1. Repaso sobre ámbitos de una variable

Como se destacó en la clase anterior, una variable es solo visible y existe únicamente dentro de su ámbito. Además, ésta existe mientras el ámbito esté activo y desaparecen al cerrar el mismo.

Programa 1: Variables locales y globales

```
#include <iostream>
using namespace std;
// * variable global *
// Se destruye al finalizar la ejecución del programa:
int variable_G = 100;

int main() { // Ámbito local de main()
  int variable_A = 0; // se crea la variable_A

  if(variable_A == 0) { // Ámbito del if
    int variable_B = 1; // se crea la variabla_B

    cout << variable_A << ", " << variable_B
         << ", " << variable_G << endl;
  } // se "destruye" la variable_B al salir del if

} // se destruye la variable_A
```

En la porción de código anterior podemos observar que dentro del ámbito local de `main()` se crea y puede acceder a la `variable_A`, dentro del mismo se crea el ámbito de una sentencia de control `if` en la *linea 11* donde se instancia la variable `variable_B`. Como este último está dentro del ámbito del `main()` puede acceder tanto a la `variable_A` como a la `variable_B`, pero esta última es solo accesible desde dentro del if y deja de existir en la *línea 15* finaliza el bloque. La `variable_G`, al ser declarada fuera de todo ámbito se considera global y accesible desde cualquier parte del archivo.

Este concepto es importante para "ahorrar" memoria al sistema operativo, como las variables globales existen hasta que se termine el programa es deseable evitarlas y "particionar" nuestro programa en funciones que posean sus propias variables locales que utilicen la memoria mientras se ejecutan y la liberen al finalizar su ejecución.

Programa 2: Localidad en funciones

```
#include <iostream>
#include <cstdlib>
using namespace std;
int pedir_entero(string msj);
int sumar(int x, int y);

int main() {
  int a = pedir_entero("Ingrese un entero: "); // se crea 'a'
  int b = pedir_entero("Ingrese un entero: "); // se crea 'b'
  cout << a << "+" << b << "=" << sumar(a,b);
} // se destruyen 'a' y 'b'

```


Matías S. Ávalos© CC BY-NC-SA 4.0

```
13 int pedir_entero(string msj) { // se crea 'msj'
14   int n;            // se crea 'n'
15   string entrada; // se crea 'entrada'
16   do {
17     cout << msj;
18     getline(cin, entrada);
19     n = atoi(entrada.c_str());
20   } while(entrada != "0" and n == 0);
21   return n; // se devuelve el valor de 'n'
22 } // se destruyen 'n', 'entrada' y 'msj'
23
24 int sumar(int x, int y) { // se crea 'x' e 'y'
25   return x + y; // se retorna el resultado de 'x'+'y'
26 } // se destruyen 'x' e 'y'
```

1.1. Error común sobre pasaje de variables

Como ya se ha explicitado, los parámetros de una función son declaraciones de variables, es decir que son variables nuevas que toman el valor que se pase como argumento. Por ejemplo, en el Programa 2, los valores adoptados por las variables **a** y **b** son copiados en las variables locales **x** e **y** de la función **sumar()**. A estos pasajes se los denomina *por valor*, porque justamente es el *valor* el adoptado por el parámetro.

Por lo tanto en el siguiente programa veremos que la variable local de **main()** no es afectada por la función **agregar_cero()**:

Programa 3: Error común

```
1 #include <iostream>
2 using namespace std;
3
4 void agregar_cero(int n);
5
6 int main() {
7   int x = 10; // se crea la variable 'x'
8   cout<<"Antes de llamar a agregar_cero() -> x = "<<x<<endl;
9   agregar_cero(x); // se copia el valor de 'x' en 'n'
10  cout<<"Después de llamar a agregar_cero() -> x = "<<x<<endl;
11 } // se detruye la variable 'x'
12
13 void agregar_cero(int n) { // se crea la variable 'n'
14   n = n*10; // se le agrega un 0 al valor de la variable 'n'
15 } // se destruye la variable 'n'
```

Salida del programa:

```
Antes de llamar a agregar_cero() -> x = 10
Después de llamar a agregar_cero() -> x = 10
```

Al compilar y ejecutar este programa veremos dos veces el valor '10' por pantalla, ya que lo ocurrido es que el argumento **x** le *pasa su valor* al parámetro **n** de la función

Matías S. Ávalos© CC BY NC SA 4.0

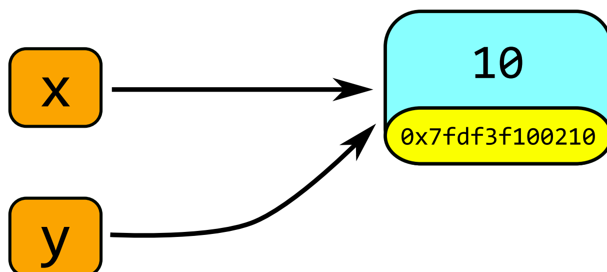
agregar_cero() y ese valor "copiado" es al que se le agrega un '0' al multiplicarlo por 10, pero nada ocurre con **x** que es una variable local de **main()**.

En linea generales, se puede decir que las variables locales solo pueden ser modificadas dentro de su ámbito.

2. Referencias

Si bien, la afirmación del último párrafo del apartado anterior es cierta. En C++ existe (como en muchos otros lenguajes) un modificador para el tipo de variable conocido como operador de referencia **&**. Esto no crea una nueva variable en memoria, sino que crea una referencia (alias) a la misma variable.

Es decir, que al mismo espacio de memoria podemos nombrarlo en dos variables de maneras diferentes y es posible modificarlo utilizando tanto una como la otra:



Programa 4: Referencia a una variable

```
#include <iostream>
using namespace std;
int main() {
  int x = 10; // se crea la variable 'x'
  int &y = x; // se crea la REFERENCIA 'y' a la variable 'x'

  // se incrementa el valor de 'x' a través de 'y':
  y++;

  cout << "x = " << x << " -> " << &x << endl;
  cout << "y = " << y << " -> " << &y << endl;
}
```

Salida del programa:

```
x = 11 -> 0x7fdf3f100210
y = 11 -> 0x7fdf3f100210
```

La variable **'y'** es una referencia, lo podemos saber por el carácter **&** antepuesto a su nombre. Eso quiere decir que si se modifica **'x'** o **'y'** es el mismo espacio de memoria el que se ve afectado. Además, podemos observar que si utilizamos el operador de referencia (**'&'**) en variables que ya existen, podemos obtener la dirección de memoria donde se encuentran las mismas. Como se observa, tanto **x** como **y** tienen la misma dirección

Matías S. Ávalos© CC BY NC SA 4.0

de memoria (son la misma variable). Es poco común utilizar en el mismo ámbito una referencia como se muestra. La potencialidad de este recurso está en la utilización de **parámetros por referencia**.

Un error común podría ser el intentar asignar a una referencia un valor literal o constante. A las referencias sólo se le puede asignar otras variables:

```
int &x = 10; // ERROR DE COMPILACIÓN
// (...)
int x = 10;
int &y = x; // Correcto, 'y' refiere a 'x'
```

2.1. Parámetros por referencia

Como se expuso, una referencia es una variable que "apunta" o "refiere" al mismo espacio de memoria que otra variable ya existente. Pero esta a su vez, no deja de ser una variable, y tiene su propia localidad. Por lo tanto puede ser utilizada como parámetro para una función.

Programa 5: Pasaje por referencia

```
1  #include <iostream>
2  using namespace std;
3
4  void agregar_cero(int &n);
5
6  int main() {
7    int x = 10; // se crea la variable 'x'
8    cout<<"Antes de llamar a agregar_cero() -> x = "<<x<<endl;
9    agregar_cero(x); // 'x' es referida por 'n' !!!
10   cout<<"Después de llamar a agregar_cero() -> x = "<<x<<endl;
11 } // se detruye la variable 'x'
12
13 void agregar_cero(int &n) { // se crea la referencia 'n'
14   n = n*10; //Se le agrega un 0 a la variable referida por 'n'
15 } // se destruye la referencia 'n'
```

Salida del programa:

```
Antes de llamar a agregar_cero() -> x = 10
Después de llamar a agregar_cero() -> x = 100
```

En este caso, si veremos por pantalla un 10 y luego un 100, ya que si bien, la variable `'n'` es local de `agregar_cero()`, ésta es una referencia, por lo tanto se modificará también `'x'`, aunque esté fuera de su ámbito.

Esta es una práctica peligrosa, en el sentido que el programador no espera que una variable local sea modificada por una función externa. Sin embargo, a veces es necesaria para solucionar problemas puntuales. Por ejemplo, cuando se requiera pasar una *variable muy pesada*, es preferible *pasar* la referencia a **copiar todo el contenido** de la misma por cuestiones de optimización. Y para otros problemas de diseño de software, pero siempre siendo cuidadosos con este recurso.

Matías S. Ávalos© CC BY NC SA 4.0

3. Variables estáticas

Supongamos que sea necesario para un algoritmo cambiar su comportamiento cada vez que es invocado. Es decir que la función se comporte ligeramente diferente según las veces o cuando es invocada por un parámetro interno de la misma (por ejemplo que el resultado anterior sea necesario). Una manera de resolver esto sería a través de variables globales:

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

int par_actual = 0;  // variable global
int siguiente_par(); // prototipo de la función

int main() {
  // Se muestran los primero 10 números pares:
  for(int i=0; i < 10; i++) {
    cout << siguiente_par() << ", ";
  }
  cout << '\n';
  // Se muestran los siguientes 10 números pares:
  for(int i=0; i < 10; i++) {
    cout << siguiente_par() << ", ";
  }
  cout << '\n';
}

int siguiente_par() {
  int valor_actual = par_actual;
  par_actual += 2;
  return valor_actual;
}
```

Si bien este recurso es válido y resulta efectivo, es una mala práctica declarar variables globales que solo se utilizarán en un ámbito específico de una función. Pero si fueran declaradas dentro de la función, éstas se destruirían y se perderían los valores de referencia. Es decir que este algoritmo no funcionaría. Para solucionar este dilema existe un modificador para las variables denominado `static` que le otorga a variables locales la persistencia de una variable global, con la diferencia de que solo son accesibles dentro de su ámbito.

Programa 6: Uso de variable estáticas

```
#include <iostream>
using namespace std;
int siguiente_par();

int main() {
  // Se muestran los primero 10 números pares:
  for(int i=0; i < 10; i++) {
    cout << siguiente_par() << ", ";
  }
  cout << '\n';
```


Matías S. Ávalos© CC BY NC SA 4.0

```
11    // Se muestran los siguientes 10 números pares:
12    for(int i=0; i < 10; i++) {
13      cout << siguiente_par() << ", ";
14    }
15    cout << '\n';
16  }
17  int siguiente_par() {
18    static int par_actual = 0;      // se crea 'par_actual'
19    int valor_actual = par_actual; // se crea 'valor_actual'
20    par_actual += 2;
21    return valor_actual;
22  } // se destruye únicamente 'valor_actual'
```

Algo que se debe tener en cuenta que **no es posible dejar variables estáticas sin inicializar**, es decir que deben tener un valor al momento de ser declaradas. A su vez, la *línea 18* de creación e inicialización de la variable estática sólo se ejecuta la primera vez que es invocada la función `siguiente_par()`, es decir que luego de la primera invocación, quedará almacenado el último valor asignado por la *línea 20*.

Esto podría utilizarse si se necesitara que un algoritmo se comporte de alguna forma diferente cada tantas veces o la primera vez que se invoca:

Programa 7: Función que cambia su comportamiento luego de la primera vez

```
1  void aplicar_veneno(int &vida) {
2    static bool es_primera_vez = true;
3    // La primera vez no quita vida...
4    if(es_primera_vez) {
5       es_primera_vez = false;
6       return;
7    }
8    // Luego resta de 1 a 3 puntos de vida
9    random_device rand;
10   vida -= (1+rand()%3);
11 }
```

En esta función, la primera vez que se ejecuta la variable *booleana* `es_primera_vez` es estática en la función. La comprobación de la *línea 4* solo dará verdadera la primera vez que esta función sea invocada, ya que en la *línea 5* esta toma el valor falso y ese bloque no volver a ejecutarse. El código a ejecutarse la segunda vez y todas las demás veces subsiguientes son las *líneas 9 y 10*.

Matías S. Ávalos© CC BY NC SA 4.0