TEMA 0
INTRODUCCIÓN AL LENGUAJE C++

1. El programa "Hola mundo"
2. Tipos simples
3. Declaración, inicialización y asignación de variables y constantes
4. Operadores
5. Tipos definidos por el programador
6. Estructuras de control
7. Procedimientos y funciones
8. Entrada y salida

Bibliografía: *El lenguaje de programación C++*. Tercera edición
Bjarne Stroustrup
Addison Wesley, 1998

0.1 Hola mundo

- El programa "Hola mundo" en C++

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

void main( )
{
  cout << "Hola mundo mundial\n";
}
```

- El programa principal es siempre una función de nombre *main*.

- Las operaciones de entrada/salida se importan de módulos de la biblioteca usando la directiva *include*. En el ejemplo se ha importado *iostream*.
  - Los módulos de la biblioteca estándar definen sus identificadores en el espacio de nombres *std*, por lo tanto, si no se quiere cualificar a los identificadores es necesario añadir la sentencia

    `using namespace std;`

    para así poder escribir *cout* en lugar de *std::cout*

- Los delimitadores { } marcan, respectivamente, el principio y el final de una función y sirven así mismo para delimitar bloques de código.

- El tipo *void* representa la *ausencia de valor*.

  En C++ un procedimiento es una función que devuelve un resultado de tipo *void*.

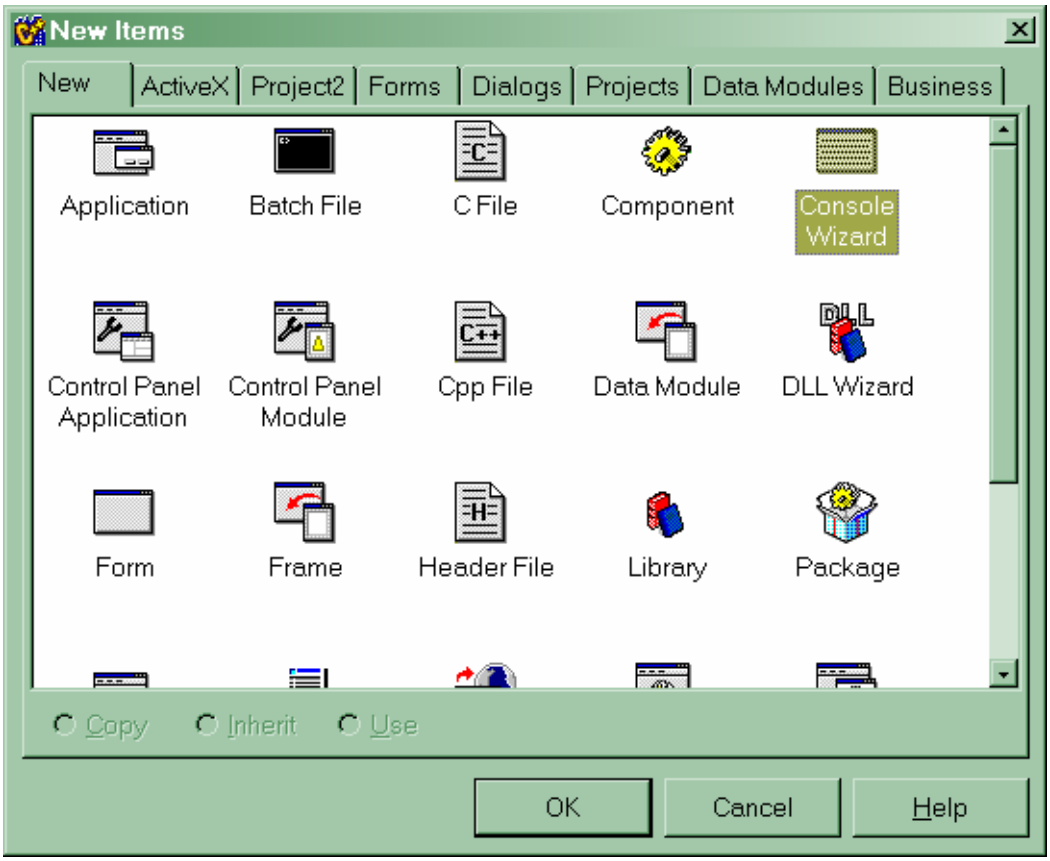
- Todas las sentencias terminan con el carácter ;

- Las cadenas se escriben entre comillas dobles.

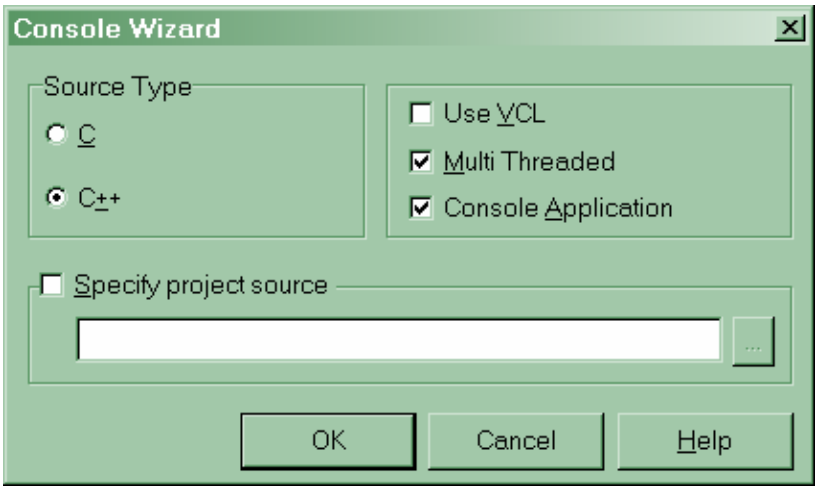
- Para mostrar un mensaje por la salida estándar (la pantalla), se aplica el operador de inserción << teniendo como primer operando *cout* (importado de *iostream*)y como segundo operando el dato que se pretende mostrar.

"Hola mundo" (en C++ Builder 5)

- *File* | *New* ...
- En el cuadro de diálogo *New Items* se selecciona *Console Wizard*



- En el diálogo *Console Wizard* se aceptan las opciones por defecto



– Y ya tenemos un programa ejecutable

```
//---------------------------------------------------------------------------
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
        return 0;
}
//---------------------------------------------------------------------------
```

sin más que seleccionar *Run* | *Run* o pulsar F9.

– Si el programa no se ha compilado previamente, se compilará al seleccionar la orden *Run*.

Para compilar se utiliza la orden *Project* | *Compile unit* (ALT+F9) para compilar el archivo actual o *Project* | *Make nombre_proyecto* (CTRL+F9) para compilar todos los archivos de un proyecto.

– Si añadimos a este esqueleto de programa las líneas de nuestro "Hola mundo"

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

int main(int argc, char* argv[])
{
  cout << "Hola mundo\n";
  return 0;
}
```

Si ejecutamos ahora este programa, veremos en Windows una consola de MS-DOS que se abre y cierra rápidamente. Para que nos dé tiempo a ver el mensaje, podemos añadir una operación de lectura

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

  cout << "Hola mundo\n";
  cin >> c;
  return 0;
}
```

0.2 Tipos simples

- Los tipos simples más utilizados

  – *char*
  caracteres, los literales se escriben entre comillas simples: 'a'

  – *int*
  números enteros en el intervalo −32768 .. 32767

  – *double*
  números reales en el intervalo 1.7e−308 .. 1.7e+308

  – *bool*
  valores lógicos *true* | *false*

  – *void*
  el tipo del resultado de las funciones que no devuelven resultado (i.e., los procedimientos)

0.3 Declaración, inicialización y asignación de variables y constantes

- Sintaxis

  tipo lista_de_variables;

  donde *lista_de_variables* se compone de uno o más identificadores separados por comas.

  En los identificadores se distinguen mayúsculas de minúsculas.

- Una declaración puede aparecer en cualquier lugar de un programa, su ámbito será local si se declara dentro de una función y global si se declara al nivel más externo.

  En general siempre declararemos las variables locales al principio del cuerpo de las funciones.

- = es el operador de asignación

- Las variables se pueden inicializar al ser declaradas

- Las constantes se declaran como las variables pero precedidas de la palabra reservada *const*

- Ejemplo

```
int global = 0;                  // variable global inicializada

void main( )
{
  int local1, local2 = 2;        // variables locales a la función main
  const int uno = 1, dos = 2;    // declaración de constantes

  local1 = uno;                  // asignación

  bool b;                        // evitaremos estas declaraciones
  b = true;
}
```

0.4 Operadores

0.4.1 Operadores aritméticos y de manipulación de bits

- Actúan sobre constantes y variables numéricas
  - +, –, *, / : suma, resta, multiplicación y división
  - % : módulo, que sólo está definido para los enteros (int o long)
  - &, |, ^, <<, >> : operaciones bit a bit y-lógica, o-lógica, o-exclusiva, desplazamiento a la izquierda y desplazamiento a la derecha

- Cada uno de estos operadores lleva asociado un *operador de asignación* de la forma *op*= tal que

  variable op= expresión

  es equivalente a

  variable = variable op expresión

- Ejemplo

```
void main( )
{
  const int tres = 3;
  int n = 289898454 % tres;
  double f = 1.34e-6, g = 6.7;
  double h = f + g * f;

  n += tres;
}
```

0.4.2 Operadores relacionales y lógicos

- Operadores relacionales
  - ==, != : igual, diferente
  - >, >=, <, <= : mayor, mayor o igual, menor, menor o igual

- Operadores lógicos
  - && : conjunción
  - || : disyunción
  - ! : negación

- Ejemplo

```
void main( )
{
  double f = 1.34e-6, g = 1.35e-6;
  char h, j;
  bool b = ! (f == g) && (f*3.6 <= g);
}
```

0.4.3 Operadores de incremento y decremento

- Permiten incrementar y decrementar el valor de una variable
  - ++ : incrementa en una unidad
  - −− : decrementa en una unidad

- Tienen dos formas de uso
  - pre-incremento/pre-decremento.
    La variable se incrementa/decrementa antes de evaluar la expresión
  - post-incremento/post-decremento.
    La variable se incrementa/decrementa después de evaluar la expresión

- Ejemplo

```
void main( )
{
  int x = 1, y, z;

  ++x;              // pre-incremento
  x++;              // post-incremento
  y = ++x;
  z = y--;
}
```

¿Cuál es el valor de las variables después de la última asignación?

0.5 Tipos definidos por el programador

- Para definir un tipo se utiliza la palabra reservada *typedef* delante de la declaración de tipo y el nombre del tipo

  `typedef` *declaración_tipo nombre_tipo*

- No existe una palabra reservada para introducir la zona de definiciones de tipo de un programa.

Tipos enumerados

- Los tipos enumerados se declaran con la palabra reservada *enum* seguida de una lista de etiquetas encerradas entre llaves { }

- Las etiquetas de un tipo enumerado son en realidad constantes con valor entero, donde la primera etiqueta toma el valor 0, la segunda el 1, ...

- Ejemplo

```
typedef int TDiaMes;
typedef enum {lun, mar, mie, jue, vie, sab, dom} TDiaSemana;

void main( )
{
  TDiaMes diaMes = 1;
  TDiaSemana diaSemana = lun;

  diaMes += 2;
  diaSemana += 2;
}
```

Registros

- Los registros se declaran con la palabra reservada *struct* seguida de la lista de declaración de los campos encerrada entre llaves { } y donde los campos se separan por ;

```
struct {
  nombre_tipo1 nombre_campo1;
  ...
  nombre_tipoN nombre_campoN;
} nombre_tipo
```

- Ejemplo

```
typedef int TDiaMes;
typedef enum {lun, mar, mie, jue, vie, sap, dom} TDiaSemana;
typedef struct {
  TDiaMes diaMes;
  TDiaSemana diaSemana;
} TDia;

void main( )
{
  TDia dia;

  dia.diaMes = 1;
  dia.diaSemana = lun;
}
```

Arrays

- Para cualquier tipo *T* se puede declarar el tipo *T[num]* que representa a los vectores de *num* elementos de tipo *T*, con índices en el intervalo 0 .. *num* –1
  - *num* ha de ser una expresión constante

- También es posible declarar una variable de tipo array sin necesidad de definir un nuevo tipo, con la siguiente sintaxis

  ```
  nombre_tipo identificador[num];
  ```

- Los arrays se pueden inicializar en la declaración proporcionando una lista de valores, separados por comas y encerrados entre llaves
  - Si la lista de inicialización contiene menos elementos que la dimensión del array el resto de posiciones se inicializan a 0

- Los vectores multidimensionales se declaran como arrays de arrays

- Ejemplo

```
typedef int TMatriz[2][2];

void main( )
{
  int lista[5] = {0, 1, 2, 3, 4};
  TMatriz matriz = { {1,1}, {2,2} };

  matriz[0][0] = lista[0];
  cout << matriz[0][0] << matriz[0][1] << matriz[1][0] \
       << matriz[1][1] << "\n";
}
```

¿qué se mostraría por pantalla?

0.6 Estructuras de control

Selección condicional

- Sintaxis

```
if ( condición ) sentencia

if ( condición ) sentencia else sentencia
```

- Ejemplos

```
int x, y;

if (x == y)
  x = y++;
```

```
int x, y, z;

if (x == y)
{ x = y++;
  z = ++x; }
else
  z = x + y;
```

```
int x, y, z;

if (x == y)
  z = 2;
else if (x > y)
{ z = 1;
  x++;   }
else
{ z = 0;
  y++;   }
```

Selección múltiple

- Sintaxis

```
switch ( expresión )
  case expresión-constante : sentencia
  ...
  default : sentencia
```

  - Se compara secuencialmente la *expresión* con las *expresiones-constantes* hasta encontrar una que coincida. A partir de ahí, se ejecutan todas las *sentencias* hasta llegar al final o hasta encontrar una sentencia *break*

  - Si ninguna *expresión constante* coincide con el valor de *expresión*, se ejecuta la *sentencia* de la cláusula *default*, si es que la hay

- Ejemplo

```
int x = 1, y;

switch (x)
{ case 1:
    y++;
    break;
  case 2:
    x++;
    break;
  default:
    x++; y++;
    break;
}
```

Composición iterativa

- Sentencia *while*

  while (*condición*) *sentencia*

  - Ejemplo:

```
int x = 10, y = 1;

while (x > y)
{ x--;
  y++; }
```

- Sentencia *do .. while*

  do *sentencia* **while** (*condición*);

  - Ejemplo:

```
int x = 10, y = 1;

do
{ x--;
  y++; }
while (x > y);
```

- Sentencia *for*

  for (*inicialización*$_{opcional}$; *condición*$_{opcional}$; *expresión*$_{opcional}$)
  sentencia

  - Se ejecuta la *inicialización* y mientras la *condición* sea cierta, se ejecuta la *sentencia* y a continuación la *expresión*
  - Es posible incluir varias *inicializaciones*, *condiciones* y/o *expresiones* separándolas por comas

- Ejemplos
  - Este fragmento de código

```
int a[10], indice;

for (indice = 0; indice < 10; indice++)
  a[indice] = 2*indice;
```

    es equivalente a este otro

```
int a[10], indice;

indice = 0;
while ( indice < 10 )
{ a[indice] = 2*indice;
  indice++;             }
```

  - Ejemplo de expresiones compuestas

```
int a[10], indice, valor;

for (indice = 0, valor = 9; indice < 10; indice++, valor--)
  a[indice] = valor;
```

0.7 Procedimientos y funciones

- Sintaxis

```
tipo-resultado nombre ( lista-parámetros )
{
    lista-de-sentencias
}
```

- El resultado de una función se devuelve con la sentencia

```
return expresión;
```

que además termina la ejecución de la función

- Un procedimiento es una función cuyo tipo de resultado es *void*

- Ejemplo

```
int suma ( int x, int y )
{
  return x + y;
}

void main( )
{
  int x = suma(2, 3);
}
```

Paso de parámetros

- Por defecto los parámetros se pasan por valor.

  Para indicar que un parámetro formal es por referencia, se escribe el carácter **&** detrás del nombre del tipo

  Los arrays, por razones de eficiencia, se pasan siempre por referencia

- Es posible utilizar como parámetro *arrays abiertos* con la sintaxis

  nombre-tipo identificador [ ]

  Así se pueden escribir funciones que pueden recibir un array de cualquier longitud. Para que la función conozca la longitud del array que se pasa como parámetro real es habitual pasar un segundo argumento con la longitud.

- También es posible definir valores por defecto para los argumentos

  Los argumentos con valor por defecto deben aparecer siempre al final de la lista de argumentos

- Ejemplo

```
void acumula ( int vector[], int longitud, int& suma )
{
  int indice;

  for( indice = 0; indice < longitud; indice++ )
    suma += vector[indice];
}

int suma ( int x, int y = 1 )
{
  return x + y;
}

void main( )
{
  int v[5] = {0, 1, 2, 3, 4};
  int s = 0;

  acumula( v, suma(4), s);
}
```

0.8 Entrada y salida

- Una de las posibles formas de realizar entrada/salida en C++ es por medio de los *flujos* o *canales –streams–*
  - En el módulo *iostream* se incluyen las funciones e identificadores estándar para el manejo de canales

- En iostream se definen tres canales predefinidos
  - *cin*, para entrada de datos
  - *cout*, para salida de datos
  - *cerror*, para salida de errores

- Los canales se manejan con los operadores de *inserción* y *extracción*
  - <<, para enviar datos a un canal
  - >>, para obtener datos de un canal

- Los operadores importados de *iostream* saben cómo manejar datos de los tipos predefinidos

- Ejemplo

```
void pesado ( )
{
  const int magico = 123;
  int intento;
  char c;

  cout << "Adivina el número mágico: ";
  cin >> intento;
  while ( intento != magico ) {
    cout << "\nTe equivocaste, " << intento << " no es correcto" \
         << "\nVuelve a intentarlo: ";
    cin >> intento;
  }
  cout << "¡¡Acetate!!";
  cout << "\nPulsa una tecla para continuar";
  cin >> c;
}
```