Algoritmos y Estructuras de Datos
Ingeniería en Informática, Curso 2º

SEMINARIO DE C++
Sesión 1

Contenidos:

1. Características generales de C++
2. Entrada/salida estándar
3. Variables y tipos de datos
4. Funciones
5. Memoria dinámica

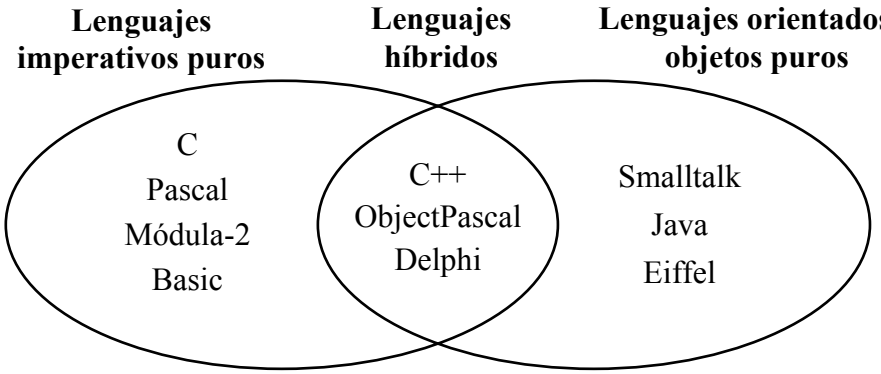
Ejercicios

1. Características generales de C++

- **C++ es un superconjunto de C** → Cualquier programa C (o casi cualquiera) es también un programa C++.

- C++ es una extensión de C para incluir **programación orientada a objetos**, así como **otras mejoras menores** de C.

- ¿Dónde está colocado C++ en el mundo de los lenguajes de programación?



- Pero, **¿en qué consiste la programación orientada a objetos?**
- **Idea sencilla:** una **clase** es como un tipo registro que incluye datos y funciones. Un **objeto** es una variable de ese tipo.

En C	En C++

`struct pila {`
`  int tope;`
`  int datos[MAX_CAPACIDAD];`
`};`
 	
`void vaciar (pila *p);`
`void push (pila *p, int v);`
`void pop (pila *p);`
`int top (pila *p);`
	`class pila {`
	`  private:`
	`  int tope;`
	`  int datos[MAX_CAPACIDAD];`
	`  public:`
	`  void vaciar (void);`
	`  void push (int v);`
	`  void pop (void);`
	`  int top (void);`
	`};`

`struct pila p;`
`vaciar(p);`
`push(p, 16);`
`push(p, 12);`
`pop(p);`
	`pila p;`
	`p.vaciar();`
	`p.push(16);`
	`p.push(12);`
	`p.pop();`

- **Naturaleza dual de las clases:**
  - Una clase es un **tipo abstracto de datos** (mecanismo para crear nuevos tipos definidos por el usuario).
  - Una clase es un **módulo** que proporciona **encapsulación** (agrupa funciones y datos relacionados) y **ocultamiento** de la implementación (parte `private` y parte `public`).

- **Modificaciones menores de C++ respecto a C.**
  - **Extensión** de los ficheros de código: **.cpp** (también .cc, .C, .Cpp ...)
  - **Extensión** de los ficheros de cabecera: **.hpp** (o sin extensión)
  - **Compilación**: **c++ (o g++)**
  - **Comentarios en una sola línea**: `// Comentario`

```
/* Esto es un comentario de C
   que se puede extender varias líneas */

// Y esto es un comentario de C++
// que se extiende hasta fin de línea
```

2. Entrada/salida estándar

- **Igual** que en C, y también...
- La librería **`iostream`** añade un nuevo estilo para la lectura y escritura en entrada y salida estándar.
  Variables:
  - **`cin`** . Flujo de entrada estándar.
  - **`cout`**. Flujo de salida estándar.
  - **`cerr`**. Flujo de error estándar.

  Operadores:
  - **`cin >> variable`**. Leer un valor de teclado y escribir en `variable`.
  - **`cout << expresion`**. Escribir la expresión en la salida estándar.
  - **`cerr << expresion`**. Igual, con la salida de error estándar.

```
#include <iostream>        // Ojo: notar que va sin .h

int main ()
{
   int numero= 1;          // Esta inicialización es irrelevante
   char nombre[80];
   cout << "¿Como te llamas? ";
   cin >> nombre;
   cout << "Hola " << nombre << '\n';  // Se pueden poner varios <<
   cout << "Un número: ";
   cin >> numero;
   cout << "El C++ es " << numero << " veces mejor que el C.\n";
    /* Añadir por aquí */
}
```

  - **`cin.getline(char *res, int max)`**. Leer una línea de teclado, hasta `'\n'` o hasta completar `max-1` caracteres. (Esta forma es mejor con `char *`, porque el `cin >> cadena` se puede desbordar.)

3. Variables y tipos de datos

- A diferencia de C, existe un tipo de datos **booleano**: **`bool`**, con los valores **`true`** y **`false`**. También los operadores **`and`**, **`or`**, **`not`** y **`xor`** (ojo, en algunas versiones no están definidos).

```
 /* Añadir por aquí */
bool pequeno= numero<4, grande= numero>3;
if (grande && ! pequeno)  //Probar:(grande and not pequeno)
   cout << "Estoy de acuerdo.\n";
else
   cout << "Muy mal!!\n";
```

- Se pueden **declarar variables en cualquier punto** del código.

```
int main (void) {
   printf("Esto está en C.\n ");
   int i= 3;
   cout << "i vale: " << i;
   for (int k= 0; k<100; k++) {
      int p;
      cout << k << '\a';
   }
}
```

- **Declaración de constantes: modificador `const`.** El compilador impide las asignaciones sobre variables de tipo `const`. Deben inicializarse siempre.

```
const double pi= 3.1415926;
const int edad= 99;
const char letra;        // ERROR, debe inicializarse
edad++;                  // ERROR, edad es una constante
```

  - En general, se recomienda usar constantes `const` en lugar de `#define`.

  - **Constantes de tipo puntero.**

    `const char *nombre= "Pepito";` → El valor apuntado es constante aunque el puntero sí puede cambiar.

    `char * const nombre= "Pepito";` → El puntero no puede cambiar aunque el contenido sí puede.

    `const char *const nombre= "Pepito";` → El puntero no puede cambiar y el contenido tampoco.

  - **Parámetros constantes** en funciones.

    ```
    char * strcpy (char * destino, const char *origen)
    ```
    → La función no va a cambiar el contenido de la cadena `origen`

- Los nombres de las estructuras y las uniones son tipos de datos.

`struct persona {...};` En C: `struct persona p1;` En C++: `persona p1;`

- **Casting con notación funcional**.

```
double a;
a= (double) 4;  // Esto es C (o C++)
a= double(4);   // Esto es C++
```

- **Operador de resolución de visibilidad '::'**. Un mismo nombre de variable puede aparecer en distintos ámbitos (por ejemplo, variable global y local a una función). Este operador permite acceder a la variable global desde la función.

```
double a= 1.0;
main ()
{
  int a= 4;
  ::a= 3.0;          // Acceso a la variable global
  a= 7;              // Acceso a la variable local
}
```

- **Espacios de nombres**. En lugar de declarar todos los datos y funciones como globales, es posible definir espacios de nombres.

```
#include <string>    // Contiene la clase string, alternativa
                     // a los (char *) de C
namespace GGM
{
   int edad;
   string nombre= "Gines";
}
main ()
{
  cin >> GGM::edad;     // Se debe añadir el prefijo GGM::
  cout << GGM::nombre << ' ' << GGM::edad;
}
```

- **using namespace**. Usar un espacio si necesidad del prefijo `GGM::` ...

```
using namespace GGM;
main ()
{
  cout << nombre << ' ' << edad;
}
```

- **Ojo**. En algunas implementaciones `cout`, `cin` y `cerr` están definidos en el espacio de nombres **`std`** .
  - Tenemos que poner **`std::cout`**, **`std::cin`**, **`std::cerr`**
  - O bien...

```
#include <iostream>
using namespace std;
int main ()
{
   cout << "El using namespace puede ser necesario.\n";
   int valor;
   cin >> valor;
}
```

4. Funciones

- Igual que en C.

- **Principales novedades:** paso de parámetros por referencia, parámetros por defecto y sobrecarga de funciones.

- **Paso de parámetros por referencia**: `(..., tipo & nombre, ...)`

```
void suma (int a, int b, int &c)
{
   c= a + b;
}
```

- También es posible declarar **variables de tipo referencia**. Sólo se puede asignar un valor en la declaración. No se usa el operador `*`.

```
int i= 1;
int &r= i;       // r referencia al mismo sitio que i
r++;
i++;
cout << r;
```

- **Parámetros por defecto**: si no aparece un parámetro en la llamada, tomar un valor por defecto.

```
tipoDevuelto nombreFuncion (tipo1 nombre1, tipo2 nombre2, ...,
                        tipon-1 nombren-1= valorn-1, tipon nombren= valorn)
```

  - Los parámetros por defecto deben ser los últimos en la lista de parámetros de la función.
  - En la llamada se pueden poner algunos y otros no. Se rellenan siempre de izquierda a derecha.

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

void defectuosa (int a, int b, int c= 1, int d= 2, int e= 6)
{
   cout<<a<<' '<<b<<' '<<c<<' '<<d<<' '<<e<<'\n';
}

int main(void)
{
   defectuosa(10, 20);
   defectuosa(10, 20, 30);
   defectuosa(10, 20, 30, 40);
   defectuosa(10, 20, 30, 40, 50);
}
```

- **Sobrecarga de funciones.** Dos, o más, funciones distintas pueden tener el mismo nombre. Es posible si los tipos de los parámetros son distintos.

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

#include <stdlib.h>

void saludar (void)
{
   cout << "Bienvenido, señor desconocido.";
}

void saludar (char *nombre)
{
   cout << "¡Hola " << nombre << "!";
}

void saludar (int n)
{
   cout << "Te saludo " << n << " veces.";
}

int main (int narg, char *cad[])
{
   if (narg==1)
      saludar();
   else if (atoi(cad[1]))
      saludar(atoi(cad[1]));
   else
      saludar(cad[1]);
}
```

5. Memoria dinámica

- **Igual** que en C, aunque también una forma nueva... y mejor...

- **Operadores: `new`**, **`new[]`**, **`delete`** y **`delete[].`**

- **Operador: `new` *`tipo`***. Crea una nueva variable de tipo *`tipo`*. Devuelve un puntero a la variable creada.

```
int *p1= new int;

struct persona {....};
persona *p2= new persona;
```

- **Operador: `new` *`tipo`*`[`*`tamaño`*`]`**. Crea un array dinámico de tipo *`tipo`* y de *`tamaño`* elementos (el primero será el 0). Devuelve un puntero al inicio del array creado.
  - No hace falta indicar el tamaño del tipo de datos. El número de elementos del array (*`tamaño`*) no tiene por qué ser constante.
  - No se inicializa la memoria.
  - Si falla provoca una excepción.
  - Mejor que `malloc` y `calloc`.

```
struct persona {....};

int cuantas;
cin >> cuantas;
int *p3= new int[cuantas];
persona *p4= new persona[cuantas*2];
```

- **Operador: `delete` *`puntero`***. Borrar una variable apuntada por *`puntero`*.

```
int *p1= new int;
delete p1;
persona *p2= new persona;
delete p2;
```

- **Operador: `delete []` *`puntero`***. Borrar un array dinámico apuntado por *`puntero`*. No hace falta indicar el número de elementos a borrar (se calcula automáticamente).

```
int cuantas;
cin >> cuantas;
int *p1= new int[cuantas];
delete [] p1;
persona *p2= new persona[cuantas*2];
delete [] p2;
```

- En general, se recomienda usar **`new`** y **`delete`** en lugar de **`malloc`** y **`free`**.

- **Ejemplo.** Reservar memoria para una matriz de tamaño **n** x **m**. Devuelve un booleano indicando si se ha podido reservar la memoria o no.

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

bool crearMatriz (const int n, const int m, double **&mat)
{
   mat= new double*[n];
   if (!mat)
      return false;
   for (int i= 0; i<n; i++) {
      mat[i]= new double[m];
      if (!mat[i])
         return false;
   }
   return true;
}

void borrarMatriz (int n, int m, double **&mat)
{
   for (int i= 0; i<n; i++)
      delete [] mat[i];
   delete [] mat;
}

void pintaMatriz (int n, int m, double **mat)
{
   for (int i= 0; i<n; i++) {
      for (int j= 0; j<m; j++)
         cout << mat[i][j] << "  ";
      cout << '\n';
   }
}

main (void)
{
   int n, m;

   cout << "Escribe n y m: ";
   cin >> n >> m;
   double **matrix;
   if (crearMatriz(n, m, matrix)) {
      cout << "Matriz reloading...\n";
      for (int i= 0; i<n; i++)
         matrix[i][0]= 1;
      for (int i= 1; i<n; i++)
         for (int j= 1; j<m; j++)
            matrix[i][j]= matrix[i-1][j-1]+matrix[i-1][j];
      pintaMatriz (n, m, matrix);
   }
   borrarMatriz (n, m, matrix);
};
```

Ejercicios

1. Encontrar los errores en el siguiente código C++. Corregirlos y predecir el resultado antes de ejecutar el programa.

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

int i, j;
char *a= "No cambiar";
const pi= 3.14;

namespace EJE {
   char i[10]= "Vaya lio";
   double j= 1.0;
}

main (void)
{
   const char i= '\a';
   const char *a;
   char * const b= "ya me voy aclarando";
   ::i= 9;
   cout << i << ::i << EJE::j << '\n';
   cin << a;
   EJE::i= "otro";
   ::a= "ya me voy aclarando";
   ::a[0]= 'S';
   a[1]= 'u';
   b[1]= 'o';
   b= ::a;
   a= EJE::i;
   EJE::j= sizeof(EJE::i);
   cout << j << ' ' << i << ' ' << EJE::j << ::a;
   return i;
}
```

2. Escribe un procedimiento `redimesionaMatriz`, que dada una matriz (creada con el programa de la página 9) le asigne otra matriz con dimensiones distintas (pasadas como parámetro). La función deberá incluir un parámetro booleano para indicar si los elementos de la matriz antigua deben guardarse o no. Por defecto, el valor del parámetro será `true`. Modificar el programa para que, usando este procedimiento, se repita varias veces el código del `main` hasta que se introduzca un **n** o **m** igual a 0.

3. ¿Qué diferencias (ventajas e inconvenientes) hay entre los mecanismos de C y C++ utilizados para las siguientes necesidades?
   - Declaración de constantes.
   - Paso de parámetros por referencia en un procedimiento.
   - Reserva de memoria dinámica.
   - Manejo de la entrada/salida estándar.