Universidad LOYOLA

Departamento de Métodos Cuantitativos

Tema 5. Programación Modular

Alfonso Carlos Martínez Estudillo (acme@uloyola.es)

Fundamentos de Informática I

1º de Grado en Ingeniería Informática y Tecnologías Virtuales

Curso 2021-2022

Índice de contenidos

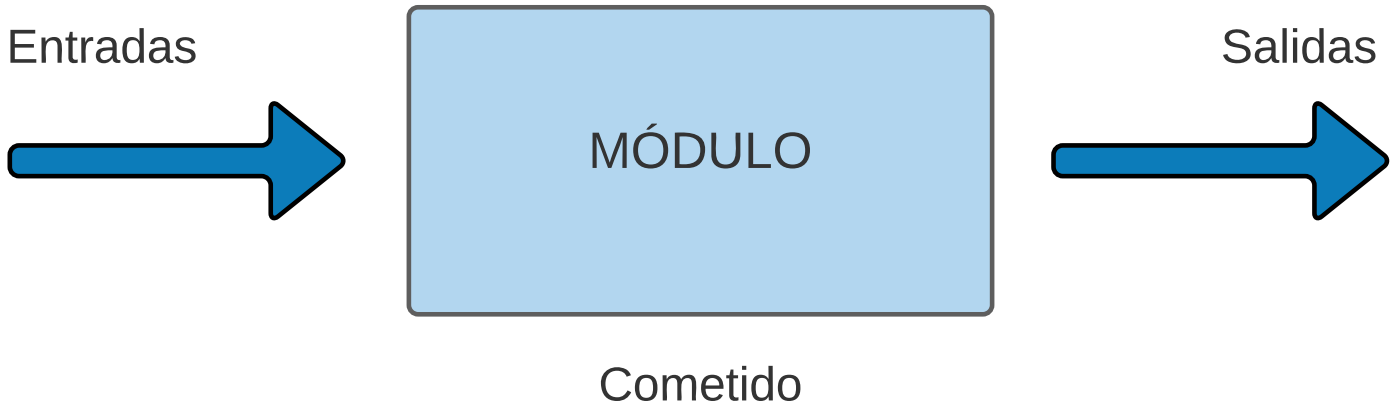
Índice de contenidos

Programación modular

- Forma de programación para descomponer de forma correcta un programa en módulos más sencillos.
- Consiste en dividir el programa en **módulos**: partes perfectamente diferenciadas, que pueden ser analizadas, programadas y optimizadas de forma independiente.
- La idea es poder cambiar la funcionalidad o código de un trozo de programa sin que el resto se vea afectado.
- Evita escribir código repetido y su reutilización en posteriores códigos.

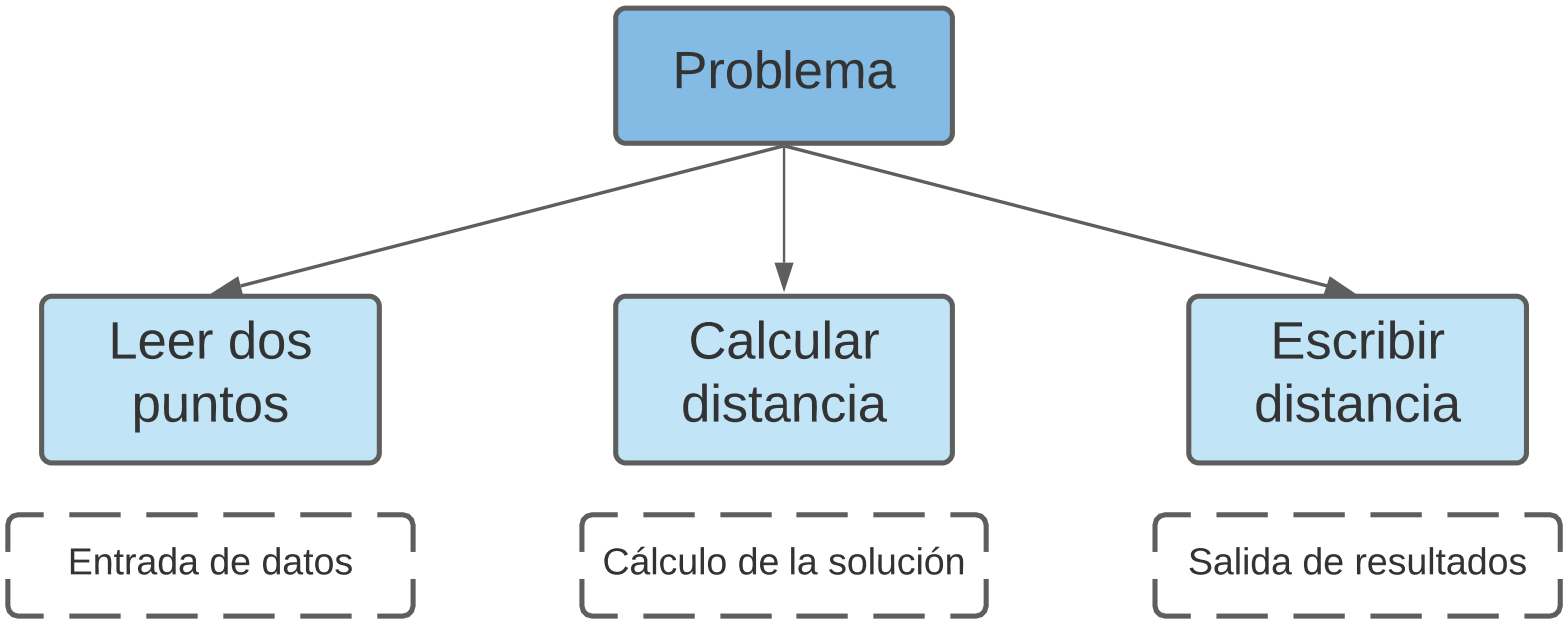
Módulos

- **Módulo**: debe actuar como una caja negra que ante unos valores de entrada suministre unos valores de salida que sean exclusivamente función de dicha entrada.
  - Una o varias instrucciones contiguas físicamente y relacionadas.
  - Se referencian por un nombre.
  - Se pueden invocar desde cualquier punto de un programa.



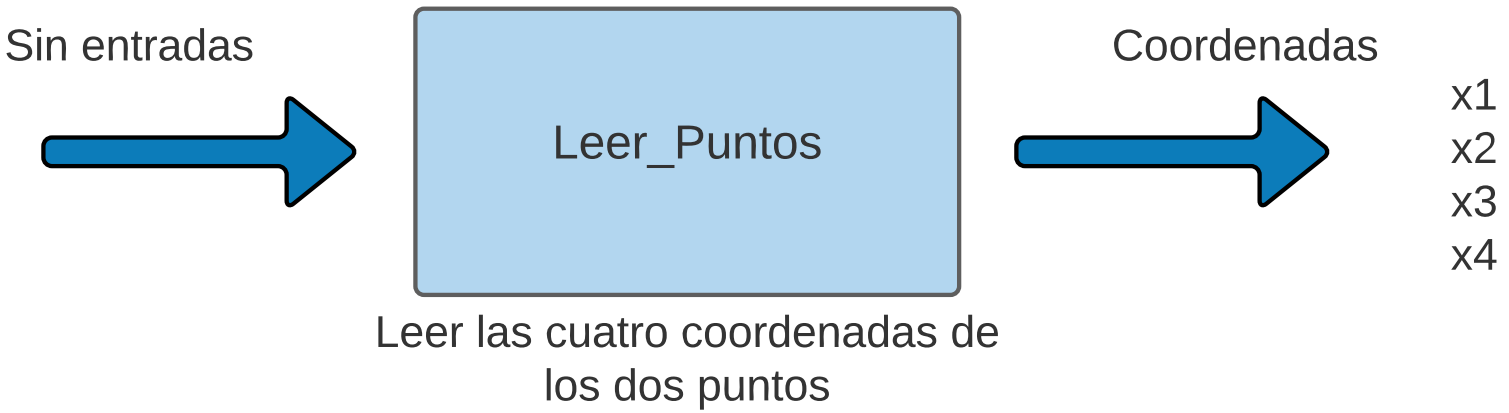
Ejemplo de modularización

- Determinar la distancia euclídea entre dos puntos.
- Podemos descomponer el programa en tres partes:
  - Leer los puntos.
  - Calcular la distancia.
  - Mostrar la distancia.



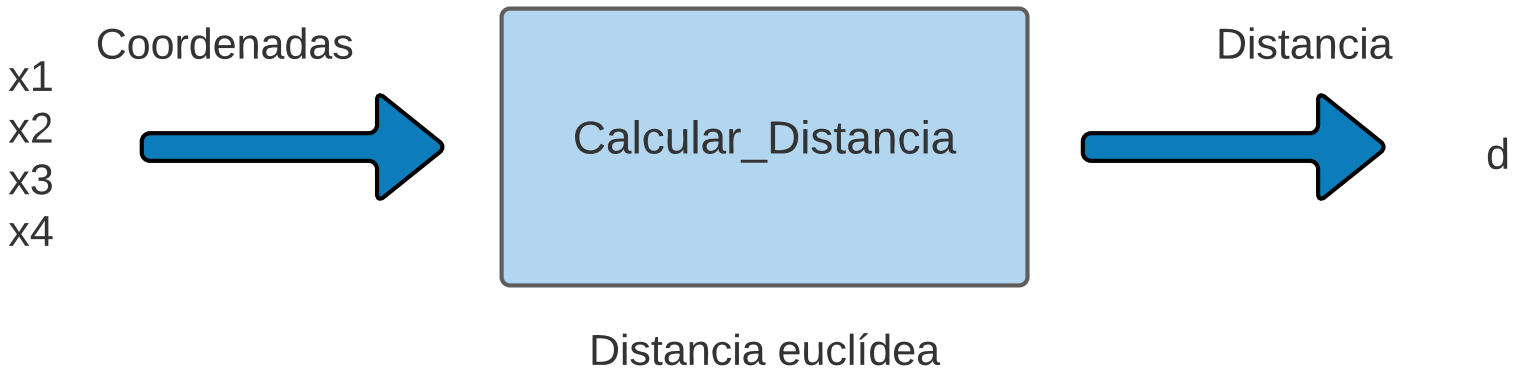
Ejemplo de modularización

- **Identificador:** Leer_Puntos.
- **Entrada:** Ninguna.
- **Salida:** Las cuatro coordenadas de tipo real.
- **Cometido:** Pedir al usuario los cuatro valores correspondientes a las coordenadas x1, y1, x2 e y2.



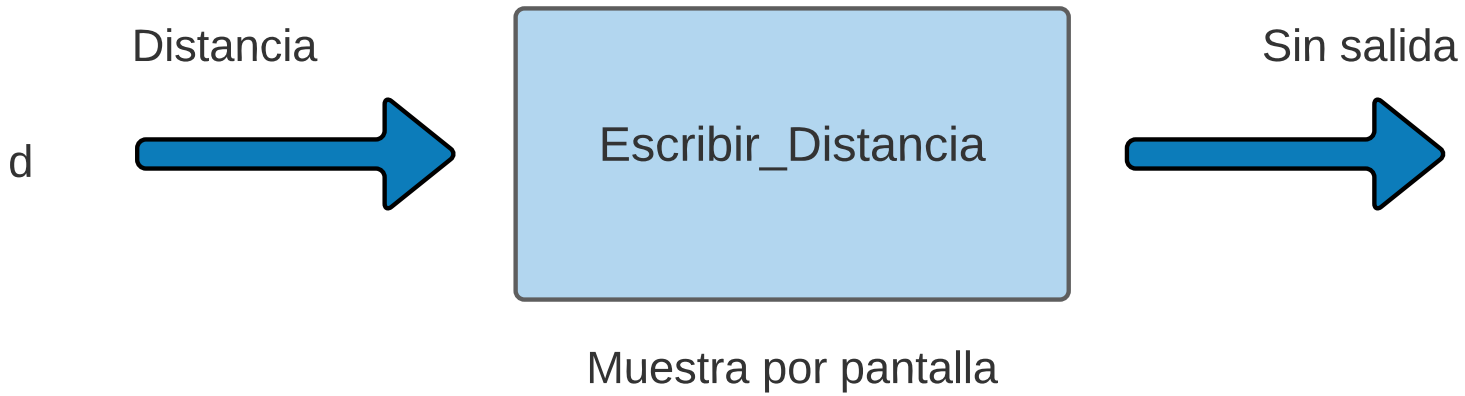
Ejemplo de modularización

- **Identificador:** Calcular_Distancia.
- **Entrada:** Cuatro datos de tipo real.
- **Salida:** Un dato de tipo real, correspondiente a la distancia.
- **Cometido:** Dadas las coordenadas x1, y1, x2 e y2, este módulo calcula la distancia euclídea mediante la fórmula $d(x1, y1, x2, y2) = \sqrt{(x1 - x2)^2 + (y1 - y2)^2}$.



Ejemplo de modularización

- **Identificador:** Escribir_Distancia.
- **Entrada:** un dato de tipo real.
- **Salida:** Ninguna.
- **Cometido:** Dadas la distancia *d*, la escribe por pantalla.



Requerimientos

- Establecimiento de un **organigrama modular**:
  - Conjunto de bloques, donde cada bloque corresponde a un módulo y relación entre módulo principal y los secundarios.
- Descripción del **módulo principal**:
  - Debe ser claro y conciso, reflejando los puntos fundamentales del programa.
- Descripción de los **módulos básicos** o secundarios:
  - Resuelven partes bien definidas; subdivisión en otros módulos.

Ventajas

- Escritura y depuración más sencilla.
- Facilita el mantenimiento, las modificaciones y las ampliaciones de programas.
- Facilita el trabajo en equipo.
- Posibilita el uso repetitivo de rutinas en el mismo o diferentes programas.
- Disminuye el coste de construcción de programas.

Índice de contenidos

Modularización en C++

- Aunque C++ es un lenguaje de programación orientado a objetos, éste puede ser utilizado como lenguaje de programación estructurado.
  - Uso de módulos para escribir los programas.
- Los módulos en C++ son subrutinas pequeñas denominadas funciones.
  - Funciones conocidas: pow(), sqrt(), ...
- Como hemos comentado anteriormente, cada función es una tarea independiente.

Estructura de una función

```
<tipo devuelto>    <nombre función>   (<parámetros formales>)
{
      [<constantes locales>]
      [<variables locales>]
      [<Sentencias>]
       return <expresión>;
}
```

Estructura de una función

```
float calcularDistancia (int x1, int y1, int x2, int y2)

{

    float cX, cY;
    float distancia

    cX = pow(x1-x2,2);
    cY = pow(y1-y2,2);
    distancia = sqrt(cX+cY);


    return distancia;

}
```

Estructura de una función

```
1 float calcularDistancia(int x1, int y1, int x2, int y2)
2 {
3   float cX, cY;
4   float distancia;
5   cX = pow(x1-x2,2);
6   cY = pow(y1-y2,2);
7   distancia = sqrt(cX+cY);
8   return distancia;
9 }
```

Estructura de una función

- **Nombre de la función:**
  - Identificador del módulo.
  - Necesario para invocarla dentro del programa.
  - Empieza con una letra o un subrayado y puede contener letras, números o subrayados.
  - Es sensible a mayúsculas y minúsculas.
- **Tipo de la función o tipo devuelto:**
  - Se especifica antes del nombre de la función.
  - Si la función no devuelve nada, será de tipo **void**.

Estructura de una función

- **Resultado de la función:**
  - Si la función devuelve algo, es necesario que contenga **return expresión**.
  - **expresión** debe ser del mismo tipo que devuelve la función.
  - **return** sólo devuelve un valor.
  - Puede haber más de una sentencia **return**.
  - Tan pronto como el programa encuentra un **return expresión**, la función termina y devuelve **expresión** al modulo llamador.
  - Si no hay una sentencia **return**, la ejecución continúa hasta la llave del cuerpo.
  - Si la función es de tipo **void** podemos omitir el **return**.

```
1 int minimo(int num1, int num2){
2   if(num1 < num2)
3     return num1;
4   else
5     return num2;
6 }
```

Estructura de una función

- **Llamada a una función:**
  - Las funciones para poder ser ejecutadas, han de ser llamadas.
  - La llamada a una función debe estar asignada a una variable o formar parte de otra expresión asignada a una variable, en caso contrario el valor se pierde.

```
variable = nombreFuncion(parametros actuales);
distancia = calcularDistancia(2,3,5,7);
valor = 7 - minimo(a,b);
cout << "El minimo entre 2 y 3 es: " << minimo(2,3) <<
    endl;
```

  - Si la función es de tipo **void**, su llamada no se asigna a una variable.

```
mostrarVector(v);
```

Estructura de una función

```
1  int main(){
2    ...
3    variable = 7 - minimo(a,b);
4    mostrarMinimo(variable);
5    ...
6  }
7
8  int minimo(int num1, int num2){
9    if(num1<num2)
10     return num1;
11   else
12     return num2;
13 }
14
15 void mostrarMinimo(int numero){
16   cout << "El valor minimo es: " << numero << endl;
17 }
```

Estructura de una función

- **Parámetros formales:**
  - Los identificadores definidos en la declaración de un módulo (cabecera de la función).
- **Parámetros reales o actuales:**
  - Las expresiones pasadas como argumentos en la llamada a un módulo.
- **Observaciones:**
  - Debe haber el mismo número de parámetros formales que de parámetros actuales.
  - Cada parámetro formal y su correspondiente parámetro actual deben ser del mismo tipo.
  - La correspondencia se establece por orden de aparición, uno a uno y de izquierda a derecha.

Estructura de una función

```
1  int  factorial (int num){ // parametro formal: num
2    int i, aux;
3    aux=1;
4    for(i=1; i<=num; i++){
5      aux*=i;
6    }
7    return aux;
8  }
9
10 float media(float num1, float num2){
11   return (num1+num2)/2;
12 }
13
14 int main(){
15   int x;
16   float suma;
17   cout << "Introduzca numero: ";
18   cin >> x;
19   cout << "El factorial es: " << factorial(x) << endl; // param actual: x
20   suma = media(factorial(x), factorial (x+1));
21   return 0;
22 }
```

Errores comunes

- Cortar la ejecución antes de que termine el bucle.

```
int  factorial (int num){
  int i, aux=1;
  for(i=1; i<=num; i++){
    aux*=aux;
    return aux;
  }
}
```

- No ejecutar una sentencia nunca.

```
int f(int n){
  return 2*n;
  cout << "Doble" << endl;
}
```

- No existir un **return** en algún camino.

```
bool par{int n){
  if(n%2==0)
    return true;
}
```

La función main()

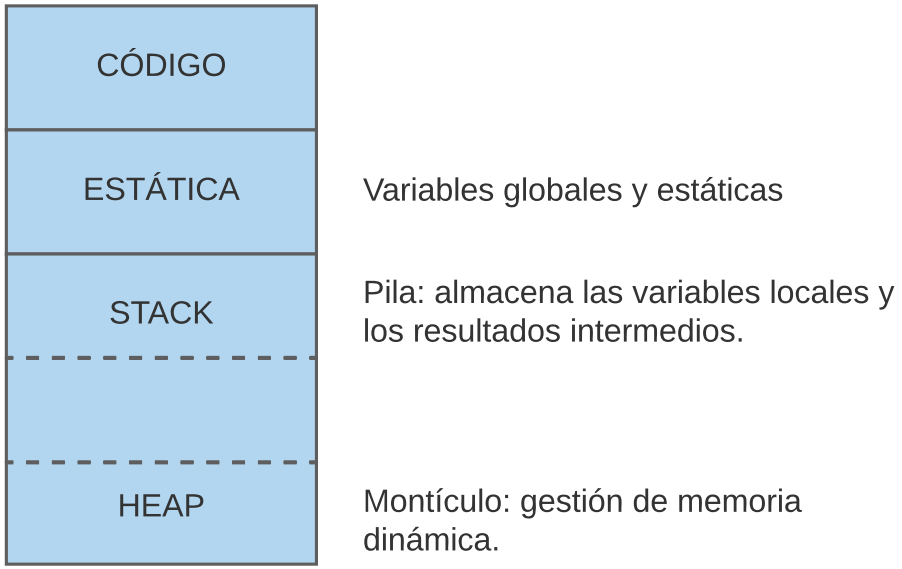
- Es una función más que debe aparecer en todo programa escrito en C++.
  - Módulo principal de un programa en C++.
- Devuelve un entero al Sistema Operativo.
  - Si el programa va bien, devuelve 0.
  - Si el programa falla, devuelve un entero distinto de cero.

```
1 int main(){
2   ...
3   return 0;
4 }
```

Índice de contenidos

La pila

La organización de la memoria en tiempo de ejecución es la siguiente:



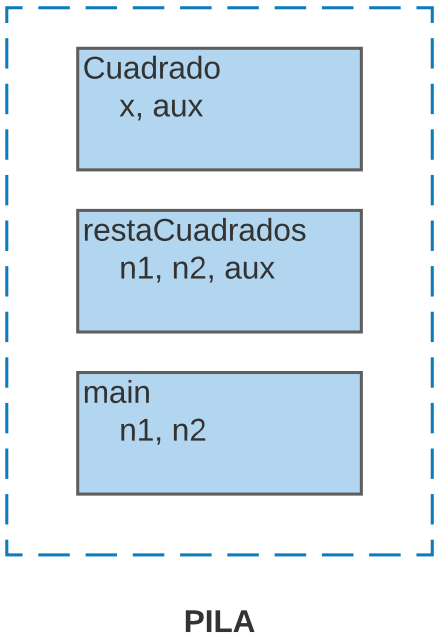
Cada vez que un módulo llama a otro, sus respectivas zonas de memoria se almacenan apiladas una encima de la otra en la memoria del ordenador: la Pila.

Ámbito de un dato

- El ámbito de una variable o constante v es el conjunto de módulos que pueden referenciar a v.
- Cada módulo tiene una zona de memoria independiente.
  - Los datos que hay en un módulo son invisibles para el resto de módulos, es decir, son locales al módulo.
  - Sólo se pueden usar en el módulo en el que están definidos (parámetro formal y dato local).
  - Sólo existen mientras la función está activa.
  - No pueden usarse en otros módulos, ni en el programa principal.
  - Los módulos se comunican a través de parámetros.
  - También se pueden comunicar con variables globales **pero no se debe realizar de esta forma**.

Ámbito de un dato

```
1  int cuadrado(int x){
2    int aux = x*x;
3    return aux;
4  }
5
6  int restaCuadrados(int n1, int n2){
7    int aux = cuadrado(n1)-cuadrado(n2);
8    return aux;
9  }
10
11 int main(){
12   int n1, n2;
13   cout << restaCuadrados(7,8);
14   return 0;
15 }
```



Paso de parámetros por valor

- La función recibe una copia de los valores de los parámetros.
- Se copia el valor de los parámetros actuales en los parámetros formales.
- Si el correspondiente parámetro se modifica dentro del módulo, esta modificación no se verá fuera del ámbito del módulo.

Ejemplo

```
1  void DemoLocal(int valor);
2
3  int main(){
4    int n=10;
5    cout << "Antes de llamar a DemoLocal: " << n << endl;
6    DemoLocal(n);
7    cout << "Despues de llamar a DemoLocal: " << n << endl;
8    return 0;
9  }
10
11 void DemoLocal(int valor){
12   cout << "Dentro de DemoLocal: " << valor << endl;
13   valor = 999;
14   cout << "Dentro de DemoLocal: " << valor << endl;
15 }
```

Índice de contenidos

Declaración vs definición

- En lenguaje C++, cada vez que se utiliza un módulo, ya sea de una biblioteca o definido por el programador, es necesario conocer previamente su cabecera o prototipo.
- Esto permite al compilador realizar la comprobación de tipos y número de argumentos.
- **Declaración** de una función o prototipo: es la cabecera de la función acabada en ";".
  <tipo devuelto> <nombre> (<param. formales>);
- **Definición** de una función:
  <tipo devuelto> <nombre> (<param. formales>){
  cuerpo de la función
  }

Organización

- **Primera forma:** en el mismo fichero, escribir los módulos al principio y el main al final.
- **Segunda forma:** en el mismo fichero, incluir los prototipos, luego el main y finalmente los módulos.
- **Tercera forma:** escribir los prototipos en un fichero de cabecera .h, y el main y los módulos en otro .cpp.
- **Cuarta forma:** prototipos en un fichero, main en otro y los módulos en otro.

Primera forma

En el mismo fichero .cpp:

```
1  #include <iostream>
2  using namespace std;
3
4  int maximo(int a, int b){ //Funciones
5    if(a>b)
6      return a;
7    return b;
8  }
9
10 int main(){ //Main
11   int num1 = 2, num2 = 3;
12   cout << "El numero mayor es: " << maximo(num1,num2) << endl;
13   return 0;
14 }
```

Segunda forma

En el mismo fichero .cpp:

```
1  #include <iostream>
2  using namespace std;
3
4  int maximo(int a, int b); //Prototipos
5
6  int main(){ //Main
7    int num1 = 2, num2 = 3;
8    cout << "El numero mayor es: " << maximo(num1,num2) << endl;
9    return 0;
10 }
11
12 int maximo(int a, int b){ //Funciones
13   if(a>b)
14     return a;
15   return b;
16 }
```

Tercera forma

En el fichero cabecera.h:

```
1 int maximo(int a, int b); //Prototipos
```

En el fichero .cpp:

```
1  #include <iostream>
2  #include "cabecera.h" //Incluimos nuestro fichero entre comillas
3  using namespace std;
4
5  int main(){ //Main
6    int num1 = 2, num2 = 3;
7    cout << "El numero mayor es: " << maximo(num1,num2) << endl;
8    return 0;
9  }
10
11 int maximo(int a, int b){ //Funciones
12   if (a>b)
13     return a;
14   return b;
15 }
```

Cuarta forma

En el fichero cabecera.h:

```
1 int maximo(int a, int b); //Prototipos
```

En el fichero cabecera.cpp:

```
1 #include"cabecera.h"
2 int maximo(int a, int b){ //Funciones
3   if (a>b)
4     return a;
5   return b;
6 }
```

En el fichero main.cpp:

```
1 #include <iostream>
2 #include "cabecera.h" //Incluimos nuestro fichero entre comillas
3 using namespace std;
4 int main(){ //Main
5   int num1 = 2, num2 = 3;
6   cout << "El numero mayor es: " << maximo(num1,num2) << endl;
7   return 0;
8 }
```

Cuarta forma

- Para compilar la forma 4:

```
g++ cabecera.cpp main.cpp -o ejecutable.out
```

- **NOSOTROS UTILIZAREMOS LA FORMA 4.**
- Las funciones y sus prototipos deben estar documentados.

```
/*
  Nombre: nombre de la función
  Tipo: tipo devuelto
  Objetivo: Explicación de lo que hace la función
  Parametros entrada:
  tipo nombreParametro1: Descripcion
  tipo nombreParametro2: Descripcion
  Precondiciones: En caso de ser necesario
  Valor devuelto: Que devuelve la funcion?
   Utiliza : funciones que invoque
  Autor: nombre del autor
  Fecha: fecha de creacion
*/
```

Índice de contenidos

Modularización con vectores

- Un módulo puede recibir como entrada un dato de tipo vector.
- Normalmente no se indica la dimensión del vector, sino una indicación genérica y el número de componentes útiles.

```
1 void imprimeVector(int vector[], int n);
2 void leeVector(int vector[], int n);
```

- Dado que un vector es un puntero (veremos esta notación con más detalle en el próximo tema), las siguientes cabeceras también son válidas.

```
1 void imprimeVector(int * vector, int n);
2 void leeVector(int * vector, int n);
```

- Cuando pasamos un vector a una función, podemos modificar el valor de sus elementos (se pasa por **referencia**).

Ejemplo

Calcular la media de los elementos de un vector:

```
1  float calcularMedia(float notas[], int n){
2    float media = 0;
3
4    for(int i=0; i<n; i++){
5      media+=v[i];
6    }
7
8    if(n!=0)
9      media=media/n;
10
11   return media;
12 }
```

Modularización con matrices

- Para pasar una matriz n-dimensional hay que especificar todas las dimensiones excepto la primera.
- Ejemplo: buscar un elemento en una matriz.

```
1  int buscarMatriz(int m[][MAX_COL], int nFil, int nCol, int
       elemento){
2    bool encontrado=false;
3    for (int i=0; i<nFil; i++){
4      for (int j=0; j<nCol; j++) {
5        if (m[i][j]==elemento){
6          encontrado=true;
7        }
8      }
9    }
10   return encontrado;
11 }
```

Modularización con estructuras

- Las estructuras se pueden pasar como parámetros a las funciones.
- Se tratan igual que los tipos básicos.
- En C++, además, una función puede devolver una estructura.

Ejemplo

Fichero cabecera.h:

```
1  #include<iostream>
2  using namespace std;
3
4  struct punto{
5    float x;
6    float y;
7  };
8
9  bool comparar(struct punto p1, struct punto p2);
10 struct punto suma(struct punto p1, struct punto p2);
```

Ejemplo

Fichero cabecera.cpp:

```
1  #include<iostream>
2  #include"cabecera.h"
3  using namespace std;
4
5  bool comparar(struct punto p1, struct punto p2){
6    if((p1.x==p2.x) && (p1.y==p2.y))
7      return true;
8    return false;
9  }
10
11 struct punto suma(struct punto p1, struct punto p2){
12   struct punto aux;
13   aux.x = p1.x + p2.x;
14   aux.y = p1.y + p2.y;
15   return aux;
16 }
```

Ejemplo

Fichero main.cpp:

```
1  #include<iostream>
2  #include"cabecera.h"
3  using namespace std;
4
5  int main(){
6    struct punto p1={1,2}, p2={3,4};
7    struct punto psuma;
8
9    (compara(p1,p2)) ?  cout << "Son iguales" << endl : cout << "
       Son distintos" << endl;
10
11   psuma = suma(p1,p2);
12   cout << "El punto suma es: (" << psuma.x << ", " << psuma.y <<
       ")" << endl;
13
14   return 0;
15 }
```

Índice de contenidos

Ficheros objeto

- Hasta ahora para compilar una serie de ficheros y generar el código ejecutable, hemos utilizado una sentencia del tipo:

```
1 g++ fichero1.cpp fichero2.cpp ... -o nombreejecutable.exe
```

- Un archivo objeto es un fichero intermedio entre un código fuente y un archivo ejecutable.
- Realmente un compilador genera el código objeto, mientras que el vinculador genera el código ejecutable.
- Para generar códigos objeto vamos a ejecutar sentencias del tipo:

```
1 g++ fichero1.cpp -c fichero1.o
```

Introducción

- Al programar, solemos crear funciones que son útiles en muchos programas, de igual o diferente índole.
- La opción más común para reutilizar estas funciones es copiar y pegar el código.
  - El tamaño del código fuente de los programas se incrementa innecesariamente y es redundante.
  - Para actualizar una versión es preciso modificar todos los programas que usan esta función.

Para solucionar este inconveniente se agrupan las funciones usadas frecuentemente en módulos biblioteca.

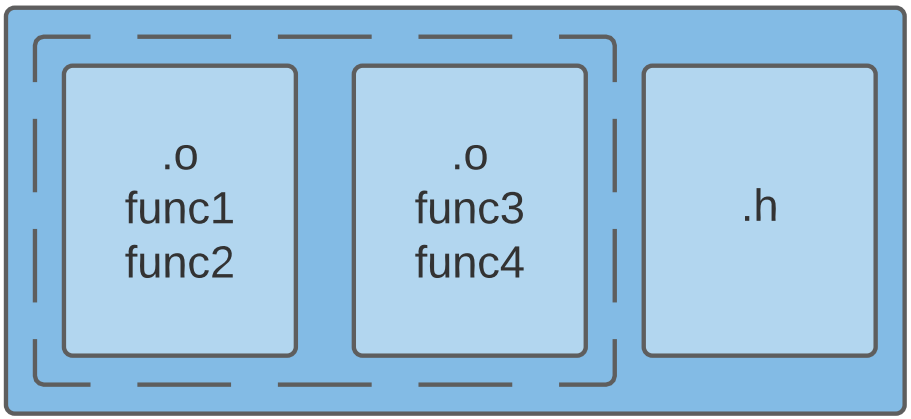
Biblioteca

Fichero que contiene el código objeto de un conjunto de funciones y que puede ser enlazado con el código objeto de un módulo que use una función de esa biblioteca.

- Sólo existe una versión de cada función. Actualización sencilla.
- Recompilar el módulo donde está esa función y los módulos que la usan.

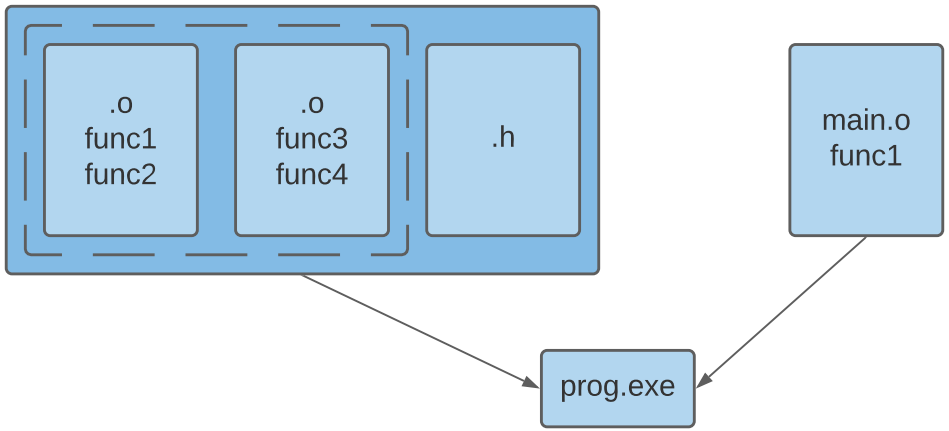
Estructura de una biblioteca

- Conjunto de módulos objeto (.o).
  - Cada uno es el resultado de la compilación de un fichero de código fuente .cpp que puede contener una o varias funciones.
  - La extensión por defecto de los ficheros de biblioteca es ".a" y su nombre suele empezar por "lib".
- Cada biblioteca lleva asociado un fichero de cabecera, con los prototipos de las funciones que se ofrecen (funciones públicas), que actúa de interfaz con los programas que la usan.



Construcción del ejecutable

- El enlazador enlaza el módulo objeto que contiene la función main() con el módulo objeto en que se encuentra la función de biblioteca usada.
- En el programa ejecutable sólo se incluyen los módulos objeto que contienen alguna función llamada por el programa.



Programa ar

- Finalmente, para crear la biblioteca utilizaremos el programa **ar** que permite:
  - Crear bibliotecas.
  - Modificar bibliotecas: añadir nuevos módulos, eliminar módulos objeto, reemplazar módulo objeto, etc.

El ejemplo más común de uso de la biblioteca es:

```
1 ar -r libejemplo.a funcs.o
2 ar -r libejemplo.a func01.o func02.o ... func10.o
```

Makefiles

- En este punto, la creación de un ejecutable pasa por distintos pasos: creación de ficheros objeto, creación de librerías y creación del ejecutable.
- Es posible que necesitemos hacer cambios por tanto, tendremos que repetir todos estos pasos.
- Una forma de automatizar el conjunto de pasos es el uso de Makefiles.
- Explicaremos su uso con el ejemplo de la siguiente diapositiva.

Ejemplo

```
#Fichero: makefile.mak
#Construye un ejecutable a partir de una libreria
all: main.x clean

main.x: main.o lib.a
    g++ -o main.x main.o lib.a

main.o: main.cpp cabecera.h
    g++ -c main.cpp

lib.a: cabecera.o
    ar -r lib.a cabecera.o

cabecera.o: cabecera.h cabecera.cpp
    g++ -c cabecera.cpp

.PHONY: clean
clean:
@echo Borrando ficheros.o ...
@rm *.o
```

Referencias

Recursos electrónicos:

- cppreference. (2020). Referencia C++: https://en.cppreference.com/w/
- Goalkicker.com (2018). C++: Note for Professionals.
- Grimes, R. (2017). Beginning C++ Programming.
- Joyanes, L. (2000). Programación en C++: Algoritmos, Estructuras de datos y Objetos.
- Juneja, B.L., Seth, A. (2009). Programming with C++.

Referencias

Libros:

- Prieto, A., Lloris, A., Torres, J.C. (2008). Introducción a la Informática (4 ed).
- Joyanes, L. (2008). Fundamentos de programación : algoritmos, estructura de datos y objetos.
- Martí, N., Ortega, Y., Verdejo, J.A. (2003). Estructuras de datos y métodos algorítmicos: Ejercicios resueltos.
- Tapia, S., García-Beltrán, A., Martínez, R., Jaen, J.A., del Álamo, J. (2012). Fundamentos de programación en C.

¿Preguntas?