Machete de C++

A. Sztrajman

29 de agosto de 2011

Índice

1. Variables

La tabla muestra las variables fundamentales más frecuentemente usadas en C++

variable	descripción	tamaño
char	contiene un caracter, o un número entero de -128 a 127	1 byte = 8 bits
bool	contiene un valor booleano, verdadero (true) o falso (false)	1 byte = 8 bits
int	contiene un número entero con signo hasta $2^{31} - 1$	4 bytes = 32 bits
float	contiene un número real con signo	4 bytes = 32 bits
double	contiene un número real con signo	8 bytes = 64 bits
long double	contiene un número real con signo	80 bits

Además podemos usar tipos de datos más complejos, como `string` y `vector`, para los cuales debemos incluir las librerías respectivas al principio del código:

```
#include <string>
#include <vector>
```

1.1. Declaración de variables

Declarar o instanciar una variable es decirle al compilador que vamos a usar un cierto tipo de dato, aclarando con qué nombre nos vamos a referir a ese dato. Por ejemplo

```
double numreal;
```

declara una variable de tipo `double` que se llama `numreal`.

Luego de declarar una variable, podemos comenzar a trabajar con ella. En el siguiente ejemplo hacemos una asignación en la declaración de la variable, y su valor final es `4.0`.

```
int main() {
        double numreal = 2.0;

        numreal = 2.0*numreal;
        return 0;
}
```

1.2. string

Una variable de tipo `string` contiene una cadena de caracteres. En el ejemplo a continuación vemos la declaración

```
#include <string>

int main() {
        string cadena = "hola mundo!";
```

```
        cout << cadena << endl;
        return 0;
}
```

1.3. vector

Una variable de tipo `vector` contiene una secuencia ordenada de variables del mismo tipo. El tipo de la variable repetida se elige en la declaración del vector, así como también la cantidad de elementos (dimensión). En el siguiente ejemplo declaramos un vector de 3 variables de tipo double, y le asignamos los valores correspondientes al primer vector canónico (1, 0, 0).

```
#include <vector>

int main() {
        vector<double> e1(3);
        e1[0] = 1.0;
        e1[1] = 0.0;
        e1[2] = 0.0;
        cout << cadena << endl;
        return 0;
}
```

2. Manejo de archivos

El manejo de archivos que vamos a necesitar para nuestros códigos es muy elemental. El primer paso consiste en declarar una variable de tipo stream, que apunta a un flujo de información. Éste puede estar asociado a un archivo del sistema, o a algún dispositivo (como en el caso de `cout` para la pantalla, y `cin` para el teclado).

2.1. Escritura en archivos

Para escribir información en un archivo, primero debemos declarar una variable de tipo `ofstream` (output file stream) y asociarla con un nombre de archivo específico (si no existe el archivo, será creado). Una vez hecha esta declaración, podemos mandar información a nuestra nueva variable de tipo `ofstream` de la misma forma en que lo hacíamos con `cout`.

En el siguiente ejemplo, declaramos una variable de tipo `ofstream` de nombre `archivosalida` que apunta a un archivo de nombre "archivo.dat". Luego, escribimos números del 1 al 9 en el archivo (cada uno en una línea distinta), y al final cerramos el archivo (es buena costumbre hacerlo antes de terminar el programa).

```
#include<iostream>
#include<fstream>

using namespace std;

int main() {
        ofstream archivosalida( "archivo.dat" );

        for(int i=1; i<=9 ; i++) {
                archivosalida << i << endl;
        }

        archivosalida.close();
        return 0;
}
```

2.2. Lectura de archivos

Para leer información de un archivo, primero debemos declarar una variable de tipo `ifstream` (input file stream) y asociarla con un nombre de archivo específico (el archivo debe existir!). Ahora podemos obtener información de nuestra nueva variable de tipo `ifstream` de la misma forma en que lo hacíamos con `cin`.

En el siguiente ejemplo, declaramos una variable de tipo `ifstream` de nombre `archivoentrada` que apunta a un archivo de nombre "archivo.dat". Luego, leemos 9 veces un entero del archivo (cada vez, depositamos lo que leimos en la variable `num`), y escribimos lo que vamos leyendo en la pantalla. Al final cerramos el archivo (es buena costumbre hacerlo antes de terminar el programa).

La línea `archivoentrada >> num` es la que lee un entero del archivo y lo pone en `num`. El programa sabe que debe leer un entero porque ese es el tipo de variable de `num`.

```
#include<iostream>
#include<fstream>

using namespace std;

int main() {
        ifstream archivoentrada( "archivo.dat" );

        for(int i=1; i<=9 ; i++) {
                int num;
                archivoentrada >> num;
                cout << num << endl;
        }

        archivoentrada.close();
        return 0;
}
```

2.3. string filename

A veces vamos a querer trabajar con un archivo cuyo nombre está guardado en una variable de tipo `string`. En el siguiente ejemplo declaramos una variable de este tipo, en la que guardamos el nombre de un archivo. Luego abrimos el archivo para escritura, y lo cerramos.

```
#include<iostream>
#include<fstream>

using namespace std;

int main() {
        string nombrearchivo;
        nombrearchivo = "archivo.dat" ;

        ofstream archivoentrada(nombrearchivo.c_str());
        archivoentrada.close();
        return 0;
}
```

Código mágico: `nombrearchivo.c_str()` transforma a la `string nombrearchivo` a un formato entendible para la definición del `ofstream archivoentrada`.

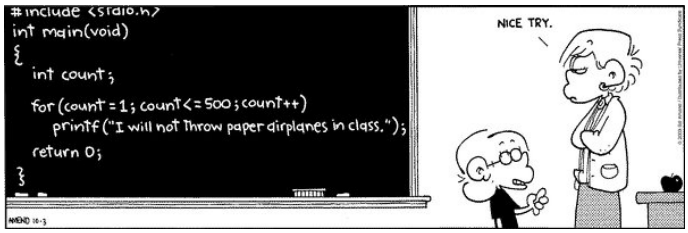
2.4. End of file

Si queremos leer un archivo completo, sin conocer su tamaño, podemos usar la propiedad booleana `eof` (end of file) de las variables `ofstream`. El siguiente código lee números reales (`double`) de un archivo de nombre "archivo.dat" hasta que llega a su final, y los va escribiendo de a uno en pantalla.

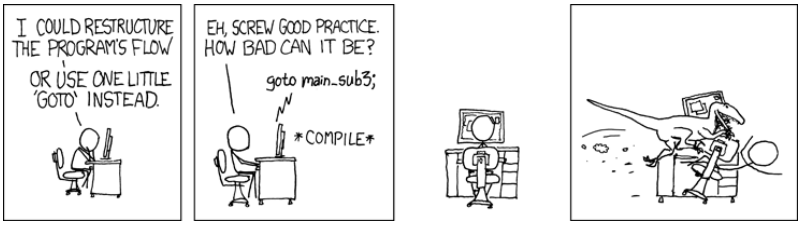
```
#include<iostream>
#include<fstream>

using namespace std;

int main() {
        ifstream archivoentrada( "archivo.dat" );

        while (archivoentrada.eof() == false) {
                double r_num;
                archivoentrada >> r_num;
                cout << r_num << endl;
        }

        archivoentrada.close();
        return 0;
}
```

3. Buenas costumbres en programación

- **Use nombres de variables declarativos**. Para las variables que tienen un análogo matemático conocido, es más claro usar, en la medida de lo posible, su nombre original (por ej. llamar `phi` a la función de onda, o `i, j, k` a los índices que recorren una matriz). Pero al resto de las variables es mejor asignarles un nombre descriptivo de su función en el programa. No ahorre letras! Las dos convenciones que se usan para nombres de variables con varias palabras son:
  - `double EstaEsUnaVariable;`
  - `double esta_es_una_variable;`
- **Indente correctamente el código**. Sea ordenado en general con su código. La indentación debe reflejar la subordinación que existe entre distintas líneas de código. Si tiene problemas para indentar, es probable que no esté entendiendo correctamente el flujo del programa y los distintos contextos que existen en él. Recuerde que algunos lenguajes modernos utilizan la indentación como parte del código (de esta forma, evitan el uso de llaves como las que usamos en C++).
- **Procure no repetir código**. Si ve tres líneas de código iguales y consecutivas, seguramente haya una forma elegante de reescribirlas, utilizando un ciclo y escribiendo una rutina general.



- **Evite por todos los medios el uso del comando** `goto`. En lenguajes más arcaicos (BASIC, assembler, Fortran 77) el uso de este comando era casi obligado. En los lenguajes modernos este comando sigue existiendo, pero su uso muy rara vez está justificado, dado que rompe con la estructura del programa permitiendo que el flujo vaya de cualquier línea a cualquier otra.



http://xkcd.com/292

- **Diseñe rutinas breves y reutilizables**. No trate de escribir todo su programa en una sola rutina o función. Mejor separe el código en varias rutinas

breves, que cumplan tareas generales. De esta forma, cuando tenga que cambiar el código para agregarle alguna funcionalidad, no necesitará reescribirlo entero.

- **Evite declarar variables globales**. No es conveniente declarar variables que sean accesibles desde todos los contextos, porque deriva en la colisión de nombres de variables. Lo correcto es que las variables lleguen a las rutinas que las necesitan a través del pasado de parámetros de las mismas.