(75.41) Algoritmos y Programación II

Cátedra Lic. Andrés Juárez

2do cuatrimestre de 2018

Flujo de control y funciones

Carolina Pistillo

Índice

El siguiente apunte representa un repaso de los temas aprendidos en Algoritmos y Programación I, en adición a ciertas estructuras y protocolos característicos de C++.

1. Estructuras de Control

Las *estructuras de control* son porciones del código del programa que contienen declaraciones dentro de ellas y, dependiendo de las circunstancias, ejecutan estas declaraciones de una cierta manera. Generalmente hay dos tipos: *condicionales* y *bucles*.

1.1. Condicionales

Para que un programa cambie su comportamiento dependiendo de la entrada, debe haber una manera de probar esa entrada. Los condicionales permiten al programa verificar los valores de las variables y ejecutar (o no ejecutar) ciertas declaraciones. C++ tiene las estructuras condicionales `if` y `switch-case`.

1.1.1. Operadores

Los condicionales utilizan dos tipos de operadores especiales: *relacionales* y *lógicos*. Estos se usan para determinar si alguna condición es verdadera o falsa.

Los operadores relacionales se utilizan para probar una relación entre dos expresiones:

Operador	**Significado**
>	Mayor que
>=	Mayor o igual que
<	Menor que
<=	Menor o igual que
==	Igual a
! =	Distinto a

Funcionan de la misma manera que los operadores aritméticos (por ejemplo, a>b) pero devuelven un valor booleano `true` o `false`, que indica si la relación probada se mantiene. (Una expresión que devuelve este tipo de valor se denomina expresión booleana).

Los operadores lógicos se usan a menudo para combinar expresiones relacionales en expresiones booleanas más complicadas:

Operador	**Significado**
&&	and
||	or
!	not

Cualquier tipo de valor se puede usar en una expresión booleana debido a una peculiaridad que tiene C ++ : `false` se representa por un valor de `0` y cualquier cosa que no sea `0` es `true`. Entonces, "`¡Hola, mundo!`" es verdadero, `2` es verdadero, y cualquier variable `int` que tenga un valor distinto de cero es verdadera.

1.1.2. `if`, `if-else` y `else if`

El `if` condicional tiene la forma:

```
if(condicion)
{
    sentencia1
    sentencia2
    ...
}
```

Si solo hay una sentencia, las llaves se pueden omitir.

La forma `if-else` se usa para decidir entre dos secuencias de sentencias a las que se hace referencia como bloques:

```
if(condicion)
{
    sentenciaA1
    sentenciaA2
    ...
}

else
{
    sentenciaB1
    sentenciaB2
    ...
}
```

El `else if` se usa para decidir entre dos o más bloques según múltiples condiciones:

```
if(condicion1)
{
    sentenciaA1
    sentenciaA2
    ...
}

else if(condicion2)
{
    sentenciaB1
    sentenciaB2
    ...
}
```

1.1.3. `switch-case`

El `switch-case` es otra estructura condicional que puede o no ejecutar ciertas sentencias. Sin embargo, el bloque del `switch` tiene una sintaxis y un comportamiento peculiares:

```
switch(expresion)
{
    case constante1:
        sentenciaA1
        sentenciaA2
        ...
        break;
    case constante2:
        sentenciaB1
        sentenciaB2
        ...
        break
    ...
    default:
        sentenciaZ1
        sentenciaZ2
        ...
}
```

El `switch` evalúa la `expresion` y, si la `expresion` es igual a `constante1`, las instrucciones debajo del `case constante1:` se ejecutan hasta que se encuentra un corte. Si `expresion` no es igual a `constante1`, entonces se compara con `constante2`. Si estos son iguales, entonces las instrucciones debajo del `case constante2:` se ejecutan hasta que se encuentre un `break` (corte). Si no es igual a `constante2`, entonces el mismo proceso se repite para cada una de las constantes, sucesivamente. Si ninguna de las constantes coincide, entonces se ejecutan las instrucciones debajo del `default:`.

Debido al comportamiento peculiar de los `switch-case`, las llaves no son necesarias para los casos en los que hay más de una declaración (pero son necesarias para encerrar toda la caja del `switch-case`). Los `switch-case` generalmente tienen equivalentes `if-else`, pero a menudo pueden ser una forma más prolija de expresar el mismo comportamiento.

Aquí hay un ejemplo que usa `switch-case`:

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int x=6;
7
8      switch(x)
9      {
10         case 1:
11             cout << "x es 1\n";
12             break;
13         case 2:
14         case 3:
15             cout << "x es 2 o 3\n";
16             break;
17         default:
18             cout << "x no es 1, 2 o 3\n";
19     }
20
21     return 0;
22 }
```

Este programa imprimirá `x no es 1, 2 o 3`. Si reemplazamos la línea 6 con int x = 2; entonces el programa imprimirá `x es 2 o 3`.

1.2. Ciclos

Los condicionales ejecutan ciertas declaraciones si se cumplen ciertas condiciones; los bucles ejecutan ciertas declaraciones mientras se cumplen ciertas condiciones. C++ tiene tres tipos de bucles: `while`, `do-while` y `for`.

1.2.1. `while` y `do-while`

El ciclo `while` tiene una forma similar al `if` condicional:

```
while(condicion)
{
    sentencia1
    sentencia2
    ...
}
```

Mientras la condición se mantenga, el bloque de sentencias se ejecutará repetidamente. Si solo hay una sentencia, las llaves se pueden omitir.

El ciclo `do-while` es una variación que garantiza que el bloque de instrucciones se ejecutará al menos una vez:

```
do
{
    sentencia1
    sentencia2
    ...
}
while (condicion);
```

El bloque de instrucciones se ejecuta y luego, si la condición se cumple, el programa regresa a la parte superior del bloque. Las llaves se requieren siempre. También tenga en cuenta el punto y coma después de la condición de `while`.

1.2.2. `for`

El bucle `for` funciona como el bucle `while` pero con algunos cambios en la sintaxis:

```
for(inicializacion;condicion;incremento)
{
    sentencia1
    sentencia2
    ...
}
```

El bucle `for` está diseñado para permitir una variable de contador que se inicializa al comienzo del bucle y se incrementa (o disminuye) en cada iteración del bucle. Las llaves se pueden omitir si solo hay una declaración. Aquí hay un ejemplo:

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      for(int x = 0; x < 10; x++)
7          cout << x << "\n";
8
9      return 0;
10 }
```

Este programa imprimirá los valores del 0 al 9, cada uno en su propia línea.

Si la variable de contador ya está definida, no hay necesidad de definir una nueva en la parte de inicialización del ciclo `for`. Por lo tanto, es válido hacer lo siguiente:

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

int main()
{
    int x = 0;
    for(; x < 10; x++)
        cout << x << "\n";

    return 0;
}
```

Tenga en cuenta que aún se requiere el primer punto y coma dentro de los paréntesis `for`.

Un ciclo `for` se puede expresar como un ciclo `while` y viceversa. Recordando que un ciclo `for` tiene la forma

```
for(inicializacion;condicion;incremento)
{
    sentencia1
    sentencia2
    ...
}
```

podemos escribir un ciclo `while` equivalente como

```
inicializacion
while(condicion)
{
    sentencia1
    sentencia2
    ...
    incremento
}
```

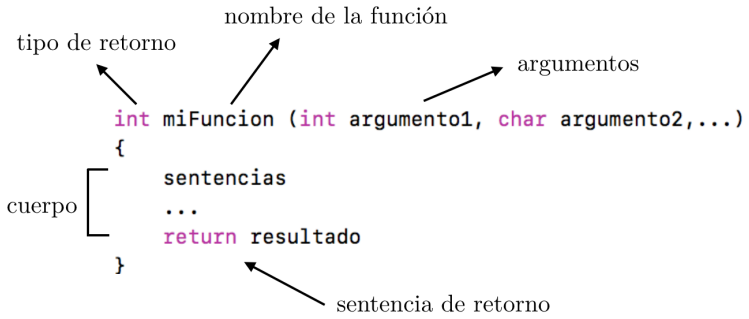
El paso de incremento puede estar técnicamente en cualquier lugar dentro del bloque de instrucciones, pero es una buena práctica ubicarlo como el último paso, particularmente si los enunciados anteriores usan el valor actual de la variable del contador.

2. Funciones

2.1. ¿Por qué definir tus propias funciones?

- **Legibilidad:** `sqrt (5)` es más claro que copiar y pegar en un algoritmo para calcular la raíz cuadrada
- **Mantenibilidad:** para cambiar el algoritmo, simplemente cambie la función (en contraste a cambiarla en cualquier lugar donde la haya usado)
- **Reutilización de código:** permite que otras personas usen los algoritmos que has implementado

2.2. Sintaxis de declaración de una función



Se puede devolver hasta un valor, que debe ser del mismo tipo que el tipo de retorno.

Si no se devuelven valores, proporcione a la función un tipo de retorno `void`. Tenga en cuenta que no puede declarar una variable de tipo `void`.

2.3. Sobrecarga de funciones

Se da cuando hay muchas funciones con el mismo nombre, pero diferentes argumentos. La función llamada es aquella cuyos argumentos coinciden con la invocación.

```
void printOnNewLine(int x)
{
    cout << "1 Entero: " << x << endl; }
void printOnNewLine(int x, int y)
{
    cout << "2 Enteros: " << x << " y " << y << endl; }
```

`printOnNewLine(3)` imprime "`1 Entero: 3`"

`printOnNewLine(2,3)` imprime " `2 Enteros: 2 y 3`"

2.4. Declarar antes de invocar

Las declaraciones de funciones deben ocurrir antes de las invocaciones

```
int foo() {
    return bar()*2; // ERROR - bar no ha sido declarada aun
}
int bar() {
    return 3;
}
```

- Solución 1: reordenar las declaraciones de funciones
- Solución 2: use un prototipo de función. Esto informa al compilador que lo implementará más tarde:

```
int bar(); ◄——— prototipo de función

int foo() {
    return bar()*2; // ok
}
int bar() {
    return 3;
}
```

Los prototipos de funciones generalmente se colocan en archivos de encabezado (*header files*) separados.

```
// myLib.h - header
// contiene prototipos
int square(int);
int cube (int);
```

```
// myLib.cpp - implementacion
#include "myLib.h"
int cube(int x)
{
    return x*square(x);
}

int square(int x)
{
    return x*x;

}
```

2.5. Recursión

Las funciones pueden llamarse a sí mismas. `fib (n) = fib (n-1) + fib (n-2)` se puede expresar fácilmente a través de una implementación recursiva:

```
int fibonacci(int n)
{
    if (n == 0 || n == 1)
    {
        return 1;

    }
    else
    {
        return fibonacci(n-2) + fibonacci(n-1); ◄——— paso recursivo
    }
}
```

2.6. Variables globales

¿Cuántas veces se llama a la función `foo()` ? Use una variable global para determinar esto. Las variables globales pueden accederse desde cualquier función.

```
int numCalls = 0; ←—— variable global
void foo()
{
    ++numCalls;
}

int main()
{
    foo(); foo(); foo();
    cout << numCalls << endl; // 3
    return 0;
}
```

2.7. Pasar por valor *vs* pasar por referencia

Hasta ahora hemos estado pasando todo por valor. Esta forma hace una copia de la variable y los cambios a la variable dentro de la función no ocurren fuera de la función.

```
// pass-by-value
void increment(int a)
{
    a = a + 1;
    cout << "a in increment " << a << endl;
}
int main()
{
    int q = 3;
    increment(q); // does nothing
    cout << "q in main " << q << endl;
    return 0;
}
```

Output:

```
a in increment 4

q in main 3
```

Si se desea modificar la variable original en lugar de hacer una copia, debe pasarse la variable *por referencia* (`int &a` en lugar de `int a`)

```
// pass-by-reference
void increment(int &a)
{
    a = a + 1;
    cout << "a in increment " << a << endl;
}
int main()
{
    int q = 3;
    increment(q); // works
    cout << "q in main " << q << endl;
    return 0;
}
```

Output:

```
a in increment 4

q in main 4
```

2.7.1. Implementacion de `swap`

Este es un ejemplo de implementación pasando por referencia.

```
void swap(int &a, int &b)
{ int t = a;
    a = b;
    b = t;
}
int main()
{
    int q = 3;
    int r = 5;
    swap(q, r);
    cout << "q " << q << endl; // q 5
    cout << "r " << r << endl; // r 3
    return 0;
}
```

2.7.2. Devolver múltiples valores

La sentencia `return` solo permite devolver 1 valor. Pasar las variables de salida por referencia quita esta limitación.

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

int divide(int numerator, int denominator, int &remainder)
{ remainder = numerator % denominator;
    return numerator / denominator;
}

int main()
{
    int num = 14;
    int den = 4;
    int rem;
    int result = divide(num, den, rem);
    cout << result << "*" << den << "+" << rem << "=" << num << endl; // 3*4+2=14
    return 0;
}
```

Referencias

- Introduction to C++ - MIT Open Course
- C++ Cómo programar - Deitel
- C++ Programming Language - Bjarne Stroustrup