Práctica 0

Introducción al entorno de programación en C++

Grado en Ingeniería Informática

Estructuras de Datos II

2020/21

Departamento de Tecnologías de la Información
Universidad de Huelva

Introducción al entorno de programación en C++

Objetivos

- Revisar aspectos generales de la programación en C++
- Aclarar temas particulares de C++ necesarios para el desarrollo de las prácticas, como la Genericidad y el manejo de Excepciones

Contenidos

1. Programación modular en C++: Proyectos
2. Genericidad: Templates
3. Miscelánea de C++
4. Excepciones

Bibliografía

"Data Structures and Algorithms in C++". Michael T. Goodrich, M.T; Tamassia, R; Mount, D.M. John Wiley & Sons, Inc. 2004. Capítulos 1 y 2.

Introducción al entorno de programación en C++

1. Programación modular: Proyectos

Normalmente de igual nombre que el TAD

- Para programar con TAD de forma modular es necesario crear un *proyecto*.
- Por ejemplo, para crear el TAD *mi_tipo*, se podría crear un proyecto al que se agregase el fichero *mi_tipo.cpp (*con la implementación de los métodos) y el cual incluyese *mi_tipo.h* (interfaz del TAD)
- Las directivas *#ifndef* y *#define* insertadas en ficheros de cabecera .h, evitan que se produzca duplicidad en la inclusión de éstos. Por ejemplo, al principio del fichero *mi_tipo.h*:

```
#ifndef   _mi_tipo_h_
#define  _mi_tipo_h_
…
#endif
```

- Una vez compilado y generado el código objeto del TAD, los ficheros que se deben facilitar para que un programa usuario pueda utilizar el tipo *mi_tipo* son:

  - *mi_tipo.h*: interfaz del TAD
  - *mi_tipo.o*: código objeto del TAD

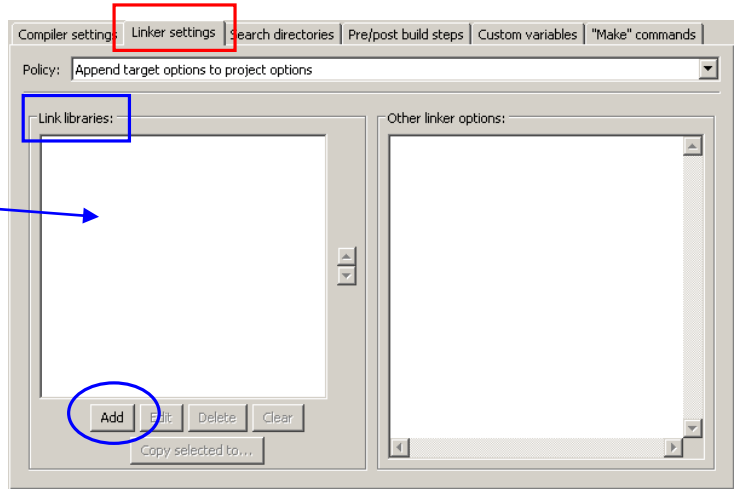
Introducción al entorno de programación en C++

- El **usuario** del TAD mi_tipo debe añadir en su proyecto:
  - **1.** el fichero *mi_tipo.h* junto con el resto de archivos del proyecto
    - **1.1** En **CodeBlocks**, la forma de hacerlo es*:*
      - *Proyecto → Add Files*
  - **2.** el fichero del código objeto *mi_tipo.o*.
    - **2.2** En **CodeBlocks**, la forma de hacerlo es*:*
      - *Proyecto → Build Options*
      - Pestaña "Linker Settings"*:* En "*Link Libraries"* añadir el fichero objeto *mi_tipo.o*

Lugar donde añadir: **mi_tipo.o**

Consejo:

"Keep as a Relative Path"?: **NO**



CodeBlocks

Introducción al entorno de programación en C++

2. Genericidad: Templates

- La genericidad en los TAD consiste en proporcionar una declaración que pueda ser utilizada para distintos tipos concretos
- La forma mas adecuada de conseguir esto en C++ es mediante el uso de los **Templates** (plantillas)
- Por ejemplo, la definición del tipo genérico *mi_tipo* sería:

mi_tipo.h

```
template <typename E> class mi_tipo
    {
     private:
       E nelementos;
       int limite;
     public:
       mi_tipo (E nelem);
       …
    };
```

mi_tipo.cpp

Método Constructor.
Para el resto de métodos aquí va: **void** / **tipo**

```
#include "mi_tipo.h"
template <typename E> mi_tipo<E>::mi_tipo (E nelem)
{
  nelementos = nelem;  limite=0;
}
…

// INSTANCIAS //
template class mi_tipo<int>;
template class mi_tipo<char>;
```

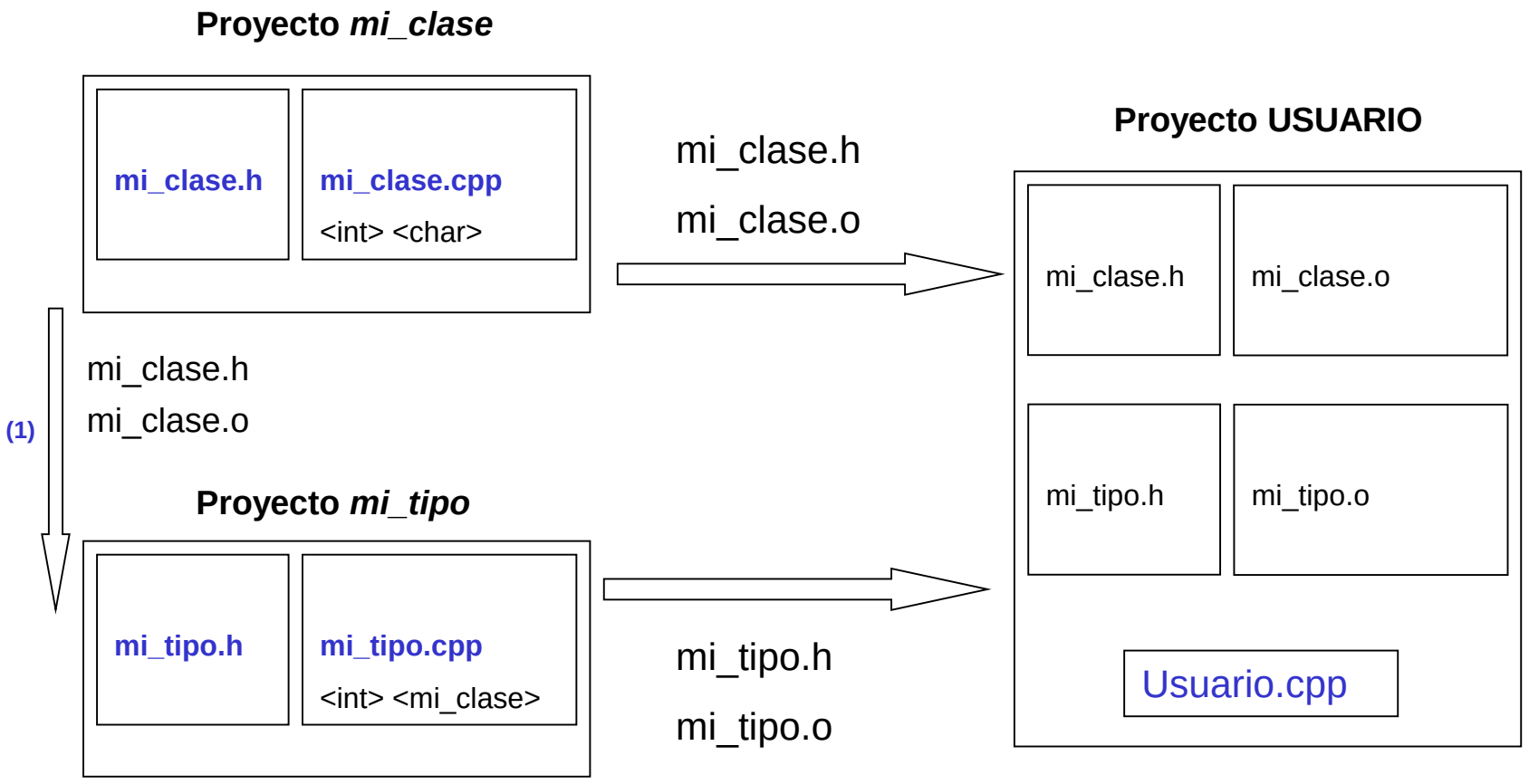
template <**typename E**> void/tipo mi_tipo <E>:: metodo_x (p1 tipodato)

Introducción al entorno de programación en C++

- Para poder exportar la clase genérica *mi_tipo,* hay que instanciarla a los tipos concretos que se deseen.
  - En el ejemplo la clase *mi_tipo* se ha instanciado para los **enteros** y los **caracteres**
  - Esto significa que un programa usuario puede definir y utilizar variables del tipo *mi_tipo*<int> y *mi_tipo*<char>
  - Si el usuario necesitase, por ejemplo, utilizar el tipo *mi_tipo*<bool>, habría que definir la instancia correspondiente en el fichero *mi_tipo*.cpp, compilar de nuevo *mi_tipo*.cpp y exportar *mi_tipo*.o, para que el usuario lo incluyese de nuevo en su proyecto

- Si la clase genérica *mi_tipo* se instancia con otra clase *mi_clase*, en el fichero *mi_tipo.cpp*, (junto con el resto de instanciaciones) hay que hacer: ***#include "mi_clase.h"***
  - En este caso podríamos definir tipos de la forma: *mi_tipo*< *mi_clase*>

Introducción al entorno de programación en C++



(1) Si no se incluye **mi_clase.o** en el proyecto **mi_tipo** → da un aviso de error de Link

Introducción al entorno de programación en C++

3. Miscelánea de C++

- Cuando se **declaran varios punteros** en la misma línea, el operador * va ligado al nombre de variable, y no al nombre de tipo:

  int* x, y, z; // *equivale a* ***int****** x;* ***int*** *y;* ***int*** *z;*

- La sintaxis de la **instanciación de una clase genérica** usando otra clase genérica debe ser así (por el problema de coincidencia sintáctica con el operador >>) :

  Tipo1< Tipo2<Tipo> > // *y no:* Tipo1<Tipo2<Tipo>>

- El **paso de parámetros y valores de retorno** de funciones, cuando sea posible, lo realizaremos mediante *referencias constantes.*
  - También, cuando sea el caso, haremos constar que los métodos no modificarán las variables miembro de la clase:

    const Objeto& método(**const** tipo1& arg1, **const** tipo2& arg2) **const**;

Introducción al entorno de programación en C++

- Si se crea un objeto en **memoria dinámica** con **new**, se debería en algún momento liberar con **delete**

- Cada clase que cree dinámicamente sus propios objetos con **new**, debería también:
  - definir un *destructor* para liberar la memoria de esos objetos
  - definir un *constructor de copia* para crear el almacenamiento de sus miembros y copiarles el contenido
  - definir un *operador de asignación* para liberar almacenamiento del objeto receptor, reservar almacenamiento nuevo, y copiar el contenido:

```
class  C {   T1 v1;  T2* v2;  };
C   c1, c2;
   ...
c2  =  c1;     //  sin un operador de asignación, se copiaría el puntero v2
               //  pero no a lo que se apunta (T2)
```

Introducción al entorno de programación en C++

- Para inicializar las variables miembro de una clase en un constructor, cuando se trate de variables de tipos no básicos, habrá que hacer uso de las ***listas de inicialización***, ubicadas entre la lista de argumentos del constructor y su cuerpo
- La sintaxis es "**:**" seguido de una lista separada por comas de la forma *miembro(valor_inicial)*:

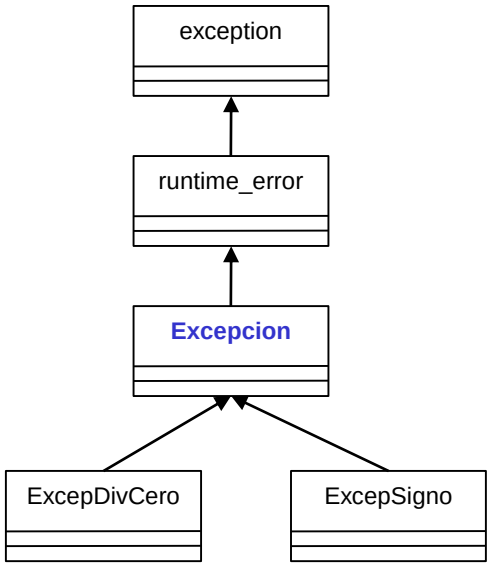
```
class  C  {
    private:
            int x;
            T   y, z;
    public:
            C(int i,  const T&  t1,  const T&  t2);
};

C::C(int i, const T&  t1,  const T&  t2) :  y(t1),  z(t2)  {
    x = i;
};
```

- El orden en el que se inicializan las variables miembro no es el de la lista de inicialización, sino el orden de declaración en la clase. Esta lista se ejecuta antes de las sentencias del cuerpo del constructor

Introducción al entorno de programación en C++

4. Excepciones

- Vamos a declarar muchas excepciones a lo largo del curso
- Para estructurarlas jerárquicamente, definiremos una nueva clase ***Excepcion*** de la que derivarán el resto de excepciones:
  - ***Excepcion*** deriva a su vez de la clase ***runtime_error*** de la librería ***<stdexcept>***




```
class Excepcion : public runtime_error
{
        public:
                Excepcion(const string& err) : runtime_error(err) { }
                string Mensaje() const { return what(); }
};
```

Introducción al entorno de programación en C++

Ejemplo

- Supongamos una función para el cálculo del porcentaje de dos operandos, con las excepciones de que el divisor sea nulo y que tengan distinto signo:

```
double  Porcentaje (double a, double b)  throw (ExcepDivCero, ExcepSigno)  {
    if (b == 0)
        throw ExcepDivCero();
    if (a*b < 0)
        throw ExcepSigno();
    return (a / b) * 100;
};
```

```
...
try {
    cout << "Porcentaje = " << Porcentaje(a,b);
}
catch (const ExcepDivCero& e)  {
    cout << e.Mensaje();
}
catch (const ExcepSigno& e)  {
    cout << e.Mensaje();
}
...
```

Cuando declaremos un procedimiento o función, se deben especificar en su cabecera las excepciones que puede lanzar su código

Introducción al entorno de programación en C++

```
class Excepcion : public runtime_error
{
        public:
                Excepcion(const string& err) : runtime_error(err) { }
                string Mensaje() const { return what(); }
};
```

```
class ExcepDivCero:  public Excepcion
{
     public :
          ExcepDivCero():  Excepcion("Divisor no puede ser 0")  { };
};
```

```
class ExcepSigno:  public Excepcion
{
     public :
          ExcepSigno():  Excepcion("Operandos con distinto signo")  { };
};
```