Ciencias computacionales

Propedeutico: Programación

INAOE

Contents

1 Funciones **2**

1.1 Definición de funciones 2

1.2 Prototipos 3

1.3 Funciones inline 4

1.4 Paso de parámetros 4

1.5 Recursividad 5

1.6 Ámbito de las variables 6

1.7 Sobrecarga de funciones 8

Material multimedia recomendado:

- Introducción a las funciones `https://www.youtube.com/watch?v=ZYCTqYvDEI8`
- Funciones inline `https://www.youtube.com/watch?v=-j65C9P7-6s`
- Paso de parametros `https://www.youtube.com/watch?v=EO2opL_U5rE`
- Recursividad `https://www.youtube.com/watch?v=z3f2ZqO78NY`
- Funciones sobrecargadas `https://www.youtube.com/watch?v=jbuyKCgbRXo`

1 Funciones

1.1 Definición de funciones

En el lenguaje de programación C permite dividir un programa en subprogramas. De esta forma se tiene un código más manejable y limpio. Cada uno de estos subprogramas deben realizar una tarea clara y especifica. De tal forma que al unir todos estos subprogramas se tenga la funcionalidad que el programa completo debe tener. Estos subprogramas son llamados funciones.

Una función es un conjunto de líneas de código que realizan una tarea específica y puede retornar un valor. Es posible que las funciones puedan recibir parámetros, lo cual, puede alterar su comportamiento. Las funciones son utilizadas para descomponer grandes problemas en tareas simples y para implementar operaciones que son comúnmente utilizadas durante un programa y de esta manera reducir la cantidad de código. Cuando una función es invocada se le pasa el control a la misma, una vez que esta finalizó con su tarea el control es devuelto al punto desde el cual la función fue llamada.

Todos los programas C/C++ inician su ejecución en la función main. Cualquier otra función debe ser llamada desde la función main.

Una función llama a otra por medio de su nombre (identificador) seguida de los argumentos o parámetros encerrados en paréntesis (), ver Figura 1.

NombreDeLaFuncion (ListaDeArgumentos)

Figure 1: Sintaxis para el llamado a funciones

La lista de argumentos es la forma en que las funciones se comunican unas con otras. Esta lista puede contener 0, 1, o más argumentos. Cuando se tiene más de un argumentos estos deben ir separados por coma.

Una función consta de dos partes: el encabezado y el cuerpo.

Encabezado

```
1 int Cube(int n)
2 {
3         return n*n*n;
4 }
```

Cuerpo

El encabezado, a su vez, se divide en tres partes:

- El valor que retornará la función, puede ser cualquiera de los tipos primitivos (short, int, char, long, etc.), arreglos, estructuras, objetos de clases, etc. En caso de que la función no retorne nada se utiliza la palabra reservada void. Cuando la función ha especificado un tipo de retorno, en el cuerpo de la función se debe usar la palabra reservada **return** seguida del valor a retornar. Las funciones pueden retornar solo un valor.
- El nombre de la función, no se permiten espacios en blanco (ver declaración de identificadores).
- Lista de argumentos encerrados en paréntesis y separados por comas.

1.2 Prototipos

Los prototipos de las funciones permiten declarar una función al inicio de un programa. Esta declaración es solo el encabezado de la función (valor a retornar, nombre de la función y lista de argumentos) seguida de punto y coma. El prototipo de las funciones no consta de instrucciones, y nos permite que el compilador sepa que más adelante estará la implementación de la función función.

```
1  #include <iostream>
2  int Cube(int);                          // PROTOTIPO
3  using namespace std;
4
5  void  main (){
6          int yourNumber;
7          int myNumber;                              //argumentos
8
9          yourNumber = 14;
10         myNumber = 9;
11
12         cout << "My Number = " << myNumber;
13         cout << "its cube is " << Cube (myNumber)    << endl;
14
15         cout << "Your Number = " << yourNumber;
16         cout << "its cube is " << Cube (yourNumber)  << endl;
17 }
```

Un programa típico consta de funciones prototipos, la función principal (main) y la implementación de las funciones prototipo y funciones de ayuda.

El siguiente código muestra un ejemplo de un programa en el lenguaje de programación C:

```
1  #include <iostream>
2
3  int  Square ( int ) ;   // prototypes
4  int  Cube ( int ) ;
5  using namespace std;
6
7  int  main (){
8          cout  <<  "The square of 27 is "
9                <<   Square (27)    <<    endl; // function call
10
11         cout  <<  "The cube of 27 is "
12               <<  Cube (27)      <<    endl;  // function call
13         return 0;
14 }
15
16
17 int Square ( int  n ){ // header and body
18       return   n * n;
19 }
20
21
22 int Cube ( int  n ){    // header and body
23      return  n * n * n;
24 }
```

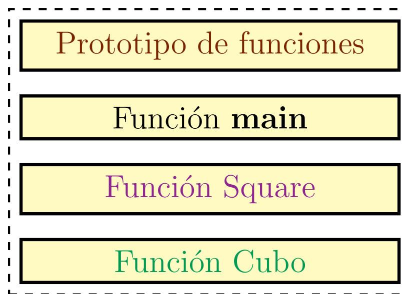

Figure 2: Programa con varias funciones y funciones prototipo

La figura 2 presenta la estructura del programa del código anterior que consta de las funciones Square y Cube, así como sus funciones prototipos.

1.3 Funciones inline

Las funciones en linea (***inline***) le proporcionan el compilador la facultad de reemplazar cualquier llamada a la función en el programa fuente por el cuerpo actual de la función. Esto quiere decir que el compilador puede tomar la iniciativa de expandir la llamada a la función con el cuerpo de la misma, pero también puede decidir no expandir la función, por ejemplo, por ser demasiase larga.

Para poder asignar el calificativo inline a una función, dicha función debe estar definida antes de ser invocada, de lo contrario el compilador no la tendrá en cuenta. Esto es por lo que las funciones inline son comúnmente definidas en ficheros de cabecera.

Declarar una función inline implica anteponer el calificativo inline al tipo retornado por la función. Por ejemplo:

```
1 inline int menor(int x, int y){
2         return (x < y) ? x : y;
3 }
```

Una de las principales características de las funciones inlie es que comunmente se obtienen tiempos de ejecución más rápidos, ya que se evitan las llamadas a las funciones, lo cual requiere almacenar registros en la pila al inicio de la función y restaurarlos al terminar la ejecución de dicha función. Por otra parte el tamaño del código puede aumentar demasiado. Por lo cual se recomienda utilizar el calificador inline cuando la función es muy pequeña, o si solo se llama desde pocos lugares.

1.4 Paso de parámetros

El paso de parámetros a una función puede realizarse de dos forma: por valor y por referencia.

Pasar los parámetros por valor significa copiar los valores actuales de las variables que se le pasan a la función en los parámetros formales de la función, con lo cual no se modifican los valores de las variables, independientemente si la función realiza cambios sobre ellas o no.

Pasar los parámetros por referencia significa que lo transferido no son los valores, sino las direcciones de memoria de las variables que contiene los valores, con lo que las variables se verán modificadas si la función realiza cambios sobres sus parámetros.

Cuando se llama a una función y en la lista de argumentos se especifican los nombres de éstos sin más, dichos argumentos son pasados por valor, excepto las matrices, que se pasan por referencia, ya que el nombre de una matriz representa la dirección de comienzo de la matriz.

Al utilizar el paso de parámetros por valor se pueden transferir constantes, variables y expresiones, y al utilizar un paso de parámetros por referencia se permite transferir las direcciones de variables de cualquier tipo, matrices he incluso direcciones de otras funciones.

En C++ para pasar una variable por referencia, se pueden utilizar dos formas:

1. Pasar la dirección de una variable a su correspondiente parámetro formal, el cual debe de ser un puntero.
2. Declarar el parámetro formal de la función como una referencia. Para ello se antepone el operador & al nombre del parámetro formal.

El siguiente código muestra un ejemplo de paso de parámetro por referencia utilizando punteros:

```
1  #include<iostream>
2  #include<stdio.h>
3
4  void permutar (int *, int *);
5
6  int main(){
7          int a = 10, b = 20;
8          permutar(&a, &b); //se pasan las direcciones de a y b
9          printf("a␣=␣%d,␣b␣=␣%b\n", a, b);
10 }
11
12 //Usando punteros
13 void permutar (int *px, int *py){
14         int z = *px;
15         *px = *py;
16         *py = z;
17 }
```

El código siguiente muestra el mismo ejemplo usando el operador de referencia en vez de punteros:

```
1  #include<iostream>
2  #include<stdio.h>
3
4  void permutar (int &, int &);
5
6  int main(){
7          int a = 10, b = 20;
8          permutar(a, b); //se pasan las direcciones de a y b
9          printf("a␣=␣%d,␣b␣=␣%b\n", a, b);
10 }
11
12 //Utilizando referencias.
13 //rx y ry son referencias a los
14 //correspondientes parámetros actuales a y b
15 void permutar (int &rx, int &ry){
16         int z = rx;
17         rx = ry;
18         ry = z;
19 }
```

En ambos ejemplos se obtienen los mismos resultados.

En el ejemplo con punteros, cualquier asignación que se haga a $*px$ afecta a la variable a. En el ejemplo con referencias, la referencia rx tiene la misma propiedad, pero sin requerir el operador de indirección $*$. La principal diferencia, es que con un puntero se puede distinguir el puntero de la variable apuntada, utilizando el operador de indirección $*$ (px describe el puntero, $*px$ describe el valor apuntado). Pero con una referencia solo nos podemos referir al dato.

1.5 Recursividad

Se considera una llamada recursiva a la llamada a una función en la cual la función llamada es la misma que la que hace el llamado. En otras palabras, la llamada recursiva ocurre cuando una función se llama

a si misma. En las llamadas recursivas necesitamos detenernos en algún momento, se debe evitar que la función se llama así misma de forma infinita.

Existen soluciones recursivas a ciertos problemas. Este tipo de soluciones deben de escribirse con cierto cuidado. Las soluciones recursivas se basan en la idea de dar un paso a la vez mas cerca de la solución, en la cual el problema puede resolverse de forma más sencilla. Se debe llegar a una etapa en la cual resolver el problema es muy sencillo, lo cuál es conocido como el caso base. Todos los algoritmos recursivos deben tener por lo menos un caso base, así como un caso general, el cual será resuelto de forma recursiva.

La forma general de una función recursiva es la siguiente.

```
1  if (condición fácil de resolver)  //Caso Base
2          sentencias de solución;
3  else                              //Caso general
4          llamada recursiva;
```

El siguiente ejemplo muestra una función recursiva que nos permite calcular la suma de los primeros n números enteros.

```
1  int   Summation (/* in */  int   n)
2  // Computes the sum of the numbers from 1 to
3  // n by adding n to the sum of the numbers
4  // from 1 to (n-1)
5  // Precondition: n is assigned && n > 0
6  // Postcondition: Return value == sum of
7  // numbers from 1 to n
8  {
9          if  (n == 1)    // Base case
10                 return  1;
11         else                     // General case
12                 return (n + Summation (n - 1));
13 }
```

El caso base se tiene cuando la función es llamada con el valor 1. En este caso, la función entra en su caso base, el cual es 'fácil' de resolver. En caso contrario, la función entra en su caso general, para el cual necesita sumar el valor n actual, y llamar a la función recursiva con un caso más pequeño (sencillo), en este caso $(n-1)$.

Los algoritmos recursivos también pueden ser escritos de forma iterativa. Sin embargo, para ciertos problemas la soluciones recursivas son soluciones más naturales y fáciles de entender. Esto ocurre muy a menudo cuando se utilizan variables estructuradas. Las soluciones iterativas utilizan ciclos, mientras que las soluciones recursivas usan una sentencia de selección (en muchos casos **if**).

1.6 Ámbito de las variables

Se le llama ámbito a la zona desde que cierto objeto es accesible. En esta sección consideraremos dos tipos de ámbito impuestos por el lenguaje C++: temporal y de acceso. El ámbito de acceso nos dice desde donde es accesible. El ámbito temporal indica el intervalo de tiempo en el que un objeto existe o es accesible.

Las variables pueden ser accedidas desde distintas partes de un programa dependiendo de donde fueron declaradas. Es decir, su ámbito de acceso y temporal dependerá del lugar en que se declaren.

Las variables declaradas dentro de un bucle, serán accesibles sólo desde el propio bucle, esto es, tendrán un ámbito local para el bucle. Esto es porque las variables se crean al iniciar el bucle y se destruyen cuando termina. Evidentemente, una variable que ha sido destruida no puede ser accedida, por lo tanto, el ámbito de acceso está limitado por el ámbito temporal.

Cada función define un ámbito local a esa función. Por lo tanto, las variables declaradas dentro de una función, y recuerda que main también es una función, sólo serán accesibles para esa función, desde el punto en que se declaran hasta el final. Esas variables son variables locales o de ámbito local de esa función.

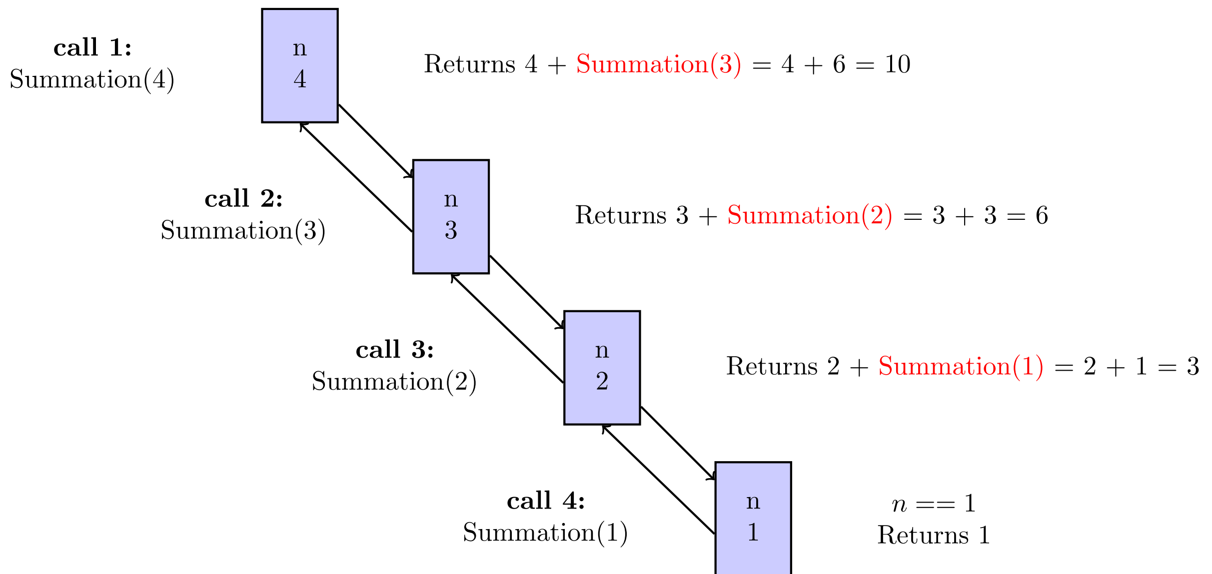

Figure 3: Llamada recursiva

Ámbito Local

El ámbito de un identificador que es declarado dentro de un bloque (esto incluye parámetros de funciones) se extiende desde el punto declaración hasta el final del bloque.

Ámbito global

El ámbito de un identificador que es declarado fuera de todos los espacios de nombres (namespaces), funciones y clases se extiende desde el punto de declaración hasta el final del archivo que contiene el código.

Figure 4: Ámbito local y global

En la figura 4 se comparan el ámbito global con el ámbito global.

Las variables globales pueden ser accedidas por medio del operador ::. Por ejemplo, en el siguiente código se declara una variable global x y una variables local a la función main también de nombre x. Ambas variables se pueden acceder de distinta forma.

```
1 int x; // Variable global
2
3 int main(){
4         int x; // Variable local que enmascara a la global
5
6         x = 10; // Accedemos a la variable local
7         ::x = 100; // Accedemos a la variable global
8         return 0;
9 }
```

Las reglas de talladas sobre el ámbito de variables en C++ son las siguientes:

1. Los nombres de las funciones tienen ámbito global.
2. El ámbito de los parámetros de una función es idéntico al ámbito de una variable local declarada en el bloque más externo del cuerpo de la función.
3. El ámbito de una variable global (o constante) se extiende desde su declaración hasta el final del archivo que la contiene.
4. El ámbito de una variable local (o constante) se extiende desde su declaración hasta el final del bloque en la cual fue declarada, incluyendo bloques anidados.
5. El ámbito de un identificador no incluye bloques anidados que contienen la declaración de un identificador con el mismo nombre que ese identificador (**los identificadores locales tienen precedencia de nombre**).

1.7 Sobrecarga de funciones

Es común dar a cada función un nombre único que la distingue de las demás. Sin embargo, en que una varias funciones ejecuten la exactamente la misma tarea, pero sobre tipos de datos distintos, o sobre distintos objetos. En estos casos, puede ser conveniente que las funciones tengas el mismo nombre. En este caso se dicen que las funciones están sobrecargadas.

Esta característica de sobrecarga de funciones hace que los programas en C++ sean mas legibles. La sobre carga de funciones se da cuando se declara una función con el mismo nombre que otra, pero con distinto numero y/o tipo de parámetros. Una función sobrecargado no puede diferir solo en el tipo de datos retornado, debe ser diferente en el tipo y/o número de parámetros. La sobre carga de funciones solo puede ocurrir dentro de su mismo ámbito.

Como ejemplo el siguiente código muestra varias definiciones de la función visualizar:

```
1 //Definiciones de la función visualizar.
2
3 void visuzlizar( char *cad = '\n');
4 void visuzlizar( long n, char car = '\n');
5 void visuzlizar( char *cad, long n, char car = '\n');
6 void visuzlizar( double n, char car = '\n');
7 void visuzlizar( char *cad, double n, char car = '\n');
```

Cuando la función visualizar sea llamada, el compilador deberá resolver cuál de las funciones con el nombre visualizar será invocada. Esto lo hace comparando los tipos de parámetros actuales con los tipos de los parámetros formales de todas las funciones llamadas visualizar. Si no encuentra una función exactamente con los mismo tipos de argumentos, realizaría las conversiones permitidas sobre los parámetros actuales buscando una función adecuada.

References

[1] Ceballos, F. J. *Programación orientada a objetos con C++*. Alfaomega Ra-Ma, 2004.

[2] Dale, N. *C++ plus data structures (4. ed.)*. Jones and Bartlett Publishers, 2007.

[3] Deitel, H. M., and Deitel, P. J. Cómo programar en C/C++ y java, 2004.