Funciones en C++

Ing José Luis MARTÍNEZ

15 de octubre de 2021

1. Funciones

1.1. Introducción

Para resolver problemas complejos o de gran tamaño es conveniente utilizar el concepto de reducción de problemas, o estrategia de divide y vencerás. El problema se descompone en subproblemas, los cuales a su vez pueden descomponerse en subsubproblemas, y así continuar hasta que el problema original queda reducido a un conjunto de actividades básicas, que no se pueden o no conviene volver a descomponer. La solución de cada una de estas actividades básicas permite, luego, aplicando razonamiento hacia atrás, la solución del problema final.

En el lenguaje de programación C++, al igual que en C, la solución de un problema se expresa por medio de un programa; la solución de un subproblema, por medio de una función. El uso de funciones tiene múltiples ventajas:

1. facilitan la lectura y escritura de los programas,
2. permiten el trabajo en paralelo — diferentes programadores se pueden encargar de diferentes funciones —
3. facilitan la asignación de responsabilidades,
4. permiten que el código de la función se escriba solamente una vez y se utilice tantas veces como sea necesario,
5. facilitan el mantenimiento de los programas, etc.

De esta manera, un programa en C++ está constituido por un programa principal y un conjunto de funciones. El programa principal consta generalmente de pocas líneas, las cuales pueden ser llamadas a funciones. La llamada a una función indica al procesador que debe continuar con el procesamiento de la función. Una vez que ésta concluye, el control regresa al punto de partida en el programa principal. Por otra parte, la función se escribe de forma similar al programa principal, pero con diferencias principalmente en el encabezado de la misma.
Una función resuelve un subproblema de forma independiente y se ejecuta sólo cuando recibe una llamada desde el programa principal o desde otras funciones. El lenguaje de programación C++ permite que una función pueda incorporar llamadas a otras funciones.

La comunicación entre las funciones y el programa principal, al igual que entre las mismas funciones, se lleva a cabo por medio de

1. parámetros por valor,
2. parámetros por referencia y
3. variables globales. Estas últimas son menos utilizadas por razones de eficiencia y seguridad en la escritura de programas.

Hemos utilizado funciones que pertenecen a bibliotecas del lenguaje C++. Por ejemplo, la función *cout* que se encuentra en la biblioteca *iostream.h* se ha utilizado en numerosas ocasiones. Pero cabe aclarar que las funciones que utilizaremos en este capítulo son diferentes, porque tienen la particularidad de que nosotros mismos las desarrollaremos. Es decir, no se encuentran en ninguna biblioteca del lenguaje C++.

2. Estructura de una función

Una función es un conjunto de instrucciones que se pueden llamar desde cualquier parte de un programa. En C++ todas las funciones son externas o globales, es decir, pueden ser llamadas desde cualquier punto del programa. Las funciones en C++ no se pueden anidar (no puede declararse una función dentro de otra función), pero una función1 puede llamar a una función 2, y esta a una función 3, etc. La estructura de una función en C++ se muestra a continuación:

tipo_de_retorno nombre_Función(Lista_de_Parámetros_Formales)
{
cuerpo de la función
return expresión;
}

tipo_de_retorno: Tipo de valor devuelto por la función o
la palabra reservada void si la función no devuelve ningún valor. *nombre_Función* Identificador o nombre de la función.
Lista_de_Parámetros_Formales Lista de declaraciones de los parámetros de la función separados por comas.
expresión Valor que devuelve la función.

Los aspectos más sobresalientes en el diseño de una función son:

- **Tipo_de_retorno**. Es el tipo de dato que devuelve la función C++. El tipo debe ser uno de los tipos simples de C++, tales como int, char o float, o un puntero a cualquier tipo C++, o un tipo struct, previamente declarado. Si no devuelve ningún valor el tipo es void.
- **nombre_Función**. Un nombre de una función comienza con una letra o un subrayado (_) y puede contener tantas letras, números o subrayados como desee.
- **Lista_de_Parámetros_Formales**. Es una lista de parámetros con tipos que utilizan el formato siguiente: tipo1 parámetro1, tipo2 parámetro2, ...
- **Cuerpo de la función.** Se encierra entre llaves de apertura ({) y cierre (}).
- **Paso de parámetros.** Posteriormente se verá que el paso de parámetros en C++ se puede hacer por valor y por referencia.
- **Declaración local.** Las constantes, tipos de datos y variables declaradas dentro de la función son locales a la misma y no perduran fuera de ella.
- **Valor devuelto por la función.** Una función puede devolver un único valor. Mediante la palabra reservada return se puede devolver el valor de la función. Una función puede tener cualquier número de sentencias return. Tan pronto como el programa encuentra cualquiera de las sentencias return, se retorna a la sentencia llamadora.
- **La llamada a una función.** Una llamada a una función redirigirá el control del programa a la función nombrada. Debe ser una sentencia o una expresión de otra función que realiza la llamada. Esta sentencia debe ser tal que debe haber coincidencia en número, orden y tipo entre la lista de parámetros formales y actuales de la función.
- **No se pueden declarar funciones anidadas.** Todo código de la función debe ser listado secuencialmente, a lo largo de todo el programa. Antes de que aparezca el código de una función, debe aparecer la llave de cierre de la función anterior.

Ejemplo. La función min devuelve el menor de los dos números enteros que se pasan como parámetro. La función main itera llamadas a min.

```
#include <cstdlib>
#include <iostream>
using namespace std;
int min ( int x, int y)
{
   if (x < y)
      return x;
   else
      return y;
}

int main( int argc, char *argv[])
{
   int m, n;
   do
   {
      cout << "introduzca dos numeros. Si primero es cero fin " ;
      cin >> m >> n;
      if ( m != 0)
         cout << " el menor es :" << min(m, n) << endl; ////Llamada a min
   } while(m != 0);

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Ejemplo. La función norma devuelve la norma euclídea (también denominada norma 2) de las tres coordenadas de un vector de R^3 (espacio de tres dimensiones).

$$Norma(x, y, z) = \sqrt{x^2 + y^2 + z^2}$$

```
#include <cstdlib>
#include <iostream>
#include <math.h> //contiene función Sqrt
using namespace std;
float Norma ( float x, float y, float z)
{
   return sqrt(x * x + y * y + z * z);
}
int main( int argc, char *argv[])
{
   float x, y, z;
   cout << " vector : (" << 3 << ","<< 4 << ","<< 5 << ")" ;
   cout << " norma = " << Norma(3, 4, 5) << endl; //Llamada a norma
   cout << " introduzca las tres coordenadas de vector " ;
   cin >> x >> y >> z;
   cout << " vector : (" << x << "," << y << "," << z << ")" ;
   cout << " norma = " << Norma(x, y, z) << endl; //Llamada a norma
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

3. Prototipos de las funciones

C++ requiere que una función se declare o defina antes de su uso. La declaración de una función se denomina prototipo. Específicamente un prototipo consta de los siguientes elementos:

nombre de la función; lista de parámetros formales encerrados entre paréntesis y un punto y coma

Los prototipos se sitúan normalmente al principio de un programa, antes de la definición de la función main() . La definición completa de la función debe existir en algún lugar del programa; por ejemplo, antes o después de main.

El compilador utiliza los prototipos para validar que el número y los tipos de datos de los parámetros actuales de la llamada a la función son los mismos que el número y tipo de parámetros formales. Si una función no tiene argumentos, se ha de utilizar la palabra reservada *void* como lista de argumentos del prototipo (también se puede escribir paréntesis vacíos). Un formato especial de prototipo es aquel que tiene un número no especificado de argumentos, que se representa por tres puntos (...).

Ejemplo. Prototipo, llamada y definición de función. La función media recibe como parámetros dos números y retorna su media aritmética.

```
#include <cstdlib>
#include <iostream>
using namespace std;

double media ( double x1, double x2); //Declaración de media. Prototipo

int main( int argc, char *argv[])
{
   double med, numero1, numero2;
   cout << "introduzca dos numeros " ;
   cin >> numero1 >> numero2;
   med = media (numero1, numero2); //Llamada a la función
   cout << " lamedia es :" << med << endl;
   system("PAUSE");
   return EXIT_SUCCESS;
}

double media( double x1, double x2) //Definición de media
{
   return (x1 + x2)/2;
}
```

Ejemplo. Escribir una función que tenga como parámetro dos números enteros positivos num1 y num2, y calcule el resto de la división entera del mayor de ellos entre el menor mediante suma y restas.

Solución: Un programa principal lee los dos números asegurándose que son positivos mediante un bucle do-while y llama a la función resto que se encargará de resolver el problema. La función resto, en primer lugar, determina el mayor y el menor de los dos números almacenándolos en las variables Mayor y menor. Mediante un acumulador inicializado a la variable menor y mediante un bucle while se suma al acumulador el valor de menor, hasta que el valor del acumulador sea mayor que el número Mayor. Necesariamente el resto debe ser el valor de la variable Mayor menos el valor de la variable acumulador más el valor de la variable menor.

```
#include <cstdlib>
```

```
#include <iostream>
using namespace std;

int resto(int n, int m);  // prototipo de función declarada al igual que en C

int main(int argc, char *argv[])
{
    int n, m;
    do
        {
            cout << " ingrese dos numeros :"; cin >> n >> m ;
      } while ((n <= 0) || (m <= 0));

     cout << " el resto es: " << resto(n,m) << endl;
     cout << "presione una tecla para finalizar";
     cin.get();
     return EXIT_SUCCESS;
}

int resto(int num1, int num2)  // función que calcula el resot
{
    int Mayor, menor, acu;  // variables locales
    if (num1 < num2)
        {
            Mayor = num2;
            menor = num1;
        }
    else
        {
            Mayor = num1;
            menor = num2;
        }
    acu = menor;

   while (acu <= Mayor)
    acu += menor;

   return (Mayor - acu + menor);
}
```

Ejemplo. Prototipo sin nombres de parámetros en la declaración y sin parámetros formales. Calcula el área de un rectángulo. El programa se descompone en dos funciones, además de main(). La función entrada retorna un número real que lee del teclado. La función area calcula el área del rectángulo cuyos lados recibe como parámetros.

```
#include <cstdlib>
#include <iostream>
using namespace std;

float area_r( float, float); //Prototipo. Nombres parámetros omitidos
float entrada(); //Prototipo sin parámetros

int main( int argc, char *argv[])
```

```
{
   float base, altura;
   cout << " Base del rectangulo. ";
   base = entrada();
   cout << " Altura del rectangulo. ";
   altura = entrada();
   cout<< " Area rectangulo: "<< area_r(base,altura) << endl;
   system("PAUSE");
   return EXIT_SUCCESS;
}

float entrada() //Retorna un numero positivo
{
   float m;
   do
   {
       cout << " Numero positivo: ";
       cin >> m;
   } while (m <= 0.0);
   return m;
}

float area_r(float b, float a) //Se declaran los nombres de parámetros
{
   return (b * a);
}
```

Ejemplo. Realice un programa que simule un juego de dados, pero sin utilizar namespace.

```
#include <iostream>
#include <cstdlib> // contiene los prototipos para las funciones srand y rand
#include <ctime> // contiene el prototipo para la función time


int tirarDados(); // tira los dados, calcula y muestra la suma
int main()
{
    // enumeración con constantes que representa el estado del juego
    enum Estado { CONTINUAR, GANO, PERDIO }; // todas las constantes en mayúsculas
    int miPunto; // punto si no se gana o pierde en el primer tiro
    Estado estadoJuego; // puede contener CONTINUAR, GANO o PERDIO
    // randomiza el generador de números aleatorios, usando la hora actual
    std::srand( time( 0 ) );
    int sumaDeDados = tirarDados(); // primer tiro del dado
    // determina el estado del juego y el punto (si es necesario), con base en el primer tiro
    switch ( sumaDeDados )
    {
        case 7: // gana con 7 en el primer tiro
        case 11: // gana con 11 en el primer tiro
            estadoJuego = GANO;
            break;
        case 2: // pierde con 2 en el primer tiro
        case 3: // pierde con 3 en el primer tiro
        case 12: // pierde con 12 en el primer tiro
```

```
            estadoJuego = PERDIO;
            break;
        default: // no ganó ni perdió, por lo que recuerda el punto
            estadoJuego = CONTINUAR; // el juego no ha terminado
        miPunto = sumaDeDados; // recuerda el punto
        std::cout << "El punto es " << miPunto << std::endl;
        break; // opcional al final del switch
    } // fin de switch

    // mientras el juego no esté completo
    while ( estadoJuego == CONTINUAR ) // no ganó ni perdió
    {
        sumaDeDados = tirarDados(); // tira los dados de nuevo
        // determina el estado del juego
        if ( sumaDeDados == miPunto ) // gana al hacer un punto
            estadoJuego = GANO;
        else
            if ( sumaDeDados == 7 ) // pierde al tirar 7 antes del punto
            estadoJuego = PERDIO;
    } // fin de while
    // muestra mensaje de que ganó o perdió
    if ( estadoJuego == GANO )
        std::cout << "El jugador gana" << std::endl;
    else
        std::cout << "El jugador pierde" << std::endl;
    return 0; // indica que terminó correctamente
    } // fin de main

    // tira los dados, calcula la suma y muestra los resultados
    int tirarDados()
    {
        // elige valores aleatorios para el dado
        int dado1 = 1 + rand() % 6; // tiro del primer dado
        int dado2 = 1 + rand() % 6; // tiro del segundo dado
        int suma = dado1 + dado2; // calcula la suma de valores de los dados

        // muestra los resultados de este tiro
        std::cout << "El jugador tiro " << dado1 << " + " << dado2
        << " = " << suma << std::endl;
        return suma; // devuelve la suma de los dados
    } // fin de la función tirarDados
```

4. Parámetros de una función

C++ proporciona dos métodos para pasar variables (parámetros) entre funciones. Una función puede utilizar parámetros por valor y parámetros por referencia, o puede no tener parámetros.

4.1. Paso por valor.

significa que cuando C++ compila la función y el código que llama a la misma, la función recibe una copia de los valores de los parámetros actuales. La función receptora no puede modificar la variable de la función (parámetro pasado).

Ejemplo. Paso de parámetros por valor a una función .

```
#include <cstdlib>
#include <iostream>
using namespace std;

void paso_por_valor( int x); //Prototipo

int main( int argc, char *argv[])
{
   int x =20;
   cout << " antes de la llamada a paso_por_valor " << x << endl;
   paso_por_valor(x);
   cout << " despues de la llamada a paso_por_valor " << x << endl;
   system("PAUSE");
   return EXIT_SUCCESS;
}

void paso_por_valor( int x)
{
   cout << " dentro de paso_por_valor " << x << endl;
   x *= 2;
   cout << " despues de x *=2 y dentro de paso_por_valor " << x << endl;
}
```

4.2. Paso por referencia.

Cuando una función debe modificar el valor del parámetro pasado y de volver este valor modificado a la función llamadora, se ha de utilizar el método de paso de parámetro por referencia o dirección.

Para declarar una variable que es parámetro formal, como paso por referencia, el símbolo *&* debe preceder al nombre de la variable en la cabecera de la función, y en la llamada el parámetro actual correspondiente debe ser el nombre de una variable. También puede usarse el método de los punteros de C: en la declaración de la variable que es parámetro formal, el símbolo $*$ debe preceder al nombre de la variable; en la llamada a la función debe realizarse el parámetro actual que debe ser & variable.

Cuando se modifica el valor del parámetro formal de un parámetro por referencia (la variable local), este valor queda almacenado en la misma dirección de memoria, por lo que al retornar a la función llamadora la dirección de la memoria donde se almacenó el parámetro contendrá el valor modificado.

Ejemplo. Paso de parámetros por referencia a una función, estilo C++.

```
#include <cstdlib>
#include <iostream>

using namespace std;

void referencia(int& x) //Parámetro por referencia
{
   x += 2;
}

int main(int argc, char *argv[])
{
```

```
   int x = 20;
   cout << " antes de la llamada " << " x= " << x << endl;
   referencia (x); //Llamada con nombre de variable
   cout << " despues de la llamada " <<" x= " << x << endl;
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Ejemplo. Paso de parámetros por referencia a una función, estilo C.

```
#include <cstdlib>
#include <iostream>
using namespace std;

void intercambio( int* x, int* y) //Declaración como puntero
{
   int aux = *x;
   *x = *y;
   *y = aux;
}

int main( int argc, char *argv[])
{
   int x = 20, y = 30 ;
   cout << " antes de la llamada " ;
   cout <<" x= " << x << " y= " << y << endl;
   intercambio (&x, &y); //Llamada con dirección
   cout << " despues de la llamada " ;
   cout <<" x= " << x << " y= " << y << endl;
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Ejemplo. Paso de parámetros a la función por referencia

```
#include <cstdlib>
#include <iostream>
using namespace std;
void referencia(int& x) //Parámetro por referencia
{
    x += 2;
}

int main(int argc, char *argv[])
{
    int x = 20;
    cout << " antes de la llamada " << " x= " << x << endl;
    referencia (x); //Llamada con nombre de variable
    cout << " despues de la llamada " <<" x= " << x << endl;
    cin.get();
    return EXIT_SUCCESS;
}
```

4.3. Parámetros *const* en una función.

El especificador *const*, indica al compilador que sólo es de lectura en el interior de la función. Si se intenta modificar en este parámetro se producirá un mensaje de error de compilación.

```
void error_en_compilacion( const int x, const float &y)
{
   x = 123; //Fallo en tiempo de compilación
   y = 21.2; //Fallo en tiempo de compilación
}
```

4.4. Argumentos por omisión o defecto.

Una característica poderosa de las funciones C++ es que en ellas pueden establecer valores por omisión o ausencia ("por defecto") para los parámetros. Se pueden asignar argumentos por defecto a los parámetros de una función. Cuando se omite el argumento de un parámetro que es un argumento por defecto, se utiliza automáticamente éste. La única restricción es que se deben incluir todas las variables desde la izquierda hasta el primer parámetro omitido. Si se pasan valores a los argumentos omitidos se utiliza ese valor; si no se pasa un valor a un parámetro opcional, se utiliza el valor por defecto como argumento. El valor por defecto debe ser una expresión constante.

Ejemplo. La función asteriscos tiene tres parámetros. El primero indica el número de filas, el segundo indica el número de columnas y el tercero el carácter a escribir. El segundo y el tercer parámetros son por omisión.

```
#include <cstdlib>
#include <iostream>

using namespace std;

void asteriscos( int fila, int col =3, char c = '*')
{
   for( int i = 0; i < fila; i++)
   {
      for ( int j = 0; j < col; j++)
         cout << c;
      cout <<endl;
   }
}

int main( int argc, char *argv[])
{
   asteriscos(4); //Correcto dos parámetros por omisión
   cout << endl;
   asteriscos( 4,6); //Correcto un parámetro por omisión
   cout << endl;
   asteriscos(4,6,'@');
   //asteriscos()llamada incorrecta. Primer parámetro obligatorio
   //asteriscos(4, , '@'); llamada incorrecta
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

4.4.1. Reglas de construcción de argumentos por defecto

- Los argumentos por defecto se deben pasar por valor. Nunca por referencia.
- Los valores de los argumentos por defecto pueden ser valores literales o definiciones *const*. No pueden ser variables.
- Todos los argumentos por defecto deben colocarse al final en el prototipo de la función. Después del primer argumento por defecto, todos los argumentos posteriores deben incluir también valores por defecto.

5. Funciones en línea *(inline)*

Existen dos opciones disponibles para la generación del código de las funciones en C++: funciones en línea y fuera de línea. Las funciones en línea *(inline)* sirven para aumentar la velocidad de su programa. Su uso es conveniente cuando la función se utiliza muchas veces en el programa y su código es pequeño. Para una función en línea *(inline)* , el compilador inserta realmente el código para la función en el punto en que se llama a la función. Esta acción hace que el programa se ejecute más rápidamente.

Una función normal, fuera de línea, es un bloque de código que se llama desde otra función. El compilador genera código para situar la dirección de retorno en la pila. La dirección de retorno es la dirección de la sentencia que sigue a la instrucción que llama a la función. A continuación, el compilador genera códigos que sitúan cualquier argumento de la función en la pila a medida que se requiera. Por último, el compilador genera una instrucción de llamada que transfiere el control a la función.

	Ventajas	**Desventajas**
Funciones en línea	Rápida de ejecutar.	Tamaño de código grande.
Funciones fuera de línea	Pequeño tamaño de código.	Lenta de ejecución.

Para crear una función en línea (inline) , insertar la palabra reservada *inline* delante de una declaración normal y del cuerpo, y situarla en el archivo fuente antes de que sea llamada. La sintaxis de declaración es:

inline TipoRetorno NombreFunción (Lista parámetros con tipos)
{ *cuerpo*}

Ejemplo. Funciones en línea para calcular el volumen y el área total de un cilindro del que se leen su radio y altura.

El volumen de un cilindro viene dado por: $volumen = \pi * radio^2 * altura$. El Area total viene dada por $Areatotal == 2 * \pi * radio * altura + 2 * \pi * radio^2$. Para resolver el problema basta con declarar las variables correspondientes, declarar la constante *pi* y las dos funciones en línea que codifican las fórmulas respectivas. El programa principal lee el radio y la altura en un bucle *while* garantizando su valor positivo.

```
#include <cstdlib>
#include <iostream>

using namespace std;

const float Pi = 3.141592;

inline float VOLCILINDRO( float radio, float altura) // Función en línea
{
   return (Pi * radio * radio * altura);
}
inline float AREATOTAL( float radio, float altura) // Función en línea
```

```
{
   return (2 * Pi * radio * altura + 2 * Pi * radio * radio);
}
int main( int argc, char *argv[])
{
    float radio, altura, Volumen, Areatotal;
    do
    {
       cout << "Introduzca radio del cilindro positivo: ";
       cin >> radio;
       cout << "Introduzca altura del cilindro positiva: ";
       cin >> altura;
    } while (( radio <= 0) || (altura <= 0));
    Volumen = VOLCILINDRO(radio, altura);//llamada sustituye el código
    // la sentencia anterior es equivalente a
    // Volumen = Pi*radio*radio*altura;
    Areatotal = AREATOTAL(radio, altura);//llamada sustituye el código
    // la sentencia anterior es equivalente a
    // Areatotal = 2*Pi*radio*altura+2*Pi*radio*radio;
    cout << "El volumen del cilindro es:" << Volumen << endl;
    cout << "El Area total del cilindro es:" << Areatotal << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

También son funciones en línea las definidas con la sentencia define. La sintaxis general es: *#define NombreMacro(parámetros sin tipos) expresión_texto*

La definición ocupará sólo una línea, aunque si se necesita más texto, se puede situar una barra in vertida ( ) al final de la primera línea y continuar en la siguiente, en caso de ser necesarias más líneas proceder de igual forma; de esa manera se puede formar una expresión más compleja. Entre el nombre de la macro y los paréntesis de la lista de argumentos no puede haber espacios en blanco. Es importante tener en cuenta que en las macros con argumentos no hay comprobación de tipos.

Ejemplo. Función en línea para definir una función matemática.

```
#include <cstdlib>
#include <iostream>

#define fesp(x) (x * x + 2 * x {1)

using namespace std;

int main( int argc, char *argv[])
{
   float x;
   for (x = 0.0; x <= 6.5; x += 0.3)
      cout << x << " " << fesp(x) << endl;
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

6. Ámbito o alcance.

El ámbito es la zona de un programa en el que es visible una variable. Existen cuatro tipos de ámbitos: programa, archivo fuente, función y bloque. Normalmente la posición de la sentencia en el programa determina el ámbito. Su aplicación es como ya hemos visto en C.

- Las variables que tienen ámbito de programa pueden ser referenciadas por cualquier función en el programa completo; tales variables se llaman variables globales. Para hacer una variable global, declárela simplemente al principio de un programa, fuera de cualquier función.
- Una variable que se declara fuera de cualquier función y cuya declaración contiene la palabra reservada static tiene ámbito de archivo fuente. Las variables con este ámbito se pueden referenciar desde el punto del programa en que están declaradas hasta el final del archivo fuente. Una variable *static* es aquella que tiene una duración fija. El espacio para el objeto se establece en tiempo de compilación; existe en tiempo de ejecución y se elimina sólo cuando el programa desaparece de memoria en tiempo de ejecución.
- Una variable que tiene ámbito de una función se puede referenciar desde cualquier parte de la función. Las variables declaradas dentro del cuerpo de la función se dice que son locales a la función.
- Una variable declarada en un bloque tiene ámbito de bloque y puede ser referenciada en cualquier parte del bloque, desde el punto en que está declarada hasta el final del bloque. Las variables locales declaradas dentro de una función tienen ámbito de bloque de la función; no son visibles fuera del bloque.

Ejemplo

```
#include <iostream>
#include <cstdlib>
using namespace std;

void usarLocal(); // prototipo de función
void usarLocalStatic(); // prototipo de función
void usarGlobal(); // prototipo de función

int x = 1; // variable global

int main(int argc, char *argv[])
{
    cout << "la x global en main es " << x << endl;

    int x = 5; // variable local para main

    cout << "la x local en el alcance exterior de main es " << x << endl;
    { // empieza nuevo alcance
        int x = 7; // oculta la x en el alcance exterior y la x global
        cout << "la x local en el alcance interior de main es " << x << endl;
    } // termina nuevo alcance

    cout << "la x local en el alcance exterior de main es " << x << endl;
    usarLocal(); // usarLocal tiene la x local
    usarLocalStatic(); // usarLocalStatic tiene la x local estática
    usarGlobal(); // usarGlobal usa la x global
    usarLocal(); // usarLocal reinicializa su x local
```

```
    usarLocalStatic(); // la x local estática retiene su valor anterior
    usarGlobal(); // la x global también retiene su valor anterior

    cout << "\nla x local en main es " << x << endl;
    cin.get();
    return EXIT_SUCCESS; // indica que terminó correctamente
} // fin de main

// usarLocal reinicializa la variable x local durante cada llamada
void usarLocal()
{
    int x = 25; // se inicializa cada vez que se llama a usarLocal

    cout << "\nla x local es " << x << " al entrar a usarLocal" << endl;
    x++;
    cout << "la x local es " << x << " al salir de usarLocal" << endl;
} // fin de la función usarLocal

// usarLocalStatic inicializa la variable x local estática sólo la
// primera vez que se llama a la función; el valor de x se guarda
// entre las llamadas a esta función
void usarLocalStatic()
{
    static int x = 50; // se inicializa la primera vez que se llama a usarLocalStatic
    cout << "\nla x local estatica es " << x << " al entrar a usarLocalStatic"
    << endl;
    x++;
    cout << "la x local estatica es " << x << " al salir de usarLocalStatic"
    << endl;
} // fin de la función usarLocalStatic

// usarGlobal modifica la variable global x durante cada llamada
void usarGlobal()
{
    cout << "\nla x global es " << x << " al entrar a usarGlobal" << endl;
    x *= 10;
    cout << "la x global es " << x << " al salir de usarGlobal" << endl;
} // fin de la función usarGlobal
```

7. Operador de resolución de ámbito unario.

Es posible declarar variables locales y globales con el mismo nombre. C++ proporciona el operador de resolución de **alcance binario (::)** para acceder a una variable global cuando una variable local con el mismo nombre se encuentra dentro del alcance. El operador de resolución de alcance unario no se puede utilizar para acceder a una variable local con el mismo nombre en un bloque exterior. Se puede acceder a una variable global directamente sin el operador de resolución de ámbito unario, si el nombre de la variable global no es el mismo que el de una variable local dentro del alcance.
En el ejemplo 8 se demuestra el operador de resolución de ámbito unario con variables local y global con el mismo nombre. Para enfatizar que las versiones local y global de la variable numero son distintas, el programa declara una variable de tipo int y la otra de tipo double.
El uso del operador de resolución de alcance unario (::) con un nombre de variable dado es opcional cuando la única variable con ese nombre es una variable global.

Ejemplo. Uso del operador de alcance ::

```
#include <iostream>
using namespace std;

int numero = 7; // variable global llamada numero

int main()
{
    double numero = 10.5; // variable local llamada numero

    // muestra los valores de las variables local y global
    cout << "Valor local double de numero = " << numero
    << "\nValor global int de numero = " << ::numero << endl;
    cin.get();
    return 0; // indica que terminó correctamente
} // fin de main
```

8. Clases de almacenamiento.

Los especificadores de clases (tipos) de almacenamiento permiten modificar el ámbito de una variable. Los especificadores pueden ser uno de los siguientes tipos: auto, extern, register, static y typedef.

8.1. Variables automáticas.

Las variables que se declaran dentro de una función se dice que son automáticas (auto), significando que se les asigna espacio en memoria automáticamente a la entrada de la función y se les libera el espacio tan pronto se sale de dicha función. La palabra reservada auto es opcional.

Declaración de variables automáticas	
auto int xl	es igual que int x1
auto float a,b	es igual que float a,b
auto char ch, cha1	es igual que char ch, ch1

8.2. Variables registro.

Precediendo a la declaración de una variable con la palabra reservada register, se sugiere al compilador que la variable se almacene en uno de los registros hardware del microprocesador. Para declarar una variable registro, hay que utilizar una declaración similar a:

```
register int k;
```

Una variable registro debe ser local a una función, nunca puede ser global al programa completo.

8.3. Variables externas.

Cuando una variable se declara externa, se indica al compilador que el espacio de la variable está definida en otro archivo fuente y que puede ser usada en el archivo actual. Una variable global definida en un archivo, puede ser usada en la compilación de otro archivo distinto pero sin reservar nuevo espacio en memoria, para que al ser montadas juntas, ambas compilaciones funcionen correctamente. Si no se hiciera la declaración de variable externa, entonces ocuparían posiciones de memoria distintas y al montar los dos archivos no funcionaría. Se declaran precediendo a la declaración de variable, la palabra extern.

Ejemplo. Las funciones *LeerReal* y *EscribirReal* leen y escriben respectivamente la variable real r. Esta variable r es global y no está en el archivo fuente de las funciones.

```
// archivo fuente extern1.cpp
#include <iostream>
using namespace std;
void LeerReal( )
{
    extern float f; // variable definida en otro archivo (extern2.cpp)
    cout << " introduzca el dato que se guardara en la funcion LeerReal  "; cin >> f;
}

void EscribirReal( )
{
    extern float f; // variable definida en otro archivo (extern2.cpp)
    cout << "El dato ingresado en la funcion LeerReal" << endl << "y mostrado en la funcion Es
    cin.get();
}


// archivo fuente extern2.cpp
#include <cstdlib>
#include <iostream>
using namespace std;
float f;
void LeerReal( );
void EscribirReal();
int main( int argc, char *argv[])
{
    LeerReal();
    EscribirReal();
    cin.get();
    return EXIT_SUCCESS;
}
```

8.4. Variables estáticas.

Las variables estáticas no se borran (no se pierde su valor) cuando la función termina y, en consecuencia, retienen sus valores entre llamadas a una función. Al contrario que las variables locales normales, una variable static se inicializa sólo una vez. Se declaran precediendo a la declaración de la variable con la palabra reservada *static*.

Ejemplo. Calcula las sucesivas potencias a^0, a^1, $a^2, \ldots, a^n$, usando una variable estática f.

```
#include <cstdlib>
#include <iostream>
using namespace std;
float potencia5( float a, int n)   // Las funciones potencia5 y potencia entregan el mismo res
{
    static float f = 1.0 ;
    f *= a;
    return f;
}
```

```
float potencia ( float a, int n)
{
    float f = 1.0;
    for ( int i = 1; i <= n; i++)
        f *= a;
    return f;
}

int main( int argc, char *argv[])
{
    float a;
    int n;
    cout << " valor de a ";
    cin >> a;
    do
        {
            cout << " valor de n ";
            cin >> n;
        } while (n<=0);

    for( int i = 1 ; i <= n ; i++)
        cout << a << " elevado a " << i << " = " << potencia5(a,i) <<endl;
    cin.get();
    return EXIT_SUCCESS;
}
```

9. El tipo de dato *bool*.

La mayoría de los compiladores de C++ incorporan el tipo de dato bool cuyos valores posibles son: ”verdadero”(true) y ”falso”(false) . El tipo bool proporciona la capacidad de declarar variables lógicas, que pueden almacenar los valores verdadero y falso. Si en el compilador de C++ no está disponible el tipo bool , deberá utilizar el tipo de dato int para representar el tipo de dato bool . C++ utiliza el valor entero 0 para representar falso y cualquier valor entero distinto de cero (normalmente 1) para representar verdadero. De esta forma, se pueden utilizar enteros para escribir expresiones lógicas de igual forma que se utiliza el tipo bool .
Las variables tipo bool se utilizan como indicadores o banderas de estado. El valor del indicador se inicializa (normalmente a false) antes de la entrada al bucle y se redefine (normalmente a true) cuando un suceso específico ocurre dentro del bucle.
Un bucle controlado por bandera-indicador se ejecuta hasta que se produce el suceso anticipado y se cambia el valor del indicador.

Ejemplo. Se leen repetidamente caracteres del teclado y se detiene cuando se introduce un dígito. Se define una bandera *dígito_leido* que se inicializa a false, y se cambia al valor de true cuando se lee un dígito. El bucle que resuelve el problema está controlado por la bandera *digito_leido*, y en cada iteración solicita un carácter, se lee en la variable *car*, y si es un dígito cambia el valor de la bandera. Al final del bucle se escribe el dígito leído.

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main( int argc, char *argv[])
```

Función	Prueba (test) de
int isalpha(int c)	Verdadero si es letra mayúscula o
int isdigit(int c)	Verdadero si es dígito (1, 2, . . . , 9)
int isupper(int c)	Verdadero si es letra mayúscula (A
int islower(int c)	Verdadero si es letra minúscula (a-
int isalnum(int c)	isalpha(c) — — isdigit(c).
int iscntrl(int c)	Verdadero si es carácter de contro
int isxdigit(int c)	Verdadero si es dígito hexadecimal
int isprint(int c)	Verdadero si Carácter imprimible
int isgraph(int c)	Verdadero si es carácter imprimibl
int isspace(int c)	Verdadero si c es carácter un espa
línea (\n), retorno de carro (\r), tabulación (\t) o tabulación vertical (\v).	
int ispunct(int c)	Verdadero si es carácter imprimibl
int toupper(int c)	Convierte a letra mayúscula.
int tolower(int c)	Convierte a letras minúscula.

Cuadro 1: Funciones de caracteres

```
{
    char car;
    bool digito_leido = false; // no se ha leído ningún dato
    while (!digito_leido)
        {
            cout << "Introduzca un caracter digito para salir del bucle :";
            cin >> car;
            digito_leido = (('0'<= car) && (car <= '9'));
        } // fin de while
        cout << car << " es el digito leido" << endl;
        cin.get();
        return EXIT_SUCCESS;
}
```

10. Concepto y uso de funciones de biblioteca

Todas las versiones de C++ ofrecen una biblioteca estándar de funciones que proporcionan soporte para operaciones utilizadas con más frecuencia. Las funciones estándar o predefinidas, se dividen en grupos; todas las funciones que pertenecen al mismo grupo se declaran en el mismo archivo de cabecera. Los nombres de los archivos de cabecera estándar utilizados en los programas se muestran a continuación encerrados entre corchetes tipo ángulo:

¡assert.h¿ ¡ctype.h¿ ¡errno.h¿ ¡float.h¿
¡limits.h¿ ¡locale.h¿ ¡math.h¿ ¡setjmp.h¿
¡signal.h¿ ¡stdarg.h¿ ¡stddef.h¿ ¡stdio.h¿
¡stdlib.h¿ ¡string.h¿ ¡time.h¿

11. Miscelánea de funciones

11.1. Funciones de carácter.

El archivo de cabecera ¡ctype.h¿define un grupo de funciones/macros de manipulación de caracteres. Todas las funciones de vuelven un resultado de valor verdadero (distinto de cero) o falso (cero). (Véase la Tabla 1)

Ejemplo. Realizar un bucle que itera hasta que se introduzca s o n.

```
include <cstdlib>
#include <iostream> // contiene <ctype.h>

using namespace std;

int main( int argc, char *argv[])
{
   char resp; //respuesta del usuario
   do
   {
      cout << " introduzca S = Si N = NO? ";
      cin >> resp;
      resp = toupper(resp);
   } while ( (resp != 'S') && (resp != 'N'));
   cout <<" respuesta leida " << resp;
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

11.2. Funciones numéricas

Virtualmente, cualquier operación aritmética es posible en un programa C++. Las funciones matemáticas disponibles son las siguientes: trigonométricas; logarítmicas; exponenciales; funciones matemáticas de carácter general; aleatorias. La mayoría de las funciones numéricas están en el archivo de cabecera *math.h*; las funciones de valor absoluto *abs* y *labs* están definidas en *stdlib.h*, y las funciones de división entera *div* y *ldiv* también están en *stdlib.h*.

Ejemplo. Generar 10 números aleatorios menores que 100 y visualiza el menor y el mayor.

```
include <cstdlib>
#include <iostream>
#include <time.h>

#define randomize ( srand (time(NULL) )) //Macro para definir randomize
#define random(num) ( rand()%(num)) // Macro para definir random

#define Tope 100
#define MAX( x, y)( x > y ? x : y ) // Macro para Maximo
#define MIN( x, y)( x < y ? x : y ) // Macro para Mínimo

using namespace std;

int main( int argc, char *argv[])
{
   int max, min, i;
   randomize;
   max = min = random(Tope);
   for (i = 1; i < 10; i++)
   {
      int x = random(Tope);
      min = MIN(min,x);
      max = MAX(max,x);
   }
   cout << "minimo "<< min << " maximo "<< max << endl;
```

```
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

11.3. Funciones de fecha y hora.

Los microprocesadores tiene un sistema de reloj que se utiliza principalmente para controlar el microprocesador, pero se utiliza también para calcular la fecha y la hora. El archivo de cabecera *time.h* define estructuras, macros y funciones para manipulación de fechas y horas. La fecha se guarda de acuerdo con el calendario gregoriano(mm/dd/aa). Las funciones *time* y *clock* devuelven, respectivamente, el número de segundos desde la hora base y el tiempo de CPU (Unidad Central de Proceso) empleado por el programa en curso.

11.4. Funciones de utilidad.

El lenguaje C++ incluye una serie de funciones de utilidad que se encuentran en el archivo de cabecera *stdlib.h* como las siguientes: *abs(n)* , que devuelve el valor absoluto del argumento n; *atof(cad)* que convierte los dígitos de la cadena *cad* a número real; *atoi(cad)* , *atol(cad)* que convierte los dígitos de la cadena cad a número entero y entero largo respectivamente.

11.5. Visibilidad de una función.

El ámbito de un elemento es su visibilidad desde otras partes del programa y la duración de un objeto es su tiempo de vida, lo que implica no sólo cuánto tiempo existe la variable, sino cuando se crea y cuando se hace disponible. El ámbito de un elemento en C++ depende de dónde se sitúe la definición y de los modificadores que le acompañan. Se puede decir que un elemento definido dentro de una función tiene ámbito local (alcance local), o si se define fuera de cualquier función, se dice que tiene un ámbito global.

11.6. Compilación separada.

Los programas grandes son más fáciles de gestionar si se dividen en varios archivos fuente, también llamados módulos, cada uno de los cuales puede contener una o más funciones. Estos módulos se compilan y enlazan por separado posteriormente con un enlazador, o bien con la herramienta correspondiente del entorno de programación. Cuando se tiene más de un archivo fuente, se puede referenciar una función en un archivo fuente desde una función de otro archivo fuente. Al contrario que las variables, las funciones son externas por defecto. Si desea, por razones de legibilidad, puede utilizar la palabra reservada *extern* con el prototipo de función. Se puede hacer una función visible al exterior de un archivo fuente utilizando la palabra reservada static con la cabecera de la función y la sentencia del prototipo de función. Si se escribe la palabra static antes del tipo de valor devuelto por la función, la función no será pública al enlazador , de modo que otros módulos no tendrán acceso a ella.

12. Sobrecarga de funciones (polimorfismo)

La *sobrecarga* de funciones permite escribir y utilizar múltiples funciones con el mismo nombre, pero con diferente lista de argumentos. La lista de argumentos es diferente si tiene un argumento con un tipo de dato distinto, si tiene un número diferente de argumentos, o ambos. La lista de argumentos se suele denominar signatura de la función.

Las reglas que sigue C++ para seleccionar una función sobrecargada son:

- Si existe una correspondencia exacta entre los tipos de parámetros de la función llamadora y una función sobrecargada, se utiliza dicha función.
- Si no existe una correspondencia exacta, pero sí se produce la conversión de un tipo a un tipo superior (tal como un parámetro int a long, o un float a un double) y se produce, entonces, una correspondencia, se utilizará la función seleccionada.

- Se puede producir una correspondencia de tipos, realizando conversiones forzosas de tipos (moldes-cast).
- Si una función sobrecargada se define con un número variable de parámetros (mediante el uso de puntos suspensivos (...)) se puede utilizar como una coincidencia potencial.

Ejemplo. Sobrecarga de funciones.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int Sobrecarga( int);
int Sobrecarga( int, int);
float Sobrecarga( float, float);
float Sobrecarga ( float, float, float);

int main( int argc, char *argv[])
{
   int x = 4, y = 5;
   float a = 6.0 , b = 7.0, c = 9.0;
   cout << "\n El cuadrado de " << x << " es: " << Sobrecarga(x);
   cout << "\n El producto de " << x << "por " << y << " es: " <<  Sobrecarga(x, y);
   cout << "\n La suma de " << a << "y " << b << " es: " << Sobrecarga(a, b);
   cout << "\n La suma de " << a << "y " << b << " es: " << Sobrecarga(a, b) << endl;
   system("PAUSE");
   return EXIT_SUCCESS;
}
// Sobrecarga, calcula el cuadrado de un valor entero
int Sobrecarga( int valor)
{
   return (valor * valor);
}

// Sobrecarga, multiplica dos valores enteros
int Sobrecarga( int valor1, int valor2)
{
   return(valor1 * valor2);
}

// Sobrecarga, calcula la suma de dos valores reales
float Sobrecarga( float valor1, float valor2)
{
   return (valor1 + valor2);
}

// Sobrecarga, calcula la media de tres valores reales
float Sobrecarga ( float valor1, float valor2 , float valor3)
{
   return (valor1 + valor2 + valor3)/3;
}
```

13. Plantillas de funciones

Las *plantillas de funciones* (function templates) proporcionan un mecanismo para crear una función genérica. Una función genérica es una función que puede soportar simultáneamente diferentes tipos de datos para su parámetro o parámetros. Una plantilla de función de tipo no genérico pero de argumentos genéricos tiene el siguiente formato: *template ¡class Tipo¿*
Tipo_de_funcion Func1(Tipo arg1, Tipo arg2)
{
// cuerpo de la función Func1()
}

Ejemplo. Función que retorna el menor con tipos de datos genéricos.

```
#include <cstdlib>
#include <iostream>

using namespace std;

template <class T>
T man(T a, T b)
{
   if (a < b)
      return a;
   else
      return b;
}

int main( int argc, char *argv[])
{
   int x = 4, y = 5;
   float a = 6.0 , b = 7.0;
   char c = 'C', d = 'A';
   cout << "\n El menor de " << x << " y " << y << " es: " << man(x, y);
   cout << "\n El menor de " << a << " y " << b << " es: " << man(a, b);
   cout << "\n El menor de " << c << " y " << d << " es: " << man(c, d) << endl ;
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

14. Breve introducción a la programación modular

Cuando se escribe un código en lenguaje C es deseable que si se tienen funciones que pueden servir para algún otro programa, poder disponer de ellas sin la necesidad de estar buscando en cada programa para copiarla y pegarla.
Esto representa un gran inconveniente al momento de reutilizar el código, además la extensión del código puede llegar a ser muy grande y difícil de leer.
Con C podemos solucionar estos inconvenientes creando bibliotecas de funciones que tengan alguna característica, que deba incluir en los programas que diseñe. Debemos entonces por un lado separar la biblioteca de la función principal de aplicación, guardando las funciones de biblioteca en archivos distintos.
Tendremos en consecuencia que generar dos tipos de archivos:

1. **Encabezado:** Es un archivo que se distingue por tener la extensión .h, en él se declarar los prototipos de función, constantes y valores a utilizar en el encabezado o en las funciones, tiene la siguiente forma

```
#ifndef AREAS_H_INCLUDED
#define AREAS_H_INCLUDED
... definir bibliotecas a utilizar
... definir constantes
... declarar prototipos de función
#endif // AREAS_H_INCLUDED
```

2. **Biblioteca:** Es un archivo con las funciones que corresponden a los prototipos declarados en el encabezado.

Con la directiva *#include .*^{en}*cabezado.h "*, se agrega la biblioteca a la función principal o también a otra biblioteca que quiera definir

Ejemplo. Construya una biblioteca de funciones para calcular las áreas de las figuras geométricas y luego aplíquelas en un programas.

Encabezado

```
/* Encabezado Areas.h */
#ifndef AREAS_H_INCLUDED
#define AREAS_H_INCLUDED
#include<math.h>
#define PI 3.141592654

/* Prototipos de funciones */
float rectangulo(float a, float b);

float trianguloBH(float a, float b);

float trianguloABC(float a, float b, float c);

float circulo(float r);

#endif // AREAS_H_INCLUDED
```

Biblioteca de funciones.

```
/* Biblioteca de funciones Areas.c, para el
cálculo de areas de distintas figuras geométricas */

#include "Areas.h"
//#include<math.h>
//#define PI 3.141592654

/* Rectangulo */

float rectangulo(float a, float b)
{
    return a*b;
```

```
}

/* Triangulo del que sabemos la altura */

float trianguloBH(float a, float b)
{
    return a*b/2;
}

/* triangulo del que conocemos la longitud de sus lados, se aplica la formula
de Herón  A= sqrt(s(s-a)(s-b)(s-c)), s=(a+b+c)/2 */
float trianguloABC(float a, float b, float c)
{
    float s;
    s=(a + b + c)/2;
    return sqrt(s*(s-a)*(s-b)*(s-c));
}

/* Circulo */
float circulo(float r)
{
    return PI * r * r;
}
```

Función de implementación

```
/* usoAreas.c, utiliza la biblioteca Areas */

#include<iostream>
#include<cstdlib>
#include<iomanip>
#include "Areas.h"

using namespace std;

int main(int argc, char *argv[])
{
    float areaRec, areaCirc;
    areaRec = rectangulo(1.45, 0.163);
    cout << "El area del rectangulo es: " << setprecision(4) << areaRec << endl;
    areaCirc = circulo(10);
    cout << "El area del circulo es: " << areaCirc << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

15. Problemas.

1. Modificar el programa de las áreas de las figuras geométricas, para que contemple la mayor cantidad de casos posibles

2. Cree una biblioteca de funciones que permita obtener el desplazamiento de un brazo robótico en longitud y ángulo dado:

   a) Los valores en un plano (x, y)

   b) Los valores en un espacio (x, y, z)

   Tenga en cuenta que para obtener la longitud a la que tiene que llegar el brazo robótico se debe usar la norma 2, $d_p = \sqrt{x^2 + y^2}$ o $d_e = \sqrt{x^2 + y^2 + z^2}$. El ángulo de desplazamiento para el plano será $\theta_p = \arctan(y/x)$, y en el espacio $\theta_e = \arctan(z/d_e)$ La biblioteca de funciones debe contemplar que los datos ingresados puedan ser enteros, flotantes, dobles; por lo tanto se debe utilizar sobrecarga de funciones. También realizar una función con una plantilla genérica

3. Dado el valor de un ángulo, escribir una función que muestra el valor de todas las funciones trigonométricas correspondientes al mismo.

4. Escribir un programa utilice una función para convertir coordenadas polares a rectangulares. Esto es dado el largo de un vector r y un ángulo de inclinación θ obtener las coordenadas $x = r\cos\theta$, $y = r\sin\theta$.

5. Escribir una función que decida si un número entero es capicúa. El número 24842 es capicúa. El número 134 no lo es

6. Escribir una función que sume los n primeros números impares.

7. Escribir un programa, mediante funciones, que visualice un calendario de la forma:

L	M	M	J	V	S	D
			1	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	23
27	28	29	30			

   El usuario indica únicamente el mes y el año. La fórmula que permite conocer el día de la semana correspondiente a una fecha dada es:

   a) Meses de enero o febrero $n = a + 31 * (m–1) + +d(a–1)/4–3 * ((a + 99)/100)/4$;

   b) Restantes meses $n = a + 31 * (m–1) + d–(4 * m + 23)/10 + a/4–(3 * (a/100 + 1))/4$; donde a = año, m = mes, d = día.
   Nota: $n\,\%7$ indica el día de la semana (1 = lunes, 2 = martes, etc.).

8. Escribir una función que tenga un argumento de tipo entero y que devuelva la letra P si el número es positivo, y la letra N si es cero o negativo.

9. Escribir una función lógica de dos argumentos enteros, que devuelva true si uno divide al otro y false en caso contrario.

10. Escribir una función lógica Vocal que determine si un carácter es una vocal.

11. Dado el valor de un ángulo, escribir una función que muestra el valor de todas las funciones trigonométricas correspondientes al mismo.

12. Escribir una función que decida si un número entero positivo es primo, devolviendo un valor booleano.

13. Escribir un programa utilice una función *inline* para convertir coordenadas polares a rectangulares.

14. La ley de probabilidad de que ocurra el suceso r veces de la distribución de Poisson de media m viene dado por:

$$Probabilidad(X = r) = \frac{\lambda^r}{r!}e^{-\lambda}$$

Escribir un programa que calcule mediante un menú el valor de:

a El suceso ocurra exactamente r veces.

b El suceso ocurra a lo sumo r veces.

c El suceso ocurra por lo menos r veces.