1. Leguanjes fuertemente tipados

En los lenguajes de programación donde es obligatorio declarar los tipos de datos de sus variables y, además, no pueden ser modificados, se los denomina **Fuertemente Tipados**. Es decir, que una vez que se declara la variable de un cierto tipo (`int`, `float`, `string`, etc.) no puede cambiarse para almacenar otro tipo de dato. En otros lenguajes de programación, las variables tiene *tipado dinámico*, les permite contener, por ejemplo, un número y luego un texto:

```
1 # Declaración e inicialización de variables en python:
2 a, b  = 5, "Programación"
3 # Se muestra el valor de a y b en pantalla:
4 print(f"{a} {b}")
5 a = "Taller de"
6 print(f"{a} {b}")
```

Como vemos, en lenguajes más flexibles se pueden declarar dos variables e inicializarlas con valores de diferentes tipos de datos, como se observa en la *línea 2* (entero y *string*). Además la variable `'a'` que inicialmente tiene almacenado el entero 5, en la *línea 5* cambia este valor numérico por un *string*, sin dar ningún tipo de error.

Este pequeño programa de `Python` produce la siguiente salida por la consola:

```
5 Programación
Taller de Programación
```

El hecho de que un lenguaje permita este tipo de asignaciones puede llevar a errores de programación en *tiempo de ejecución*, es decir, si el lenguaje no logra convertir la variable de un tipo a otro, esto producirá que el programa termine abruptamente o se "cuelgue". En cambio los lenguajes de *tipado estático* son más seguros en este sentido.

2. Reinterpretación de datos (*Typecasting*)

Como se indicó en encuentros anteriores, se puede almacenar en una variable del tipo entero un valor originalmente decimal (`float` o `double`), lo cual causa una perdida de información (el truncamiento de la parte decimal):

```
int i = 4.5;  // i <= 4
```

Lo que ocurre aquí es lo que se conoce como una conversión implícita de tipos o casteo (*casting*) implícito. El valor decimal 4,5 es transformado a entero. Esto puede *explicitarse* de la siguientes maneras:

```
// a. modo funcional:
int f = int(4.5);  // i <= 4
// b. modo tradicional:
int t = (int)4.5;  // i <= 4
```

En el fragmento de código anterior se exponen dos maneras diferentes de hacer un casteo explícito en C++, el primer modo es funcional, en el cual se especifica el tipo de dato al que se desea *transformar* y se cubre entre paréntesis el valor. Y luego el modo tradicional *heredado* del lenguaje C, en el cual sólo existe esta posibilidad, donde se antepone a la expresión a transformar el tipo de dato a reinterpretar entre paréntesis. Para evitar confusiones se alentará el modo funcional donde queda de forma más clara qué es lo que se está *casteando*.

Utilizar esta técnica podría ahorra la declaración de variables intermedias innecesarias, observe el siguiente ejemplo de redondeo:

Programa 1: Redondeo con casteo

```
1 #include <iostream>
2 using namespace std;
3 int main() {
4   float pi = 3.14159;
5   // Redondeo:
6   float pi_redondeado = int(pi*10000+.5)/10000.0;
7   cout << "Pi con 5 decimales: " << pi << endl;
8   cout << "Pi con 4 decimales: " << pi_redondeado << endl;
9 }
```

La técnica es la misma, pero en vez de declarar una variable para truncar el número con la cantidad de decimales deseados, simplemente aclaramos que queremos ese resultado como entero (*linea 6*). De este modo se abrevia el código y queda más simple, incluso ganando claridad sobre la operación.

3. Modificadores y alias

3.1. Codificación y peso

En las computadoras los datos se almacenan en la memoria como simples unos y ceros, es decir que si se pudiera leer como la página de un libro veríamos renglones llenos de esto únicos 2 dígitos. Entonces, ¿cómo se interpreta o diferencia un dato entero de uno flotante o un texto? Esto se logra gracias a diferentes codificaciones estándar como el Complemento A2, IEEE754, ASCII, Unicode, etc.[1] Por eso es importante saber exactamente el tipo de dato que se está manipulando y la longitud en bits del mismo.

Si se quisiera conocer la longitud en bytes (grupo de 8 bits) de cada tipo de dato, esto podríamos hacerlo con el comando **sizeof**, el cual puede utilizarse sobre una variable, constante o tipo de dato:

```
cout<<"Un char ocupa "<<sizeof(char)<<" bytes"<<endl;
cout<<"Un int ocupa "<< sizeof(int)<<" bytes"<<endl;
cout<<"Un float ocupa "<<sizeof(float)<<" bytes"<<endl;
cout<<"Un double ocupa "<<sizeof(double)<<" bytes"<<endl;
cout<<"Un bool ocupa "<<sizeof(bool)<<" bytes" << endl;
```

[1] Esta temática se aborda en otra materia (EFSI I).

Como los *strings* son variables, su peso dependerá de la longitud de los mismos, recordando que cada carácter es un `char`:

```
cout<<"La cadena 'texto' ocupa"<<sizeof("texto")<<" bytes"<<endl;
cout<<"La cadena 'programacion' ocupa"<<sizeof("programacion")
    <<" bytes"<<endl;
```

3.2. Declaración de Constantes y modificadores

3.2.1. Punto flotante

Se observó que un `float` ocupa 4 bytes (32 bits) y un `double` ocupa 8 byes ya que están codificados con el estándar IEEE754 simple y doble precisión respectivamente. En el Programa 1 hay otro casteo implícito que pasa desapercibido en la *linea 4*. Analicemos el siguiente caso donde se hace un `sizeof` de una constante decimal:

```
cout<<"La constante 4.5 ocupa "<<sizeof(4.5)<<" bytes"<<endl;
```

Al ejecutar esa linea de código observaremos la siguiente salida:

```
La constante 4.5 ocupa 8 bytes
```

Eso quiere decir que cuando se encuentra un número constante decimal está codificado como `double`, por lo tanto en el Programa 1 se hace un casteo implícito de `double` a `float`. Si se desea especificar que la constante sea un `float`, debemos colocar al final la letra 'f'(mayúscula o minúscula indiferentemente).

```
cout << "La constante 4.5f ocupa " << sizeof(4.5f) << " bytes" <<
    endl;
```

Produciendo la siguiente salida:

```
La constante 4.5f ocupa 4 bytes
```

Eso quiere decir que en el Programa 1 lo ideal sería especificar en la *linea 4*:

```
float pi = 3.14159f;
```

3.2.2. Enteros

Los enteros declarados como `int` en `C/C++` están codificados en CA2, y dependiendo el compilador pueden ocupar 2, 4 ú 8 bytes. En gran parte de los sistemas modernos el peso más común es 4 bytes (32 bits), pero ante la duda siempre puede ejecutar un `sizeof(int)`.

Si lo anteriormente mencionado es verdad, eso quiere decir que el rango de un `int` es de −2.147.483.648 a +2.147.483.647, lo cual podría no ser óptimo desde el punto de vista de la eficiencia. Por ejemplo, si se desea llevar el control de un *stock* hogareño de productos de almacén, sería poco común tener hasta 2.147.483.647 paquetes de fideos.

Es por ello que existen modificadores que podrían acortar o alargar el rango. Así, pues, existe:

```
short int a;     // 2 bytes (16 bits) -32.768 a +32.767
long int b;      // 4 u 8 bytes (32 o 64 bits)
long long int c;// 8 bytes (64 bits)
```

El caso del `long int` así como el `int` es un valor que depende del copilador que se utilice para codificar, y al momento de declarar una constante ocurre lo mismo que con los `float` y `double`:

```
short int a = 4; // casteo implícito de int a short
int b = 4;
long int c = 4L;
long long int d = 4LL;
```

Además en algunos casos no se hace necesario tener el rango de negativos y positivos. Por eso, es posible indicarle al compilador que queremos utiliza el entero codificado en **Binario Natural** es decir, aprovechar la cantidad de bits para representar del 0 en adelante. Esto lo hacemos anteponiendo la palabra reservada `unsigned`:

```
unsigned short int a;    // 0 a 65.535
unsigned int b;          // 0 a 4.294.967.295
unsigned long int c;     // 0 a 18.446.744.073.709.551.615
```

También podemos especificar las constantes de la siguiente manera:

```
unsigned short int a = 4U; // casteo implícito de int a short
unsigned int b = 4U;
unsigned long int c = 4UL;
unsigned long long int d = 4ULL;
```

En algunos sistemas es crítico conocer y optimizar el uso de la memoria, y depender del compilador para que decida cuanto ocupa un `int` o un `long int` no sería la mejor opción. Por lo tanto se adoptó el uso de los siguiente *alias* que especifican exactamente cuanto ocupan las variables:

```
int8_t a1;  // entero de 1 byte (-128 a +127)
uint8_t a2; // entero sin signo 1 byte (0 a 255)
int16_t b1; // entero de 2 bytes (-32.768 a +32.767)
uint16_t b2;// entero sin signo 2 bytes (0 a 65.535)
int32_t c1; // entero de 4 bytes
uint32_t c2;// entero sin signo 4 bytes
int64_t d1; // entero 8 bytes
uint64_t d2;// entero sin signo 8 bytes
```

Aquí ya no hay confusiones posibles, ya que se especifica explícitamente la cantidad de bits que ocupa cada una de estas variables.

3.2.3. Constantes en binario, hexadecimal y octal

Para facilitar al programador escribir valores en otras bases el lenguaje admite que escribamos constantes en binario, hexadecimal y octal que poseen un relación de fácil conversión entre sí[2].

Por ejemplo, si se desea escribir valores representados en binario, debemos anteponer los caracteres `0b` antes de escribir la constante:

```
int16_t a = 0b1000000000000000; // constante binaria
cout << "El binario 1000000000000000 interpretado en CA2 de 16
   bits es: " << a << endl;
```

Esto producirá la siguiente salida en la consola:

```
El binario 1000000000000000 es: 32768
```

El mismo ejemplo podría escribirse de manera más sintética si en vez de expresar el número en binario lo expresáramos en hexadecimal anteponiendo `0x` de la siguiente manera:

```
int16_t a = 0x8000; // constante hexadecimal
cout << "El hexa 8000 interpretado en CA2 de 16 bits es: " << a
   << endl;
```

De forma menos frecuente, se podría expresar en octal anteponiendo simplemente un `0` a la constante:

```
int16_t a = 0100000; // constante octal
cout << "El ocatal 100000 interpretado en CA2 de 16 bits es: " <<
    a << endl;
```

3.2.4. Caracteres

Como hemos visto los caracteres constantes son expresados con comillas simples. Estos están codificados en ASCII estándar. Existen algunos caracteres de control (no imprimibles) que son de utilidad para formatear la salida por la pantalla aquí se enumeraran algunos de ellos:

- `\n`: **Nueva Linea**. Este carácter posiciona al cursor en la linea inferior inmediata, produciendo que los próximos caracteres se escriban debajo. Comportamiento similar a `endl`
- `\r`: **Retorno de carro**. Este carácter vuelve el cursor al principio de la linea sobreescribiendo cualquiera de los caracteres escritos anteriormente en esas posiciones.
- `\t`: **Tabulador**. Este carácter deja un espacio horizontal definido por el sistema operativo. Se utiliza para encolumnar valores.

[2]Esta temática se aborda en otra materia (EFSI I).

- \b: ***Backspace***. Este carácter produce que el cursor vuelva a la posición anterior inmediata. De esta forma se puede sobre-escribir el carácter anterior.

Nótese que estos caracteres se escriben anteponiendo el carácter \, conocido como *carácter de escape*. Éste indica que lo siguiente que se escribirá es uno de esto caracteres especiales mencionados o un carácter imprimible que no podría hacerse de otra manera a causa de la sintaxis del lenguaje, como por ejemplo las comillas dobles ("). Las comillas dobles se utilizan para delimitar los *strings*, es por ello que si quisiéramos mostrar en pantalla este carácter debemos recurrir a la \..

```
cout << "Esto es \"encomillado doble\".\nEsto es 'encomillado
   simple'." << endl;
```

En el fragmento de código anterior produce la salida:

```
Esto es "encomillado doble".
Esto es 'encomillado simple'.
```

Obsérvese que las contrabarras no son parte del texto y que entre las dos oraciones hay un "enter" impuesto por el \n entre el primer '.' (punto) y la 'E' del segundo "Esto".

Analice el siguiente ejemplo de uso de algunos caracteres de control y no imprimibles:

Programa 2: Uso de caracteres de control

```
1  #include <iostream>
2  using namespace std;
3  int main() {
4    cout << "Los números del 0 al 10:" << endl;
5    for(int i=0; i < 11; i++) {
6      // Muestra valor de i separado por coma y una tabulación
7      cout << i << ",\t";
8    }
9    // reemplazamos el último ",\t" por "."
10   cout << "\b\b." << endl;
11 }
```

El ejemplo anterior produce la siguiente salida:

```
0,  1,  2,  3,  4,  5,  6,  7,  8,  9,  10.
```

Si luego de salir del bucle no se volviera el cursor 2 lugares para atrás y se imprimiera el '.' la salida hubiera quedado:

```
0,  1,  2,  3,  4,  5,  6,  7,  8,  9,  10,
```

Por último podemos aclarar que los caracteres pueden escribirse también en hexadecimal utilizando el carácter de escape seguido de una x (\xNN) u octal utilizando el carácter de escape y el número correspondiente (\NNN) recordando que escribiremos el valor numérico y este luego será interpretado por el sistema operativo dependiendo de la codificación correspondiente:

```
char una_L = '\x4C';
char una_o = '\157';
cout << una_L << una_o << endl; // Se imprime "Lo"
```

3.2.5. Strings

Por último se abarcarán los *strings* constantes y utilizar la codificación estándar de sistemas Unicode con UTF-8[3]. Por ejemplo, se pueden escribir cadenas de caracteres de la siguiente manera:

```
string cadena1 = "Esto " "es un " "texto.";
```

Lo cual equivale a:

```
string cadena1 = "Esto es un texto.";
```

Uno de los principales motivos para escribir *strings* de esta forma es la clarificación de programas que poseen grandes textos o párrafos de varios renglones, ya que no estamos obligados a que se encuentren en una misma linea.

```
cout <<
  "\n*** MENÚ PRINCIPAL ***\n\n"
  "\t1. Opción A\n"
  "\t2. Opción B\n"
  "\t3. Opción C\n"
  "\t4. Opción D\n"
  "\nIngrese una opción: ";
```

La única sentencia que hay en este fragmento de código produce la siguiente salida por la pantalla:

```
*** MENÚ PRINCIPAL ***

    1. Opción A
    2. Opción B
    3. Opción C
    4. Opción D

Ingrese una opción: _
```

Esto ayuda al los programadores a tener una mejor idea de cómo se ve el *string* en la pantalla, ya que sería engorroso verlo en una sola linea y es más eficiente que hacer un `cout` por cada linea.

En Sistemas Operativo modernos que son *Tipo-UNIX*, por ejemplo, GNU/Linux y MacOS, utilizan para la codificación de los textos por *default* el Unicode, específicamente UTF-8. Sabiendo esto, si se realiza un software para este tipo de sistemas, es posible imprimir cualquier carácter del código correspondiente a éstos teniendo en cuenta que tienen longitudes variables. Es decir, el código ASCII estándar es soportado (7 bits), pero

[3]Esta temática se aborda en otra materia (EFSI I).

luego del carácter 127 los caracteres ocupan 2 bytes, y luego según la codificación pueden ocupar hasta 6 bytes. Sabiendo esto, y conociendo el código UTF-8 del carácter deseado se puede imprimir cualquiera de los soportados (incluyendo emojis).

```
cout << "\xC2\xA2" << endl;     // Centavos U+00A2 (2 bytes)
cout << "\xE2\x82\xAC" << endl; // Euros U+20AC (3 bytes)
```

El fragmento de programa anterior muestra por pantalla los caracteres '¢' y '€' respectivamente.

3.3. Valores Constantes

Hay software que tiene o necesita valores fijos (cantidad de piezas, cuadros en un tablero, movimientos en una partida, etc.). Es deseable evitar que estos números aparezcan como literales en medio del código donde no es claro de donde vienen o cuál es su función real. A éstos se los denomina **números mágicos** (*magic numbers*) y es deseable que no estén presentes en el código.

```
if(cantClientes < 500) { // magic number...¿qué significa?
//...
}
```

Para ello existen dos mecanismos que se comportan de maneras similares. Uno es la utilización de etiquetas con el comando de preprocesador `#define` seguido de un identificador que por convención se escribe en **mayúsculas** seguido del valor a ser reemplazado:

```
#define MAX_CLIENTES 500 // etiqueta
// ...
if(cantClientes < MAX_CLIENTES) {
// ...
}
```

Si bien esta forma es clásica de C/C++, se aconseja utilizar el modificador `const`, en este caso son variables que deben ser inicializadas con un valor y este no puede ser modificado. Ya que la etiqueta debe ser declarada antes de utilizarla, generalmente luego de los `#include` y siempre tiene acceso global dentro del archivo de código fuente.

En cambio, de esta forma puede aplicarse el ámbito donde debe actuar esta constante, generando desacoplamientos, que supone un mejor desarrollo de software.

```
int main() {
  const int MAX_CLIENTES = 500; // valor constante
  //...
  if(cantClientes < MAX_CLIENTES) {
  //...
  }
  //...
}
```