Variables y tipos de datos: Programación en C++

Tema 4

José Ramón Paramá Gabía

Declaración de variables

- `tipo_datos nombre_variable;`
- El nombre de una variable como el de otros elementos del lenguaje es un *identificador*. Los identificadores deben cumplir una serie de normas:
  - No puede ser una palabra reservada del lenguaje.
  - No se admiten letras con tildes, diéresis o la ñ.
  - C++ distingue entre mayúsculas y minúsculas.
  - No pueden contener espacios en blanco.

Declaración de variables

- Dependiendo del punto donde se ha declarado una variable, será:
  - Variable local: declaradas dentro de bloques.
  - Parámetros: cuando son parámetros de funciones.
  - Variables globales: declaradas fuera de funciones y bloques de código.
- En la sentencia de declaración de una variable también se le puede asignar un valor inicial:

```
tipo_datos nombre variable=expre;
```

Declaración de variables

- En una misma sentencia pueden declararse varias variables, pudiéndose inicializar ninguna, alguna o todas ellas.
- `tipo_datos nombre_variable_1, …, nombre variable_n;`
- `tipo_datos nombre_variable_1=expre_1, …, nombre variable_n=expre_n;`

Tipos de datos básicos

- Son los proporcionados al programador. Se puede distinguir entre los proporcionados por el propio lenguaje C++ (*tipos básicos o primitivos)* y los que son definidos en la librería estándar.

Tipo	Tamaño típico en bits	Rango mínimo
char	8	-127 a 127
int	16 ó 32	-32,767 a 32,767
unsigned int	16 ó 32	0 a 65,535
long int	32	-2,147,483,647 a 2,147,483,647
unsigned long int	32	0 a 4,294,967,295
float	32	6 dígitos decimales
double	64	10 dígitos decimales
long double	80	10 dígitos decimales
bool	8	{ *true*, *false* }
void		

(C. Martín, A. Urquía, M. A. Rubio. *Lenguajes de programación,* Ed. Uned, 2011)

Tipos de datos básicos

declaracionVars.cpp

```cpp
// Fichero: declaracionVars.cpp
#include <iostream>

int main()
{
    int    num = 30;
    std::cout << "num:\t\t"    << num << std::endl;
    double distancia = 1.34654e3, diametro = 2.4;
    std::cout << "distancia:\t"  << distancia << std::endl;
    std::cout << "diametro:\t"  << diametro << std::endl;
    bool   b1         = true;
    bool   b2         = false;
    std::cout << "b1:\t\t"  << b1 << "\nb2:\t\t" << b2 << std::endl;
    char   c          = 'a';
    std::cout << "c:\t\t"  << c << std::endl;
    return 0;
}
```

```
num:            30
distancia:      1346.54
diametro:       2.4
b1:             1
b2:             0
c:              a
```

Estructuras

- La declaración de un tipo estructura:

```
struct nombre_tipo{
 sentencias_declaración_miembros
};
```

- Por ejemplo:

```
struct Estrella{
 char    tipo;
 double  distancia;
 int     brillo;
};
```

- La declaración de una variable:

```
Estrella e1;
```

- Se puede inicializar en la declaración:

```
Estrella e1={'a',1.75e23,4};
```

Arrays

- La sintaxis para declarar un array unidimensional es:

```
tipo nombre_variable[tamaño];
```

- Por ejemplo:

```
int num[5];
double x[3];
```

- Al declarar un array es posible inicializarlo

```
tipo nombre_variable[tamaño]={valor0, valor1, …};
```

- Como se da valor a todos los componentes del array, se está inidicando de manera implícita el tamaño. Por ese motivo cuando se inicializa el array, opcionalmente se puede omitir el tamaño:

```
tipo nombre_variable[]={valor0, valor1, …};
```

- Por ejemplo:

```
int num[]={2,3,5,2,100};
double x[]={1e3, 0.23, 2};
```

- Para acceder a los elementos se da el nombre del array y entre corchetes el índice: `num[0], num[1]`…

Arrays

- Se pueden definir arrays multidimensionales, por ejemplo:

```
int coords[3][2];
```

- Tiene 3 elementos en la primera dimensión y 2 en la segunda. Son en total 6 elementos. Sería análogo a una matriz con 3 filas y 2 columnas.
- En general la sintaxis es:

```
tipo nombre_variable [tamaño1][tamaño2]…[tamañoN];
```

- Es posible inicializarlo al declararlo, en este caso (opcionalmente) se puede omitir el tamaño de la primera dimensión, el resto se debe declarar

```
int coords[][2]={{1,2},
                 {3,4},
                 {5,6}};
```

Arrays

- En todos los casos el primer componente es 0:

```
coords[0][0]        coords[0][1]
coords[1][0]        coords[1][1]
coords[2][0]        coords[2][1]
```

- Es posible incluir array en estructuras:

```
struct Estrella{
 char    nombre[30];
 char    tipo;
 double  distancia;
 int     brillo;
 double  coordenadas[3];
};

Estrella e1={“k2323-16”, ’a’, 1.75e23, 4, 1.254e4, 1.432e6, -1.323e5};
```

Arrays

- También es posible definir arrays donde los elementos son estructuras. Por ejemplo, un array de dos componentes del tipo `Estrella`, e incluso se pueden inicializar en la declaración:

```
Estrella e[]={
{“k2323-16”, ’a’, 1.75e23, 4, 1.254e4, 1.432e6, -1.323e5},
{“k773-15”, ’c’, 8.72e22, 2, 8.234e2, 1.334e4, 1.344e7}
};
```

Tipos definidos en la librería estándar

- Están definidos en un espacio de nombres denominado *std*.
- Los nombres de los tipos de datos definidos en la librería deben escribirse precedidos por *std::*
- Para poder usar dichas facilidades es necesario incluir en el programa las correspondientes cabeceras mediante directivas *#include*

Strings

- Permite almacenar cadenas de caracteres.
- Necesita la cabecera <*string*>

```
// Fichero: holaMundoVars.cpp
#include <iostream>
#include <string>

int main()
{
    std::string frase;
    frase = "Hola mundo!";
    std::cout << frase << std::endl;
    return 0;
}
```

- En ejemplo *frase* es una variable local.
- El tipo *std::string* impone que sus variables siempre se deben inicializarse al declararlas. Si no se hace explícitamente, se asigna un valor por defecto: un string vacío al que comúnmente se denomina *empty* o *null*.

Contenedores estándar

- La librería estándar define unos tipos de datos, denominados *contenedores*, que implementan determinadas estructuras de datos como vectores, listas, pilas, colas, mapas, conjuntos, etc.
- En general se denomina *contenedor* a un objeto que almacena otros objetos, siendo posible añadir y eliminar objetos de él.
- En este capítulo sólo se presenta el uno de esos contenedores, el *vector*, que es una generalización del array.

Contenedores estándar

- Una limitación de los arrays es que al declararlos es necesario especificar su tamaño y éste ya no puede modificarse.
- Los vectores no presentan esta limitación, ya que su tamaño puede crecer en el momento que sea necesario.
- La librería tiene operadores y funciones que permiten manipular estos vectores de modo sencillo.
- Pero en este tema sólo se explica cómo declarar e inicializar estos vectores, es decir, variables del tipo `std::vector`.

Contenedores estándar

- El tipo de datos `std::vector` está declarado en la cabecera `vector,` así que si usamos vectores debemos incluir la directiva: `#include<vector>`
- La sentencia:
  `std::vector <tipo> nombre_variable;`
  declara una variable de tipo vector cuyos componentes son del tipo `tipo` y donde el contenedor se encuentra vacío.
- Como añadir un elemento al vector es costoso computacionalmente, se suele asignar un tamaño al vector cuando se declara:
  `std::vector <tipo> nombre_variable (tamaño, valor);`
- Donde *valor* indica un valor inicial de los componentes.
  `std::vector <double> v (100, 3.3);`

Flujos

- El sistema de entrada salida de C++ está basado en *flujos (streams)*.
- Tanto para E/S por consola/teclado como en el caso de ficheros.
- En la cabecera `iostream` están declarados los tipos `std::ostream` y `std::istream`.
- Cuando comienza la ejecución de un programa están disponible los flujos:
  - `std::cout`, `std::cerr`, `std::clog` del tipo `std::ostream`
  - `std::cin` del tipo `std::istream`.
- En la cabecera `fstream` están declarados los tipos `std::ifstream` y `std::ofstream` que facilitan las operaciones E/S sobre ficheros.

Punteros

- Un puntero es una variable que almacena posiciones de memoria. Esta posición es donde está almacenado un objeto, típicamente otra variable. En tal caso, se dice que la primera variable *apunta* a la segunda.
- Al declarar el puntero hay que indicar el tipo de la variable apuntada, por ejemplo:

```
int *p;
```

- En general la sintaxis es:

```
tipo_base *nombre_puntero;
```

- En una misma sentencia se pueden declarar varios punteros:

```
int *a, *b;
```

Variables en memoria dinámica

- La declaración de una variable en memoria dinámica:
  `nombre_puntero = new tipo_dato;`
- El operador *new* reserva memoria para el tipo especificado y devuelve una referencia al espacio de almacenamiento reservado.

```
// Fichero: varIntMemDin.cpp
#include <iostream>

int main()
{
    int *p;
    p = new int;
    *p = 3;
    std::cout << "En la posicion de memoria " << p
        << " esta almacenado el valor " << *p << std::endl;
    return 0;
}
```

```
En la posicion de memoria 0x804a008 esta almacenado el valor 3
```

Variables en memoria dinámica

- Es posible inicializar una variable dinámica en su declaración:

  ```
  nombre_puntero = new tipo_dato(valor);
  ```

  Por ejemplo:

  ```
  p=new int(3);
  ```

- Como se indicó la variables en memoria dinámica se deben borrar, para poder reutilizar el espacio de memoria.

  ```
  delete nombre_puntero;
  ```

  Por ejemplo:

  ```
  delete p;
  ```

- El operador delete se aplica a punteros que apuntan a variables en memoria dinámica, si no es así da un error de ejecución.

Variables en memoria dinámica

- Por ejemplo:

```
main(){
int i; //variable normal
int *p;

p=&i;
delete p; //error
}


 main(){

 int *p;

 p= new int(3);
 delete p; //no hay error
 }
```

Variables en memoria dinámica

- Es posible declarar arrays en memoria dinámica, con estas restricciones:
  - Al declarar el array debe especificarse su tamaño, mediante un entero o expresión que devuelva un entero.
  - Los arrays en memoria dinámica no se pueden inicializar en su declaración.
- La sintaxis es como sigue:

```
nombre_puntero = new tipo_dato [tamaño];
```

- Para liberar el espacio:

```
delete [] nombre_puntero;
```

- La *zona de almacenamiento dinámico* tiene un tamaño limitado, así que si se pide demasiada memoria puede que no se pueda atender la petición, en cuyo caso se debe realizar acciones alternativas.

Ámbito y visibilidad de las variables

- Un conjunto de sentencias escrito entre llaves ({}) constituye un bloque de código.
- Dentro de un bloque de código se pueden definir variables locales y variables en memoria dinámica.
- Las variables locales son eliminadas cuando la ejecución abandona el bloque.
- Las variables en memoria dinámica existen hasta se que eliminan con *delete* o hasta que finaliza la ejecución del programa.
- Ya explicamos que cuando dos variables locales tienen el mismo nombre y su ámbito se solapa, una se oculta.

Ámbito y visibilidad de las variables

- Cuando una variable global tiene el mismo nombre que una variable local, se usa el *operador ámbito* (::) para diferenciar entre ambas.
- La variable global se referencia anteponiendo el operador de ámbito a su nombre, y la local, por su nombre exclusivamente.
- Por ejemplo, *::x* (la global) y *x* (la local)