PROGRAMACIÓN EN C++. INTRODUCCIÓN

1. Introducción

C++ es un lenguaje de programación diseñado en 1979 por Bjarne Stroustrup. Para su creación se tomó como base un lenguaje de programación popular en aquella época el cual era C.

Por lo tanto, C++ es un derivado del mítico lenguaje C, el cual fue creado en la década de los 70 por la mano Dennis Ritchie para la programación del sistema operativo Unix (un sistema parecido a Unix es GNU/Linux), el cual surgió como un lenguaje orientado a la programación de sistemas y aplicaciones siendo su principal característica la eficiencia del código que produce.

C++ añade otro paradigma de programación a C: **la Programación Orientada a Objetos**, que nos permite ampliar los tipos de datos que podemos usar, definiendo clases y objetos.

C es un lenguaje de programación compilado. Utilizaremos un compilador que convertirá nuestro programa (código fuente) en un programa ejecutable. Existen compiladores de C para las distintas plataformas y sistemas operativos.

Para escribir un programa en C++ necesitamos un editor de texto (para escribir el código fuente) y un compilador para la plataforma y sistema operativo que estemos utilizando. Normalmente estas herramientas se unifican en los Entornos Integrados de Desarrollo (IDE) que son aplicaciones que nos ofrecen en un mismo programa distintas funcionalidades (editor de texto, compilador,…).

Algunos ejemplos de IDE que podemos usar: Visual Studio Code, Dev-C, Visual C, Zinjai, Eclipse, …

Durante los últimos años se han estandarizados distintas versiones de C donde se han ido añadiendo nuevas funcionalidades al lenguaje. Estas versiones se nombran con el año en que son publicadas, de esta manera tenemos: C98, C03, C11, C14, C17 (versión estable actual), C++20 (versión en desarrollo).

Podemos indicar como características fundamentales de C++:

- Su sintaxis es heredada del lenguaje C.
- Programa orientado a objetos (POO).
- Lenguaje muy didáctico, con este lenguaje puedes aprender muchos otros lenguajes con gran facilidad.
- Es portable y tiene un gran número de compiladores en diferentes plataformas y sistemas operativos.
- Permite la separación de un programa en módulos que admiten compilación independiente.
- Es un lenguaje de alto nivel.

2. Estructura general de un programa

Esta sería inicialmente la estructura de un programa en C++:

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

/* funcion main()
Es la función donde empieza la ejecución */

int main()
{
   cout << "Hola Mundo!!!"; // Imprime Hola Mundo
   return 0;
}
```

Veamos algunos elementos que forman parte de la estructura de un programa C++:

- `#include <iostream>`: con esta instrucción indicamos que vamos a utilizar la librería `iostream`. En esta librería están definidas las funciones de entrada/salida, por ejemplo `cout`.
- `using namespace std;`: Para que usemos el espacio de nombres estándar (`std`). Como podemos tener diferentes elementos en el lenguaje que se llamen igual se utiliza espacios de nombres para agruparlas. El espacio de nombres de las funciones de entrada / salida como `cout` o `cin` están definidos en el espacio de nombres `std`, por lo tanto, si indicamos que vamos a usarlos no será necesario nombrarlo cuando escribamos las instrucciones. Si no indicamos que vamos a usar el espacio de nombres `std` la instrucción que escribe en pantalla quedaría de la siguiente forma:

```
std::cout << "Hola Mundo!!!";
```

- `int main()`: Es la función principal del programa. Al ejecutar el programa son las instrucciones de esta función las que se empiezan a ejecutar. La función principal devuelve un valor entero (`int`) al sistema operativo. Si el programa va a tener parámetros en la línea de comandos, nos podemos encontrar esta función definida de esta manera:

```
int main(int argc, char *argv[])
```

- `cout << "Hola Mundo!!!";`: Instrucción que imprime en pantalla.
- `return 0;`: Como hemos dicho anteriormente la función `main()` devuelve un valor entero (0 si todo ha salido bien, distinto de 0 si se ha producido algún error). Esta instrucción devuelve el valor 0.
- Los bloques de instrucciones se guardan entre los caracteres `{` y `}`.
- Todas las instrucciones deben acabar en ;.
- En C++ podemos poner comentarios de una línea (utilizando los caracteres //) o comentarios de varias líneas (con los caracteres /* y */). Todos los comentarios son ignorados por el compilador.
- El lenguaje C++ distingue entre mayúsculas y minúsculas. Hay ciertas convenciones al nombrar, por ejemplo, el nombre de las variables se suele poner siempre en minúsculas, mientras que el nombre de las constantes se suele poner en mayúsculas.

3. Primer programa en el editor Zinjal

Los pasos serían los siguientes:

- **Paso 1**: Lo primero que debe hacer, es crear un nuevo programa simple. Se desplegará inmediatamente el Asistente para Nuevo Archivo. Allí seleccione la opción `Utilizar Plantilla`. A continuación, elegimos la versión de C que vamos a usar, por ejemplo "`Programa C14 en blanco`" y obtendremos un fichero con la estructura de nuestro programa:

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


int main(int argc, char *argv[]){

   return 0;
}
```

- **Paso 2:** Escribimos el programa en el editor. Recuerda que el programa se guarda con la extensión `cpp`.
- **Paso 3**: Para intentar ejecutar el programa presione `F9`, o seleccione la opción `Ejecutar` del menú `Ejecutar`.
  Esta acción guarda el archivo (si aún no tiene nombre la hará en un directorio temporal), lo compila, y si la compilación es exitosa lo ejecuta. Aparecerá en la parte inferior de la ventana principal el **Panel de Resultados del Compilador**, en el cual se muestra el estado de la compilación y los resultados de la misma.
  Si la compilación no tiene errores se ejecuta el programa y aparece una ventana de terminal donde vemos la ejecución. Luego de finalizar la ejecución, Zinjal informará el código de retorno de su programa (el `0` de la línea final `return 0;`, el cual sirve para saber si se ejecutó correctamente) y esperará a que presione `enter` una vez más antes de cerrar la ventana, para permitirle observar los resultados.
- **Paso 4**: Si hemos cometido algún error, por ejemplo se nos ha olvidado un `;` al final de una instrucción, en el proceso de compilación obtendremos errores:
  Al hacer doble `click` sobre el error en el panel de compilación el cursor se desplaza hacia la línea que lo provocó

Atajos de teclado

A continuación, se listan algunas combinaciones de teclas para aprovechar mejor algunas facilidades de edición Zinjal:

- `F9`: Este atajo realiza todos los pasos necesarios para probar un programa (guardar, compilar y ejecutar). Si se presiona `Shift+F9`, se evita el último paso; es decir, solo se compila. Esto sirve para saber si el código es sintácticamente correcto.
- `Ctrl+<`: Si la compilación arroja errores o advertencias, con esta combinación se pueden recorrer los mismos.
- `Ctrl+H`: Esta combinación busca la librería que contiene la declaración de una determinada clase, función, variable o macro e inserta al principio del archivo el #include que corresponda para poder utilizarla.

4. Datos y tipos de datos

Podemos clasificar los tipos de datos de la siguiente manera:

- **Tipos de datos simples**:
  - Números enteros (`int`)
  - Números reales (`float` o `double`)
  - Valores lógicos (`bool`)
  - Caracteres (`char`)
- **Tipos de datos complejos**:
  - Arrays
  - Cadena de caracteres
  - Estructuras de datos

Los datos con los que podemos trabajar en un programa lo podemos indicar de tres formas distintas:

- **Literales**: Los literales nos permiten representar valores. Por ejemplo, un literal entero podría ser el 5.
- **Variables**: Una variable es un identificador que guarda un valor. Las variables se declaran de un determinado tipo de datos. Por ejemplo, una variable entera puede guardar datos enteros.
- **Constantes**: Podemos declarar un valor de un determinado tipo por medio de un identificador. Mientras el valor de una variable puede cambiar a lo largo de la ejecución de un programa, las constantes no pueden cambiar.
- **Expresiones**: Por último, indicar que podemos hacer operaciones entre los distintos datos. El tipo de dato de una expresión dependerá del resultado de la operación. Según el tipo de datos con los que trabajemos tenemos distintos tipos de:
  - **Operadores aritméticos**: Para hacer operaciones con tipos de datos numéricos.
  - **Operadores relacionados**: Nos permiten comparar datos y nos devuelven valores lógicos
  - **Operadores lógicos**: Nos permiten trabajar con valores lógicos.
  - **Operadores de asignación**: Nos permiten asignar valores a variables.
  - **Otros operadores**: Durante el curso veremos algunos operadores más, por ejemplo, para trabajar con bits o para trabajar con punteros.

5. Literales y Constantes

Literales

Los literales nos permiten representar valores. Estos valores pueden ser de diferentes tipos:

- **Literales enteros**: Ejemplos números en base decimal: 5,-12…, en base octal: 077 y en hexadecimal 0xfe.
- **Literales reales**: Utilizamos un punto para separar la parte entera de la decimal. Por ejemplo: 3.14159. También podemos usar la letra e o E seguida de un exponente con signo para indicar la potencia de 10 a utilizar, por ejemplo: 6.63e-34, 35E20.
- **Literales booleanos o lógicos**: Los valores lógicos solo tienen dos valores: `false` para indicar el valor falso, y `true` para indicar el valor verdadero.
- **Literales carácter**: Para indicar un valor de tipo carácter usamos la comilla simple `'`. Por ejemplo `'a'`. tenemos algunos caracteres especiales que son muy útiles, por ejemplo `\n` indica nueva línea y `\t` indica tabulador.
- **Literales cadenas de caracteres**: Una cadena de caracteres es un conjunto de caracteres. Para indicar cadenas de caracteres usamos las dobles comillas `"`, por ejemplo: `"Hola"`.

Constantes

Una constante es un valor que identificamos con un nombre cuyo valor no cambia durante la ejecución del programa. Para crear constantes usamos:

```
# define identificador valor
```

Un ejemplo:

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

#define ANCHURA 10
#define ALTURA  5
#define NUEVALINEA '\n'

int main() {
   int area;

   area = ANCHURA * ALTURA;
   cout << area;
   cout << NUEVALINEA;
   return 0;
}
```

Otra forma de crear constantes es usar el modificador `const` al crear una variable. Lo veremos cuando estudiemos las variables.

6. Variables y operadores de asignación

Una variable nos permite almacenar información. Cada variable tiene un nombre y al crearlas hay que indicar el tipo de datos que va a almacenar.

El nombre de una variable puede estar formado por letras, dígitos y subrayados, pero no puede empezar por un dígito. Los nombres se suelen indicar en minúsculas.

Declaración de variables

Para poder usar una variable tenemos que declararla, la declaración consiste en indicar el tipo y el nombre de la variable. Además, en una declaración también podemos inicializar la variable con un valor. Veamos distintos ejemplos donde declaramos variables de tipo entero:

```
int variable1;
int variable2=10;
int variable3, variable4, variable5;
int variable6=100, variable7=-10;
```

Podemos declarar una variable y posteriormente inicializarla (con el operador de asignación =).

```
int variable8;
variable8=1000;
```

Si declaramos una variable y no la inicializamos tendrá un valor por defecto: las variables numéricas tendrán un valor de `0` y los caracteres tendrán un valor de `\0`.

Las variables hay que declararlas antes de usarla, pero es muy recomendable declarar todas las variables al principio de la función.

Ámbito de las variables

Según donde declaremos las variables podrán ser:

Variables locales

Están definidas dentro de una función o un bloque de instrucciones. Existen y se pueden usar sólo en esa función o bloque. Ejemplo:

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

int main () {
   // Declaración de variable local
   int a, b;
   int c;
```

Variables globales

Están declaradas fuera de las funciones, se pueden usar en cualquier función del programa. No es recomendable su uso. Ejemplo:

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

// Declaración de variable global
int g;

int main () {
   // Declaración de variable local
   int a, b;
```

Operadores de asignación

Nos permiten guardar información en las variables. En una variable podemos guardar un literal, otra variable o el resultado de una expresión. Los operadores de asignación son los siguientes:

- `=`: Asignación simple, por ejemplo: `a=b+7;`
- `+=`: Suma y a continuación asigna. Por ejemplo: `a+=b` es igual que `a=a+b`. También podemos usar los operadores, `-=`, `*=`, `/=`,...

7. Tipos de datos numéricos y operadores aritméticos

Los principales tipos de datos numéricos son los siguientes:

- `int`: Nos permite guardar valores enteros.
- `float`: Nos permiten guardar números reales de precisión simple (7 dígitos decimales).
- `double`: Nos permite guardar números reales de doble precisión (16 dígitos decimales).

El tipo de un dato determina el tamaño de memoria que se va a utilizar para guardarlo y por lo tanto determina los valores que podemos representar.:

Tipo	Memoria	Rango de valores
`int`	4 bytes	-2147483648 a 2147483647
`float`	4 bytes	-3.4E+38 a +3.4E+38
`double`	8 bytes	-1.7E+308 to +1.7E+308

El tipo de dato carácter

Para guardar un carácter usamos el tipo de dato `char`. Realmente el tipo de dato `char` es un tipo entero, ocupa 1 byte de memoria por lo que puede guardar 256 valores. En un variable de tipo `char` guardaremos un número que corresponde código ASCII del carácter.

Veamos un ejemplo donde se imprimen dos caracteres a:

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

int main(int argc, char *argv[]) {
    char caracter1='a';
    char caracter2=97;
    cout << caracter1 << endl;
    cout << caracter2 << endl;
    return 0;
}
```

Modificadores de tipos

Podemos modificar los tipos anteriores para dos cosas: para aumentar la memoria utilizada y por lo tanto aumentar el rango de valores representables, y para indicar si se usan número negativos o no.

Los modificadores son los siguientes: `signed`, `unsigned`, `long` y `short`.

- Los modificadores `signed`, `unsigned`, `long` y `short` se pueden aplicar al tipo `int`.
- Los modificadores `signed`, `unsigned` se puede aplicar al tipo `char`.
- El modificador `long` se puede aplicar al tipo `double`.
- Los modificadores `signed`, `unsigned` también se pueden usar como prefijos de los modificadores `long` y `short`.

Con estas reglas nos quedarían esta tabla con todos los tipos:

Tipo	Memoria	Rango de valores
`char`	1 byte	-127 a 127 o 0 a 255
`signed char`	1 byte	-127 a 127
`unsigned char`	1 byte	0 a 255
`int`	4 bytes	-2147483648 a 2147483647
`signed int`	4 bytes	-2147483648 a 2147483647
`unsigned int`	4 bytes	0 a 4294967295
`short int`	2 bytes	-32768 a 32767
`signed short int`	2 bytes	-32768 a 32767
`unsigned short int`	2 bytes	0 a 65535
`long int`	8 bytes	-9223372036854775808 a 9223372036854775807
`signed long int`	8 bytes	-9223372036854775808 a 9223372036854775807
`unsigned long int`	8 bytes	0 a 18446744073709551615
`float`	4 bytes	-3.4E+38 a +3.4E+38
`double`	8 bytes	-1.7E+308 a +1.7E+308
`long double`	16 bytes	3.3621e-4932 a 1.18973e+4932

Al declarar variables podemos usar una notación corta para declarar enteros largos, cortos o sin signos, donde no tenemos que indicar el tipo `int`, por ejemplo, estas dos declaraciones son correctas:

```
unsigned short var1;
long var2;
```

Podemos usar sufijos para indicar literales numéricos largos (con el sufijo `L`) y sin signo (con el sufijo `U`). Por ejemplo:

```
var1=123U;
var2=123L;
```

Conversión de tipos

- Por defecto C++ hace ciertas conversiones de forma implícitas: Por ejemplo, de `char` a `int` o de `int` a `float`. Por ejemplo, esto es correcto:

```
int i = 10;
float d = i;
```

- Si tenemos expresiones donde hacemos operaciones entre datos del mismo tipo, el resultado de la expresión será del mismo tipo.
- Si tenemos expresiones donde hacemos operaciones entre datos de distintos tipos el resultado de la expresión será del tipo con más precisión de los datos operados.
- Por último, podemos hacer una conversión explicita usando una expresión de `typecast`, por ejemplo, para convertir un `int` a un `float`:

```
int num1=10,num2=3;
float res;
res=(float)num1 / float(num2)
//Tenemos dos formas de hacer la conversión
```

Veamos un ejemplo con el operador aritmético de división:

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

int main(int argc, char *argv[]) {
    int num1=4,num2=3;
    cout << 4 / 3 << endl;
    cout << 4.0 / 3 << endl;
    cout << float(num1) / num2 << endl;
    return 0;
}
```

1. El resultado de la primera instrucción `cout` es `1`, porque hemos dividido dos enteros y el resultado es entero. Realmente hemos hecho una división entera donde la parte decimal la hemos truncado.
2. El resultado de las otras instrucciones `cout` es `1.3333` ya que dividimos un `float` por un `int` y el resultado es `float`.

Operadores aritméticos

- `+`: Suma dos números
- `-`: Resta dos números
- `*`: Multiplica dos números
- `/`: Divide dos números.
- `%`: Módulo o resto de la división
- `+`, `-`: Operadores unarios positivo y negativo
- `++`: Operador de incremento. Suma uno a la variable, i`++` es lo mismo que `i=i+1`.
- `--`: Operador de decremento. Resta uno a la variable.

Funciones matemáticas

En la librería `cmath` tenemos distintas funciones matemáticas. Las más útiles que podemos usar en nuestros programas son:

- `double pow(double, double);`: Realiza la potencia, la base es el primer parámetro y el exponente el segundo. Recibe datos de tipo `double` y devuelve también un valor `double`.
- `double sqrt(double);`: Realiza la raíz cuadrada del parámetro `double` que recibe. Devuelve un valor `double`.

- `int abs(int);`: Devuelve el valor absoluto (valor entero) del número entero que recibe como parámetro.

Veamos un ejemplo:

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

int main(int argc, char *argv[]) {
    int num1=4, num2=2;
    cout << "Potencia:" << pow(num1,num2) << endl; //Potencia
    cout << "Raíz Cuadrada:" << sqrt(num1) << endl; //Raíz cuadrada
    num1=-4;
    cout << "Valor absoluto:" << abs(num1) << endl; //Valor absoluto
    return 0;
}
```

8. Introducción a las cadenas de caracteres

En C++ tenemos dos formas de trabajar con cadenas de caracteres:

- La forma tradicional del lenguaje C: una cadena de caracteres es un array de caracteres (un conjunto de caracteres).
- C++ nos ofrece la clase `string` que nos permite trabajar con cadenas de caracteres.

Usando la clase string

Para declarar una variable de tipo cadena utilizamos el tipo de datos (que en este caso es una clase) `string`, además podemos inicializar las cadenas en la declaración, ejemplos:

```
string cadena1;
string cadena2="Hola Mundo"
```

Indexación de caracteres

Los caracteres que forman una cadena se pueden referenciar por un índice (un número), de tal modo que el primer carácter está en la posición `0`, el segundo en la posición `1`, y así consecutivamente. Por ejemplo, para imprimir el primer carácter:

```
cout << cadena2[0];
```

El método length o size nos devuelve el número de caracteres de una cadena, por ejemplo:

```
cout << cadena2.length();
10
```

Por lo tanto, para mostrar el último carácter de la cadena (que está en la posición 9):

```
cout << cadena2[cadena2.length() - 1];
o
```

Concatenación de cadenas

El operador `+` me permite unir datos de tipo cadenas.

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

int main(int argc, char *argv[]) {
    string nombre="Pepe";
    string apellidos="García";
    string nombre_completo;
    nombre_completo=nombre+" "+apellidos;
    cout << nombre_completo;
    return 0;
}
```

9. Entrada y salida estándar

Veamos un ejemplo para estudiar la entrada y salida estándar:

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

int main(int argc, char *argv[]) {
    string nombre;
    int edad;
    cout << "Bienvenido..." << endl;
    cout << "Dime el nombre:";
    getline(cin,nombre);
    cout << "Dime la edad de " << nombre << ":";
    cin >> edad;
    cout << nombre << " tiene " << edad << " años.";
    return 0;
}
```

La librería `iostrem` que incluimos al principio del programa tiene las funciones necesarias para gestionar la entra y salida estándar.

Salida por pantalla

El identificador `cout` representa la salida estándar (pantalla). El operador `<<` nos permite enviar información a la salida estándar. Como vemos en el programa en una misma instrucción podemos enviar a la salida estándar varios datos de distintos tipos. El identificador `endl` cambia de línea al terminar de imprimir por pantalla, sería lo mismo que imprimir un carácter `\n`.

Entrada por teclado

El identificador `cin` representa la entrada estándar (teclado). El operador `>>` nos permite guardar la información que introducimos por teclado en una variable. El operador `>>` hace la conversión necesaria para guardar la información en la variable según del tipo que la hayamos declarado. Por ejemplo, en la siguiente instrucción:

```
cin >> edad;
```

Sabemos que edad es de tipo entero, si introducimos por teclado un dato que no numérico, no se guardará nada en la variable y su valor será el que tiene por defecto, un `0`. Si introducimos un valor con decimales, se guarda sólo la parte entera. Si introducimos un valor que empieza por número y continúa con caracteres, se guarda la parte numérica entera.

Lectura de cadena de caracteres

Por defecto el operador `>>` no es capaz de leer los espacios que podemos poner entre palabras al leer una variable cadena, por ejemplo:

```
...
string nombre;

cout << "Dime el nombre:";
cin >> nombre;
cout << nombre;
...
```

Si introducimos por teclado “Pepe García”, la salida del programa será “Pepe”.

Por lo tanto, para poder leer los espacios al leer cadenas de caracteres vamos a usar la función `getline`, donde indicamos como parámetros la entrada estándar (`cin`) y la variable que de tipo `string` que vamos a leer. Por ejemplo:

```
...
string nombre;


cout << "Dime el nombre:";
getline(cin,nombre);
cout << nombre;
...
```

En ocasiones cuando hemos leído con `cin >>`, la siguiente instrucción `getline` no lee nada por teclado, en realidad lo que pasa es que lee del buffer de entrada el carácter de retorno. Para evitar esto antes de usar la instrucción `getline` hay que borrar el buffer usando `cin.ignore()`, veamos un ejemplo:

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

int main(int argc, char *argv[]) {

    int edad;
    string nombre;
    cout << "Introduce la edad:";
    cin >> edad;
    cout << "Introduce el nombre:";
    cin.ignore();
    getline(cin,nombre);
    return 0;
}
```

C++. Estructuras de control y tipos de datos

1. Datos lógicos o Booleanos

El tipo booleano o lógico `bool` puede tener dos estados expresados por las constantes predefinidas `true` (lo que lo convierte en el entero `1`) y `false` (lo que lo convierte en el entero `0`). Realmente cualquier valor entero distinto de `0` será verdadero, y el `0` será falso.

Operadores de comparación

El valor devuelto por una operación de comparación es un valor lógico:

- `>`: Mayor que
- `<`: Menor que
- `==`: Igual que
- `<=`: Menor o igual
- `>=`: Mayor o igual
- `!=`: Distinto

Operadores lógicos

El valor devuelto por una operación lógica es un valor lógico:

- `&&`: Conjunción, operación AND (Y).
- `||`: Disyunción, operación OR (O).
- `!`: Negación, operación NOT (NO).

a	b	AND a && b	OR a||b	NOT !a
V	V	V	V	F
V	F	F	V	F
F	V	F	V	V
F	F	F	F	V

Comparación de cadenas de caracteres

Las cadenas se comparan carácter a carácter, en el momento en que dos caracteres no son iguales se compara alfabéticamente (es decir, se convierte a código ASCII y se comparan).

Ejemplos

- “a”>”A” es verdadero.
- “informatica”>”informacion” es verdadero.
- “abcde”>”abcdef” es falso.

2. Estructura alternativa IF

Alternativa simple

La estructura es la siguiente:

```
if (expresión lógica)
{
    instrucciones
}
```

Alternativa doble

La estructura es la siguiente:

```
if (expresión lógica)
{
    instrucciones A
}
else
{
    instrucciones B
}
```

El bloque de instrucciones "instrucciones A" e "instrucciones B" se encierran entre llaves si son más de una, Enel caso de que sólo sea una instrucción es opcional poner las llaves. Ejemplo:

```
if (edad>=18)
{
      cout << "Eres mayor de edad" << endl;
}
else
{
      cout << "Eres menor de edad" << endl;
}
```

```
if (edad>=18)
      cout << "Eres mayor de edad" << endl;
else
      cout << "Eres menor de edad" << endl;
```

Estas dos estructuras alternativas son idénticas.

3. Estructura alternativa SWITCH

La secuencia de instrucciones ejecutada por una instrucción `switch` depende del valor de una variable numérica.

```
switch(expresión)
{
    case valor1:
        instrucciones;
        break; //opcional
    case valor2:
        instrucciones;
        break; //opcional
    ...
default: //opcional
    statement(s);
}
```

- Esta instrucción permite ejecutar opcionalmente varias acciones posibles, dependiendo del valor de una expresión.
- La expresión que se utiliza en una instrucción `switch` debe ser un entero (`int` o `char`).
- Un `switch` puede tener varios comparaciones `case` donde indicaremos el valor con el que se va a comparar, terminado por "`:`". El valor debe ser del mismo tipo de la variable que hemos indicado en la instrucción `switch`.
- En el momento que una comparación `case` se cumple se ejecutarán todas las instrucciones (incluido los `case` posteriores) hasta que se encuentre una instrucción `break`.
- Podemos indicar una opción por defecto `default`, que debe aparecer al final de la instrucción y que se ejecuta sin ninguna opción anterior se ha cumplido.

Ejemplo:

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

int main(int argc, char *argv[]) {
    int nota;
    cout << "Dime tu nota:";
    cin >> nota;
    switch(nota)
    {
        case 1:
        case 2:
        case 3:
        case 4:
            cout << "Suspenso";
            break;
        case 5:
            cout << "Suficiente";
            break;
        case 6:
            cout << "Bien";
            break;

        case 7:
        case 8:
            cout << "Notable";
            break;
        case 9:
        case 10:
            cout << "Sobresaliente";
            break;
        default:
            cout << "Nota incorrecta";
    }
    cout << endl << "Programa terminado";
    return 0;
}
```

4. Estructuras repetitivas

Estructura repetitiva WHILE

Ejecuta un conjunto de instrucciones mientras una condición sea verdadera:

```
while (condición)
{
   Instrucciones;
}
```

- Al iniciarse el bucle, la condición es evaluada, si es verdadera, ejecuta el conjunto de instrucciones indicado entre las llaves. Una vez que alcanza la llave de cierre de la estructura, vuelve a la condición y la vuelve a evaluar. Si sigue siendo verdadera, vuelve a ejecutar las instrucciones y si no continúa secuencialmente con las instrucciones indicas después de la llave de cierre de la estructura `while`.
- Se puede dar la circunstancia de que nunca se ejecuten las instrucciones del bucle si la primera vez la condición es falsa.
- En las instrucciones del bucle, debe haber una que pueda asignar un valor o valores a las variables implicadas en la condición para que, al evaluarla, de como respuesta el valor falso. En caso contrario, si la primera vez la condición es falsa, entraríamos en un bucle infinito.

Instrucción break

Esta instrucción funciona en todos los tipos de bucle, su función es la de romper o finalizar la ejecución de un bucle y saltar a la instrucción posterior a la llave de cierre de este.

Es un tipo de instrucción no muy bien vista entre los programadores que defiende un tipo de programación estructurada o purista. Estos prefieren que sea la condición del bucle la que determine la finalización de ejecución de este. Aún así, es importante conocerla porque nos podría hacer falta en algún programa.

Ejemplo:

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

int main(int argc, char *argv[]) {
   string secreto,clave;
   char otra;

   secreto="asdasd";
   cout << "Introduzca clave: ";
   getline (cin,clave);

   while (clave!=secreto)
   {
        cout << "Clave incorrecta" << endl;
        cout << "Desea introducir otra clave (S/N): ";
        cin >> otra;
        if (toupper(otra)=='N')
             break;
        cin.ignore();
        cout << endl << "Introduzca clave: ";
        getline (cin,clave);
   }

   if (clave==secreto) cout << "Clave correcta \n";

   cout << "Fin del programa";
   return 0;
}
```

Instrucción continue

Esta instrucción también funciona en todos los tipos de bucle y realiza un salto directamente a la condición del bucle, saltándose todas las instrucciones que haya desde ella hasta la llave de cierre del bucle.

Un pequeño ejemplo, escribe lo números pares entre 0 y 10:

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

int main(int argc, char *argv[]) {
  int numero=0;
  while (numero<10)
  {
       numero++;
       if (numero % 2 != 0) continue;
       cout << numero << endl;
  }
  return 0;
}
```

Estructura repetitiva DO - WHILE

La instrucción `do-while` ejecuta una secuencia de instrucciones mientras que la condición sea verdadera:

```
do
{
    instrucciones;
}
while(condición lógica);
```

- Al ejecutarse esta instrucción, la secuencia de instrucciones que forma el cuerpo del ciclo se ejecuta una vez y luego se evalúa la condición. Si la condición es verdadera, el cuerpo del ciclo se ejecuta nuevamente y se vuelve a evaluar la condición. Esto se repite hasta que la condición sea falsa.
- Dado que la condición se evalúa al final, las instrucciones del cuerpo del ciclo serán ejecutadas al menos una vez.
- Además, a fin de evitar ciclos infinitos, el cuerpo del ciclo debe contener alguna instrucción que modifique la o las variables involucradas en la condición de modo que en algún momento la condición sea false y se finalice la ejecución del ciclo.

Ejemplo:

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

int main(int argc, char *argv[]) {
    string secreto, clave;

    secreto = "asdasd";
    do
    {
            cout << "Dime la clave:";
            getline(cin,clave);
            if (clave!=secreto) cout << "Clave incorrecta!!!" << endl;
    } while(clave!=secreto);
    cout << "Bienvenido!!!" << endl;
    cout << "Programa terminado";
    return 0;
}
```

Estructura repetitiva FOR

La instrucción `for` ejecuta una secuencia de instrucciones un número determinado de veces:

```
for(<variable> = <inicial>; <condición>;
                              <incremento/decremento variable>)
{
    <instrucciones>
}
```

- Al ingresar al bloque, la variable `<variable>` recibe el valor `<inicial>` (1ª parte de la instrucción) y se ejecuta la secuencia de instrucciones que forma el cuerpo del ciclo.
- Luego se `incrementa/decrementa` la variable como hayamos indicado (3ª parte de la instrucción) y se evalúa la `condición` (2ª parte de la instrucción).
- Si la condición es verdadera se realiza otra iteración al bucle.
- Si la condición es falsa se termina el bucle.

Ejemplo:

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

int main(int argc, char *argv[]) {
    int var;

    for(var=1;var<=10;var++)
    {
        cout << var << " ";
    }
    return 0;
}
```

Contadores, acumuladores e indicadores

Un `contador` es un tipo especial de variable que utilizamos principalmente para contar una serie de sucesos. Realmente es una variable que va incrementando su valor de forma constante.

Este tipo de variables estará siempre compuesto por dos instrucciones: una primera que le asigna un valor inicial; y la segunda, la expresión matemática de incremento.

Los `contadores` se definen como de tipo entero (`int`)

Ejemplo:

Introducir 5 número y contar los números pares.

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

int main(int argc, char *argv[]) {
    int var,cont,num;

    cont=0;
    for(var=1;var<=5;var++)
    {
        cout << "Dime un número:";
        cin >> num;
        if(num % 2 == 0)
            cont=cont+1; //cont++
    }
    cout << "Has introducido " << cont << " números pares.";
    return 0;
}
```

Los `acumuladores` son parecidos a los contadores, pero la diferencia es que su incremento no es constante, puede tener un valor distinto en cada ejecución de su instrucción de incremento. Al igual que los `contadores`, un `acumulador` también debe llevar una instrucción de inicialización.

Este tipo de variables podemos definirlas como reales.

Ejemplo:

Introducir 5 número y sumar los números pares.

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

int main(int argc, char *argv[]) {
    int var,suma,num;

    suma=0;
    for(var=1;var<=5;var++)
    {
        cout << "Dime un número:";
        cin >> num;
        if(num % 2 == 0)
            suma=suma+num; //suma+=num
    }
    cout << "La suma de los números pares es " << suma;
    return 0;
}
```

Un `indicador` es una variable lógica, que usamos para recordar o indicar algún suceso. Un indicador:

- Se **declara** como una variable lógica:
  `bool indicador;`
- Se **inicializa** a un valor lógico que indica que el suceso no ha ocurrido.
  `indicador = false;`
- Cuando ocurre el suceso que queremos recordar cambiamos su valor.
  `indicador = true;`

Ejemplo:

Introducir 5 número e indicar si se ha introducido algún número par.

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

int main(int argc, char *argv[]) {
    int var,num;
    bool indicador;

    indicador = false;
    for(var=1;var<=5;var++)
    {
        cout << "Dime un número:";
        cin >> num;
        if(num % 2 == 0)
            indicador = true;
    }
    if(indicador)
        cout << "Has introducido algún número par";
    else
        cout << "No has introducido algún número par";
    return 0;
}
```

5. Cadenas de caracteres

Principales métodos de la clase string

Vamos a ver los ejemplos a partir de esta cadena:

```
string cadena="informática";
```

- `Clear()`: Borra todos los caracteres de una cadena. Ejemplo:

```
cadena.clear(); // Igual que: cadena = "";
cout << cadena;
```

- `length()` o `size()`: Devuelve la cantidad de caracteres que tiene una cadena:

```
cadena.length()
```

- `front()` y `back()`: Devuelve el primer carácter y el último de una cadena:

```
cadena.front()
cadena.back()
```

- `append()`: Nos permite añadir otra cadena al final de la primera. Ejemplo:

```
cadena.append(" moderna")
```

La salida sería:

```
informática moderna
```

Para añadir una variable de tipo carácter tengo que indicar el número de caracteres:

```
char car='0';
cadena.append(1,car);
cout << cadena;
```

La salida sería:

```
informática moderna0
```

- `find()`: Busca la aparición de una subcadena dentro de una cadena. Devuelve la posición de la subcadena encontrada. Si no la encuentra devuelve -1. Ejemplo:

```
cout << "Posición del primer carácter i: " <<
                cadena.find("i") << endl;

cout << "Posición del segundo carácter i: " <<
                cadena.find("i",1) << endl;
```

La salida sería:

```
Posición del primer carácter i: 0
Posición del segundo carácter i: 8
```

- `substr()`: Devuelve una subcadena de una cadena. Si utilizo `substr(a,b)` obtiene una subcadena desde el índice `a` que tiene `b` caracteres, si utilizo substr(a) obtengo una subcadena desde el índice `a` hasta el final. Ejemplo:

```
string subcad;
subcad=cadena.substr(2,3);
cout << subcad << endl;
subcad=cadena.substr(5);
cout << subcad << endl;
```

La salida sería:

```
for
mática
```

- `replace():` Reemplaza parte de una cadena por otra. Se utiliza como replace(a, b, str) que remplaza b caracteres desde el índice a por la cadena str. Ejemplo:

```
cadena.replace(6,5,"ación");
cout << cadena << endl;
```

La salida sería:

```
Información
```

- `toupper():` Convierte una cadena a mayúscula. Ejemplo:

```
cadena=toupper("Informatica");
cout << cadena << endl;
```

La salida sería:

```
INFORMATICA
```

- `tolower():` Convierte una cadena a minúscula. Ejemplo:

```
cadena=toupper("Informatica");
cout << cadena << endl;
```

La salida sería:

```
informatica
```

6. Tipos de datos complejos: Arrays

Hasta ahora, para hacer referencia a un dato utilizábamos una variable. El problema se plantea cuando tenemos gran cantidad de datos que guardan entre sí una relación. No podemos utilizar una variable para cada dato.

Para resolver estas dificultades se agrupan los datos en un mismo conjunto, estos conjuntos reciben el nombre de **estructura de datos**.

Arrays

Un array es una estructura de datos con elementos homogéneos, del mismo tipo, numérico o alfanumérico, reconocidos por un nombre en común. Para referirnos a cada elemento del array usaremos un índice (empezamos a contar por **0**).

Declaración de arrays

Para declarar un array debemos indicar el número de elementos que va a tener el array.

```
<tipo> <identificador>[N1][N2]...[Nn]
```

Esta instrucción define un array con el nombre indicado en y **n** dimensiones. Los n parámetros indican la cantidad de dimensiones y el valor máximo de cada una de ellas. La cantidad de dimensiones puede ser una o más, y la máxima cantidad de elementos debe ser una expresión numérica positiva.

Por ejemplo, definimos un array de una dimensión (también llamado **vector**) de 10 elementos enteros:

```
int vector[10];
```

Otro ejemplo, definir un array de dos dimensiones (también llamado **matriz** o **tabla**) de 3 filas y 4 columnas de cadenas:

```
string tabla[3][4];
```

Para acceder a un elemento de un array se utiliza un índice. El primer elemento está en la posición 0.

Para asignar un valor a un elemento del vector:

```
vector[0]=10;
```

Para mostrar el primer elemento del vector:

```
cout << vector[0];
```

Otro ejemplo, asignamos y mostramos el segundo elemento de la segunda fila de la tabla:

```
tabla[1][1] = "Hola";
cout << tabla[1][1];
```

Características de los arrays

- Un array es una estructura estática. Si crea un array de enteros con 10 elementos, se reservarán en memoria el espacio para guardar los 10 enteros, y durante la ejecución del programa ese tamaño no podrá cambiar.
- Al acceder o al cambiar el valor de un elemento de un array no se comprueba que estemos accediendo o modificando un elemento del array. Es decir, si tenemos un array llamado `vector` de 10 enteros, esta instrucción no producirá error:

  ```
  vector[10] = 3;
  ```

  La última posición del vector es la 9, sin embargo, estamos modificando la posición 10, es decir estamos modificando una posición de memoria que no corresponde al vector.

  Del mismo modo puedo mostrar valores que están guardados en memoria y no corresponden al vector:

  ```
  cout << vector[15];
  ```