << this->c;    // Equivalente a: cout << c;  
    return this;  
}
```

```
main (void)  
{  
    Simple *s= new Simple;  
    s->asigna('a')->escribe()->asigna('b')->escribe();  
    s->asigna('c')->escribe()->asigna('\n')->escribe();  
    delete s;  
}
```

- **Ojo:** en las sentencias

```
    this->c= valor;  
    cout << this->c;
```

el uso de `this` es completamente innecesario. `this->XXX` se puede sustituir siempre por `XXX`. Usar `this` en esos casos sólo hace el código menos legible, por lo que debe evitarse su uso.

⁶ muy parecido a la referencia indirecta `this` en java; ¿la conocéis?

5. Redefinición de operadores⁷

- Los **operadores** son las funciones básicas del lenguaje: +, -, *, =, ==, <<, >>, &&, ||, etc.
- En C++ es posible **redefinir los operadores** dentro de una clase.
- **Redefinición de operadores.** Definir e implementar funciones con el nombre: operator+, operator-, operator*, operator=, operator==, etc.
- Igual que una función normal, pero con esos nombres especiales.
- **Ejemplo.** Dentro de la definición de la clase `pila<T>` incluir un operador de comparación, ==. Redefinir los operadores << y >> para que signifiquen *apilar* y *tope*, respectivamente.

```
template <class T>
class pila
{
    ...
    bool operator==(pila<T> &otra);
    void operator<<(T valor);
    void operator>>(T &valor);
    ...
}
...
// Implementación
template <class T>
bool pila<T>::operator==(pila<T> &otra)
{
    if (ultimo != otra.ultimo)
        return false;
    for (int i= 0; i<ultimo; i++)
        if (datos[i] != otra.datos[i])
            return false;
    return true;
}

template <class T>
void pila<T>::operator<<(T valor)
{
    if (ultimo<MAXIMO)
        datos[++ultimo]= valor;
}
}
```

- **Uso.** Igual que los operadores predefinidos en el lenguaje.

```
pila<int> p1, p2;
p1 << 4;
p2 << 6;
....

if (p1==p2) { // Ojo: si no redefinimos el operador ==, esta
    ...      // comparación de aquí no funcionaría
}
```

⁷ no existe en java.

```
}
```

Ejemplo

- Vamos a definir un tipo genérico `pila<T>`, con uso de memoria dinámica, asertos, el puntero `this` y redefinición de operadores.

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

//Declaración de la clase generica pila<T>
template <class T>
class pila
{
    private:
        int ultimo;
        int maximo;
        T *datos;
    public:
        pila (int max= 10);           //Constructor, max es el tamaño
        ~pila ();                    //Destructor
        bool operator==(pila<T> &otra); //Comparación de igualdad
        void operator=(pila<T> &otra); //Asignación entre pilas
        void push (T v);
        pila<T> &operator<<(T valor); //Como push, devolviendo this
        void pop ();
        T tope();
        pila<T> &operator>>(T &valor); //Como tope+pop, devolv. this
        void escribe (char *nombre= NULL, ostream &co= cout);
};

//Operador de salida, para poder hacer: cout << pila;
template <class T>
ostream& operator<<(ostream& co, pila<T> &p)
{
    p.escribe(NULL, co);
    return co;
}

//Implementacion de los metodos de la clase pila<T>
template <class T>
pila<T>::pila (int max)
{
    assert(max>0); //Precondicion
    datos= new T[max];
    maximo= max;
    ultimo= 0;
    assert(datos!=NULL); //Postcondicion
}

template <class T>
pila<T>::~~pila ()
{
    assert(datos!=NULL); //Precondicion
    delete[] datos;
}

template <class T>
bool pila<T>::operator==(pila<T> &otra)
{
```

```
    if (ultimo != otra.ultimo)
        return false;
    for (int i= 0; i<ultimo; i++)
        if (datos[i] != otra.datos[i])
            return false;
    return true;
}
```

```
template <class T>
void pila<T>::operator=(pila<T> &otra)
{
    delete[] datos;
    ultimo= otra.ultimo;
    maximo= otra.maximo;
    datos= new T[maximo];
    for (int i= 0; i<ultimo; i++)
        datos[i]= otra.datos[i];
    assert(*this==otra);    //Postcondicion
}
```

```
template <class T>
void pila<T>::push (T v)
{
    if (ultimo<maximo)
        datos[ultimo++]= v;
    else {
        // Reservar un nuevo array para el doble de valores
        maximo= maximo*2;
        T *nuevo= new T[maximo];
        assert(nuevo!=NULL);
        for (int i= 0; i<ultimo; i++)
            nuevo[i]= datos[i];
        delete[] datos;
        datos= nuevo;
        datos[ultimo++]= v;
    }
}
```

```
template <class T>
pila<T> &pila<T>::operator<<(T valor)
{
    push(valor);
    return *this;
}
```

```
template <class T>
void pila<T>::pop ()
{
    assert(ultimo>0);    //Precondicion
    ultimo--;
}
```

```
template <class T>
T pila<T>::tope ()
{
    assert(ultimo>0);    //Precondicion
    return datos[ultimo-1];
}
```

```
template <class T>
pila<T> &pila<T>::operator>>(T &valor)
{
    valor= tope();
    pop();
}
```

```
    return *this;
}

template <class T>
void pila<T>::escribe (char *nombre, ostream &co)
{
    if (nombre)
        co << nombre << "= ";
    co << "[";
    for (int i= 0; i<ultimo; i++)
        co << datos[i] << (i<ultimo-1?"", ":\n");
}

//Funcion principal
int main (int narg, char *sarg[])
{
    pila<int> *p1= new pila<int>(5);
    *p1 << 4 << 9 << 6 << 14 << 65 << 11;
    p1->escribe("p1");
    pila<int> *p2= new pila<int>(10);
    *p2= *p1;
    *p2 << 99 << 76;
    cout << "p2= " << *p2;
    cout << "¿Iguales p1 y p2? " << (*p1 == *p2) << "\n";
    int tmp1, tmp2;
    *p2 >> tmp1 >> tmp2;
    cout << *p2;
    cout << "¿Iguales p1 y p2? " << (*p1 == *p2) << "\n";
    delete p1;
    delete p2;

    pila<char *> *p3= new pila<char *>;
    for (int i= 0; i<narg; i++)
        *p3 << sarg[i];
    cout << *p3;
    delete p3;

    pila< pila<double> > *p4= new pila< pila<double> >;
    p4->push(*(new pila<double><<2<<4);
    p4->push(*(new pila<double>(10))<<8<<9<<10);
    p4->push(*(new pila<double>(1))<<1<<2<<3<<4);
    cout << *p4;
    delete p4;

    pila<char> *p5;
    p5= new pila<char>;
    char aux1, aux2;
    try {
        *p5 << 6;
        *p5 >> aux1 >> aux2;
        *p5 << 7;
        delete p5;
    }
    catch (...) {
        cerr << "Error, pila vacia.\n";
        delete p5;
    }
    p1->escribe();
}
```