Contenidos

1. Justificación del uso de funciones.
2. Declaración de funciones: prototipos.
3. Prototipos y ficheros de cabecera.
4. Polimorfismo (sobrecarga de funciones).
5. Argumentos formales y actuales.
    a) modificador de argumentos : const
    b) valores por defecto para los argumentos
6. Matrices como argumento de funciones.

Justificación del uso de Funciones

El uso de funciones en los programas permite:

- **Organización**: nuestros algoritmos serán más legibles y más ordenados.
- **Reutilización de código**. Las funciones diseñadas por el usuario pueden ser utilizadas en cualquier otro programa que las necesite.

Algunos autores mantienen que la cantidad de instrucciones que se incluyen en una función debe limitarse al código que pueda incluirse en una pantalla. Esto permite tener una visión completa de la misma de un solo vistazo y será más fácil de entender el funcionamiento .

Declaración de las funciones : Prototipos

- Las declaraciones de las funciones, también llamados prototipos, deben realizarse antes de usar la función.
- El **prototipo**, informa de la existencia de la función, el tipo de datos que devuelve y los parámetros que tiene definidos.
- En ocasiones la declaración y la implementación se realiza en el mismo punto, aunque es normal colocar al principio del programa principal los "prototipos" de las funciones que serán utilizadas en su interior, y las implementaciones al final.

```
float cuadrado (float x);                    // prototipo o declaración
float cuadrado (float x)                     // codificación o definición
{
   return   x*x  ;
}
```


Declaración de las funciones : Prototipos

Ejemplo:

```
int funcion_tres(char c, int i);
```

Permite al compilador efectuar una comprobación de tipos de los argumentos que pasan y del valor devuelto

El compilador puede realizar un modelado de tipo ("Casting") de los argumentos para garantizar que coinciden con el tipo esperado.

Declaración de las funciones : Prototipos

Ejemplo:

```
int fun_max ( int v1, int v2);              // prototipo

void main()
 {
   float limite = 32;
   char c = 'A';
   int mval;
   mval = fun_max(limite, c);   // Llamada a la funcion.
}
```

Antes de utilizar una función debe de estar anterioremente declarada o definida, sino se producirá un error de compilacion.
Los tipos de los parámetros actualese tienen que ser los mismo que los del prototipo (parámetros formales) o que se pueda hacer un casting de tipos por el compilador.

Declaración de las funciones :

Prototipos y ficheros de cabecera

Es costumbre que los prototipos de las funciones incluidas en las librerías del lenguaje se agrupen en ficheros específicos, los denominados ficheros de cabecera, que son ficheros de texto en los que se agrupan todas las declaraciones que se utilizan en la librería.

MisFunciones.h

MisFunciones.cpp

Librería: Contiene las implementaciones de las funciones del fichero de cabecera.

Fichero de cabecera

Contiene prototipos de funciones, definición de estructuras, macros y clases.

Declaración de las funciones :

Prototipos y ficheros de cabecera

- El hecho de tener agrupados los prototipos en un fichero de cabecera es de gran utilidad, porque solo es preciso incluir dicha librería mediante la directiva ya conocida **include** al principio del programa.

```
#include <MisFunciones.h>
  ….
```

- Así podemos estar seguros de que todos los prototipos estarán presentes. De otro modo tendríamos que escribirlos manualmente en cada programa que usara funciones de la librería.

Polimorfismo

C++ permite realizar **sobrecarga de funciones.**

- Esto significa que puedan definirse varias funciones con el mismo nombre pero con distintos parámetros, y distintas implementaciones.
- Cuando hacemos una llamada a una función sobrecargada, el compilador es capaz de saber a cual de ellas nos estamos refiriendo analizando los parámetros que pasamos a la función.

Proceso denominado ***resolución de sobrecarga***

Ejemplo:

```
int  area  ( int a, int b );
float  area    (int radio );
int  area ( float c, float d, float e );
```

```
void main()
{
   int m;
   m = area( 2 , 3 );

   m = area( 2.3 , 8.3, 7.0 );
}
```

Polimorfismo

Resolución de sobrecarga: El compilador aplica ciertas reglas para verificar cual de las declaraciones se ajusta mejor al número y tipo de los argumentos utilizados. ¡ Busca máxima concordancia !

- Si el compilador encuentra que dos funciones distintas concuerdan, entonces se produce un error: existe ambigüedad.
- Los valores devueltos por las funciones no son tenidos en cuenta a efectos del mecanismo de sobrecarga.

```
int funcion_a (int x, char y);        //  primer prototipo
int funcion_a (float a, char c);       // correcto
void funcion_a (int i, char c);    // Error prototipo repetido con 1º
```


Polimorfismo

Uso de la sobrecarga:

Solo tiene sentido usar sobrecarga cuando se asigna el mismo nombre a funciones que realizan tareas similares en objetos diferentes.

El ejemplo anterior teníamos tres prototipos para la misma función. Las tres funciones calculan el área de una figura geométrica.

```
int area (int a, int b)
{
   int aux;
   aux = a*b;
   return aux;
}
```

Area del rectángulo

```
float area (int radio)
{
    return ( pi *radio * radio );
}
```

Area del cículo

Argumentos formales y actuales

Los parámetros formales son los que se definen en el prototipo y en la implementación de la función.

Los parámetros actuales son los valores pasados.

```
int area_rectangulo (int a, int b);
```

Parámetros formales

```
int area_rectangulo (int a, int b)
{
   int aux;
   aux = a*b;
   a=0;
   b=0;
   return aux;
}
```

Parámetros actuales

```
void main()
{
   int m;
   int lado1 = 2, lado2 = 6 ;
   m = area_rectangulo( lado1 , lado2 );
   cout << m;
}
```


Argumentos formales y actuales

Modificador de parámetros CONST:

En la lista de parámetros formales podemos usar el modificador CONST.

- Esto se utiliza cuando se pasan parámetros por referencia y queremos garantizar que la función no modifica el valor recibido.
- No tiene sentido utilizarlo para parámetros pasados por valor.

```
int  func_vector ( const int v[ ] ) ;

int  calcula ( const int& a) ;
```

```
bool funcion_ejemplo ( int a, const int& b)
{
        b  =b+a; //ERROR !!
        cout <<  b ;
        return true;
}
```

Argumentos por defecto

C++ permite tener valores por defecto para los parámetros; esto supone que si no se pasa el parámetro correspondiente, se asume un valor predefinido.

```
float calcula ( float x, float y = 0);
```

```
void main()
{
   ...
   m1 = calcula (2.0 , 0);
   m2 = calcula (2.0);
  …
}
```

Ambas llamadas son correctas y producen el mismo resultado

```
float calcula (float a, float b)
 {
    return (a*b);
}
```

Los parámetros con valores por defecto deben ser los últimos de la lista.

Matrices como argumento de funciones

Los arrays se pasan siempre por referencia.

Cuando hay que pasar una matriz bidimensional como argumento de una función, debemos especificar al menos el número de columnas, ya que dicha función debe conocer la estructura interna de la matriz.

```
void  visualizar_matriz( int m[3][4] )
   { … }
```

```
void  modificar_matriz( int m[][4] )
   { … }
```

Recordar que en C++ es responsabilidad del programador no salirse del ámbito de la matriz al acceder a sus elementos.