# GRÀFICS I VISUALITZACIÓ DE DADES

GiVD – curso 2013-2014

## Breve tutorial de C++ para programadores de Java

## 1. Organización de los archivos .cpp i .h

El código se encuentra en archivos .cpp, declaraciones y demás en archivos de cabecera .h. En las cabeceras declararemos variables, funciones, definiciones de clases... En los archivos .cpp añadiremos el cuerpo de las clases y de las funciones.

Cuando queramos acceder a dichas clases/funciones desde el código (.cpp) deberemos hacer un include de sus cabeceras.

La clausula #ifndef del preprocesador en una cabecera nos permite evitar las inclusiones cíclicas:

```
#ifndef _INCL_GUARD_H
#define _INCL_GUARD_H
...
#endif
```

## 2. Nuevos recursos

2.1. Los **namespace** sirven para organizar el código. Permiten declarar y utilizar código en distintos contextos.

```
namespace operaciones {
      int suma (int a, int b) {
            return a + b;
      }
}
```

Luego, para utilizarlo deberemos hacer:

```
operaciones::suma(3, 4);
```

Aunque podemos hacer un "acceso directo" de ese namespace usando "using" (lo que equivaldría al import de java):

```
using namespace operaciones;
suma (3, 4);
```

2.2. La **librería standard** (std) contiene nuevos recursos:

- Flujos de datos (ver más abajo):

```
std::cout << "hola mundo" << std::endl;
```

- Clase String para tratar con cadenas de texto (en string, con propiedades y métodos tales como length, substr, find):

```
string foo = "fighters";
```

- Clase Vector para arrays dinámicos (en vector):

```
vector<int> myVector;
myVector.push_back(9);
myVector.push_back(3);
myVector.push_back(1);
for(unsigned int i=0; i<myVector.size(); ++i)
std::cout << myVector[i] << ' ';
std::cout << std::endl;
```

2.3. Los **streams** nos sirven para manipular flujos de datos:

- cout y cin son los flujos de datos para escribir y leer por consola (disponible en iostream)

```
int nManzanas;
cout << "Cuantas manzanas tienes? ";
cin >> nManzanas;
cout << "Tienes " << nManzanas << " manzanas";
```

- En fstream encontramos estructuras de datos para manipular ficheros.

  Para escribir en ficheros:

```
#include <iostream>
#include <fstream>
using namespace std;
int main () {
  ofstream myfile ("example.txt");
  if (myfile.is_open()) {
    myfile << "Linea 1.\n";
    myfile << "Linea 2.\n";
    myfile.close();
  }
  return 0;
}
```

Para leer de ficheros:

```
ifstream myfile ("test3.py");
char* buffer = new char[255];
while (!myfile.eof()) {
  myfile >> buffer;
  cout << buffer;
}
myfile.close();
return 0;
```

# 3. Definición de clases

3.1. Organización del código:

| Archivo .h | Archivo .cpp |
|---|---|

```
class CRectangle {
  private:
    int x, y;
  public:
    void set_values (int,int);
    int area ();
};
```

```
void CRectangle::set_values (int a, int
b) {
  x = a;
  y = b;
}
int CRectangle::area () {
      return (x*y);
}
```

3.2. Creación de un objeto utilizando el constructor por defecto:

```
CRectangle rect;
rect.set_values(3, 4);
int area = rect.area();
```

3.3. Creación de un objeto con new:

```
CRectable* rect = new CRectangle;
rect->set_values(3, 4);
int area = rect->area ();
delete rect;
```

Cabe destacar que cuando la variable es una referencia utilizamos '->'para acceder a sus métodos y propiedades, cuando no se utiliza '.' .

3.4. Constructores y destructores (son llamados al retirar el objeto de memoria):

| Archivo .h | Archivo .cpp |
|---|---|
| `class CRectangle {`<br>`  private:`<br>`  int *width, *height;`<br>`  public:`<br>`  CRectangle ();`<br>`  CRectangle (int,int);`<br>`  ~CRectangle ();`<br>`  int area;`<br>`};` | `CRectangle::CRectangle () {`<br>`  width = new int;`<br>`  height = new int;`<br>`  *width = 5;`<br>`  *height = 5;`<br>`}`<br><br>`CRectangle::CRectangle (int a, int b) {`<br>`  width = new int;`<br>`  height = new int;`<br>`  *width = a;`<br>`  *height = b;`<br>`}`<br><br>`CRectangle::~CRectangle () {`<br>`  delete width;`<br>`  delete height;`<br>`}`<br><br>`int CRectangle:: () {`<br>`  return (*width * *height);`<br>`}` |

3.5. Tipos de **herencia**:

- Pública: Tal y como se entiende en Java. Los miembros públicos de la clase

base pasan a ser públicos en la clase derivada, los protegidos siguen siendo protegidos.

- Protegida: Miembros public y protected pasan a ser protected en la clase derivada.
- Privada: Miembros public y protected pasan a ser private en la clase derivada.

| Archivo .h | Archivo .cpp |
|---|---|
| class CPolygon {<br>  protected:<br>  int width, height;<br>  public:<br>  void set_values (int , int);<br>};<br>class CRectangle: public CPolygon {<br>  public:<br>  int area ();<br>};<br>class CTriangle: public CPolygon {<br>  public:<br>  int area ();<br>}; | void CPolygon::set_values (int a, int b)<br>{<br>  width=a;<br>  height=b;<br>}<br><br>int CRectangle::area () {<br>  return (width * height);<br>}<br>int CTriangle::area () {<br>  return (width * height / 2);<br>} |

Únicamente pueden ser sobreescritos los métodos virtuales:

| Archivo .h | Archivo .cpp |
|---|---|
| class CPolygon {<br>  protected:<br>  int width, height;<br>  public:<br>  void set_values (int , int );<br>  virtual int area ();<br>};<br>class CRectangle: public CPolygon {<br>  public:<br>  int area ();<br>};<br>class CTriangle: public CPolygon {<br>  public:<br>  int area ();<br>}; | void CPolygon::area (){<br>  return 0;<br>} |

3.6. Las clases **abstractas** se definen como classes que no implementan directamente un método y que fuerzan a sus hijos a implementarlos. En C++ se definen igualando un método virtual a 0. A partir de entonces, no se podrán crear objetos de esa clase sino de sus derivadas.

```
class CPolygon {
  protected:
    int width, height;
  public:
```

```
    void set_values (int a, int b);
    virtual int area () =0;
};
```

3.7. Existe también la **herencia múltiple**, de forma directa:

```
class CRectangle: public CPolygon, public COutput {
  public:
    int area ();
};
```

3.8. En C++ la clausula **this** se refiere al propio objeto y es un puntero:

```
void CRectangle::set_values (int a, int b) {
  this->x = a;
  this->y = b;
}
```

## 4. Punteros: son similares a C, aunque con ciertas extensiones

&variable indica la dirección de memoria de una variable.

```
int x=2;
int *q = &x;            // q es un puntero a int, apunta a la dirección de x
cout << q << endl;      // imprime la dirección de x
cout << *q << endl;     // imprime el valor de x
```

Para utilizarlos en llamadas a funciones:

```
// por valor
void suma1 (int valor) {
      valor++;
}
// modificamos localmente la direccion del puntero
void suma2 (int* valor) {
      valor++;
}
// modificamos el valor al que apunta el puntero
void suma3 (int* valor) {
      (*valor)++;
}
// pasamos el valor por referencia. Se modifica el valor directamente
void suma4 (int& valor) {
      valor++;
}
// al ser const no nos permite cambiar su valor aunque sea referencia
void suma5 (const int& valor) {
      //valor++;
}
int main () {
      int v = 5;
      suma1(v);
      cout << v << endl;      // 5
      suma2(&v);
      cout << v << endl;      // 5
      suma3(&v);
      cout << v << endl;      // 6
      suma4(v);
      cout << v << endl;      // 7
      suma5(v);
```

```
        cout << v << endl;      // 7
}
```

## 5. Reservar memoria con new y liberarla con delete

Para valores únicos:

```
int* i = new int;
*i = 5;
delete i;
```

Para arrays:

```
char* pvalue  = NULL;
int size = 20;
pvalue  = new char[size];
pvalue[3] = 'hola';
delete [] pvalue;
```

De arrays de objetos:

```
CRectangle* rectangleArray = new CRectangle[4];
delete [] rectangleArray;
```