avante
¿hasta dónde quieres llegar?

Programación con C++

Del 4 al 8 Marzo 2019

1

avante
¿hasta dónde quieres llegar?

ÍNDICE DE CONTENIDOS DEL MÓDULO

2

1.- Introducción al lenguaje C++

avante
¿hasta dónde quieres llegar?

3

avante
¿hasta dónde quieres llegar?

1.- Introducción al lenguaje C++

1.1. Un poco de historia.

- C++ fue desarrollado a partir del lenguaje de programación C.
- El lenguaje fue creado por Bjarne Stroustrup en 1985 y documentado en varios libros suyos.
- El nombre de C++ se debe a Rick Mascitti: carácter evolutivo de C, C con clases.
- Ha sido ampliamente revisado y refinado.
- También se han hecho pequeños cambios para incrementar la compatibilidad con C.
- C++ está estandarizado por la ISO (The International Standards Organization). El estándar original de C ++ se emitió en 1998, una revisión menor en 2003, y una actualización importante, el C ++ 11 (2011), y el estándar actual es C ++ 14. Actualmente, el comité de estándares está trabajando para producir un nuevo estándar, una revisión importante, la C++ 17.

1.2. Características del lenguaje.

- Lenguaje híbrido.
- Ha adoptado todas las características de la programación orientada a objetos (POO).
- Perfectamente válido para la programación estructurada heredada de C.
- Mejora sustancialmente las capacidades de C.
- Potente, eficaz y flexible.
- Es un estandar dentro de los lenguajes de POO.

4

1.- Introducción al lenguaje C++

1.3. Algunos tecnicismos.

- **Fichero fuente**: un fichero que contiene programas en C++ en forma de texto.
- **Código fuente**: el texto contenido en el fichero fuente.
- **Fichero objeto**: fichero equivalente a nuestro programa fuente, comprensible para el ordenador.
- **Código objeto**: el contenido del fichero objeto (no tiene significado para los seres humanos).
- **Precompilador**: Programa que fusiona el fichero de código fuente y los ficheros de cabecera, resuelve las directivas y genera un nuevo fichero de código fuente listo para compilar.
- **Compilador**: programa que lee un fichero fuente y genera un fichero objeto.
- **Librería**: contiene el código objeto de operaciones comunes que se puede compartir por muchos programas.
- **Linker**: toma todos los ficheros objeto que componen nuestro programa y crea un fichero ejecutable.
- **Fichero ejecutable**: contiene código binario que es entendible perfectamente por el ordenador.
- **Pila**: zona de memoria usada para que el programa intercambie datos con otros programas o con otras partes del propio programa.

1.- Introducción al lenguaje C++

1.4. Fases de desarrollo de un programa en C++.

- Tomamos un editor de texto y creamos uno o varios ficheros fuente.
- Compilamos todos los ficheros fuente: obtenemos los ficheros objeto.
- El linker coge todos los ficheros objeto combinados con las librerías y genera el fichero ejecutable.
- Ejecutamos el programa: si no hace lo que debería, tenemos que depurarlo.

1.5. Tipos de errores.

Errores de sintaxis: errores en el código fuente. Expresiones erróneas o incompletas. En caso de que existan los revelará el compilador.
Warnings: son avisos de que algo no es completamente correcto, pero no lo suficientemente graves como para impedir la generación del código objeto.
Errores de enlazado: errores en funciones, librerías inexistentes, ficheros objeto mal escritos o ausentes, etc.
Errores de ejecución: el programa finalizará bruscamente. Son los más difíciles de detectar y corregir. Los depuradores o debuggers nos ayudan a detectarlos.
Errores de diseño: si nos hemos equivocado al diseñar nuestro algoritmo, no habrá ningún programa que nos pueda ayudar a corregirlo. Sólo se evitan con la experiencia.

1.- Introducción al lenguaje C++

1.6. Ficheros de cabecera.

El C++ es un lenguaje fuertemente tipado y el compilador realiza una comprobación de tipos antes de generar el código objeto.

Dado que en el momento de la compilación puede no disponer de todos los tipos para realizar la comprobación se debe apoyar en los ficheros de cabecera.

Los ficheros de cabecera contienen los prototipos y definiciones necesarias para realizar la comprobación de tipo.

Por lo tanto, el código fuente se divide en dos tipos de ficheros:

- Ficheros de cabecera:
  - Tienen extensión .h (o .hpp o .hxx)
  - Se incluyen el fichero de código fuente mediante la directiva #include
  - Deben contener: comandos del preprocesador, declaraciones de funciones, definiciones de estructuras, declaraciones externas
  - No deben contener sentencias ejecutables que no sean declaraciones.
- Ficheros de código fuente:
  - Tienen extensión .c o .cpp (o .cc o .cxx o .c++)
  - Incluyen los ficheros de cabecera necesarios
  - Contienen código fuente que implementan los prototipos de las cabeceras.

1.- Introducción al lenguaje C++

1.7. Propósito de C++.

- ¿Qué clase de programas y aplicaciones se pueden crear usando C++? TODOS

- C++ sirve para todo: S.O., compiladores, BBDD, procesadores de texto, juegos, …

- La leyenda de los largos listados de código C++ frente a Visual Basic y otros.

- Crear librerías reutilizables SIEMPRE: créala una vez y úsala mil veces.

- Generan los programas más rápidos y compactos. (Salvo ENSAMBLADOR)

- C/C++ tienen más de 30 años de vida y su uso no se debilita.

- Unix, GNU/Linux o Windows se escriben casi por completo en C.

- Miles de programadores en todo el mundo se dedican exclusivamente a crear librerías C/C++.

1.- Introducción al lenguaje C++

1.8. El moderno C++

Ámbito de memoria basado en pila en lugar de ámbito de montón o global estático.

Inferencia de tipos automática en lugar de nombres de tipo explícitos.

Punteros inteligentes en lugar de punteros sin formato.

Los tipos std::string y std::wstring en lugar de matrices char[].

Biblioteca estándar de C++ con colecciones (vector, list y map) en lugar de matrices sin formato o contenedores personalizados.

Biblioteca estándar de algoritmos C++ en lugar de la forma manual los codificados.

Excepciones, para notificar y controlar errores.

La comunicación entre subprocesos mediante la biblioteca estándar de bloqueos C++ con std::atomic<> en lugar de otros mecanismos de comunicación entre subprocesos.

Funciones lambda en lugar de pequeñas funciones Inline implementadas por separado.

Funciones de rangos para bucles para poder escribir bucles más eficaces que funcionan con matrices, contenedores de la biblioteca estándar de C++ y colecciones.

1.- Introducción al lenguaje C++

1.8. Sintaxis

Sensible a mayúsculas y minúsculas.

Sentencias separadas por punto y coma (;).

Formato libre en las sentencias, una sentencia puede utilizar varias líneas y una línea puede incluir varias sentencias.

Los espacios en blanco se compactan a uno solo, las marcas de identación se ignoran.

Bloques marcados por llaves:

```
{ <sentencias> }
```

Comentarios:

```
/* una o varias líneas */
// Hasta el final de la línea
```

Directivas al precompilador:

```
# toda la línea
```

61 palabras reservadas, todas en minúsculas.

Introducción al lenguaje C++.

1.9. Identificadores

- Para crear un identificador hay que tener en cuenta algunas reglas:
  - Sólo se pueden usar letras, números, _ y $.
  - Se usa el juego de caracteres ANSI o ASCII (sin ñ ni letras tildadas).
  - El primer carácter no puede ser un número.
  - Palabra única, no pueden contener espacios en blanco.
  - Son sensibles a mayúsculas y minúsculas.
  - Se recomienda la notación Pascal (VariasPalabras) o Camel (variasPalabras).
  - NO debe tener el mismo nombre que otro elemento del mismo ámbito (repetirse).
  - No pueden llamarse igual que alguna de las palabras reservadas:

asm	auto	bool	break	case	catch
char	class	const	const_cast	continue	default
delete	do	double	dynamic_cast	else	enum
explicit	extern	false	float	for	friend
goto	if	inline	int	long	mutable
namespace	new	operator	private	protected	public
register	reinterpret_cast	return	short	signed	sizeof
static	static_cast	struct	switch	template	this
throw	true	try	typedef	typeid	typename
union	unsigned	using	virtual	void	volatile
while					

2.- Variables, operadores, constantes y expresiones.

2.- Variables, operadores, constantes y expresiones.

2.1. Tipos de datos.

- Una variable es un nombre simbólico que le damos a una zona de memoria.
- La usaremos para almacenar y recuperar datos de distinto tipo.
- Ocupará más o menos espacio en memoria dependiendo del **tipo de dato** que almacene.

Tipo de dato	*Tamaño*	*Intervalo de valores posibles*
`short`	2 bytes	-32768 a +32767
`unsigned short`	2 bytes	0 a +65535
`long`	4 bytes	-2147483648 a +2147483647
`unsigned long`	4 bytes	0 a +4294967295
`int`	2/4 bytes	-32768 a +32767
`unsigned int`	2/4 bytes	0 a +65535
`float`	4 bytes	-1.17549535e-38 a +3.402823466e+38
`double`	8 bytes	-2.2250738585072014e-308 a +1.7976931348623158e+308
`long double`	10 bytes	-2.2250738585072014e-308 a +1.7976931348623158e+308
`char`	1 byte	-128 a 127
`unsigned char`	1 byte	0 a +255
`bool`	1 byte	true ó false
`"complejos"`	tipos de datos definidos por el usuario como combinación de los anteriores	

2.- Variables, operadores, constantes y expresiones.

2.2. Declaración y asignación de variables.

Se declaran como:

<tipo> <nombre> [= <valor inicial>];

Pueden declararse varias a la vez del mismo tipo separando los nombres por comas.
Se deben inicializar antes de consultar.
Se pueden declarar en cualquier punto, pero siempre antes de utilizarlas. Se recomienda definirlas al principio del bloque.
Las definidas en bloques interiores solapan a las de bloques superiores con el mismo nombre.
Alcance: desde donde se definen hasta el final del bloque.
En general las variables se asignan de la forma:

<variable> = {<valor>|<variable>};

Los valores constantes se expresan:

- Enteros
  - Base 10 dddd...
  - Base 8 0ddd
  - Base 16 0xddd
- Punto Flotante
  - estándar 3.14159
  - científica 6.022E23
- Sufijos: L, l (long) F, f (double)
- Caracteres
  - Apostrofes: 'a', '\n', '\\' ...
- Cadenas
  - Comillas: "..."
- Sin referencia
  - NULL

2.- Variables, operadores, constantes y expresiones.

2.2. Declaración y asignación de variables.

Modificadores:

- **auto**: (automáticas) son las variables normales y que el compilador toma por defecto.
- **register**: El compilador intenta situar la variable en uno de los registros de la CPU, con lo que el acceso es mucho mas rápido que si estuvieran en memoria. Solo pueden ser locales.
- **static**: Son variables locales que no pierden su valor aunque termine el bloque que las creo. Solo se inicializan la primera vez que se ejecute el bloque que las crea. Estas variables se alojan en una parte especial de la memoria.
- **extern**: Indica al compilador que la variable ya esta declarada en otro punto del programa y, por lo tanto, que no le reserve espacio en memoria.
- **volatile**: Indica al compilador que el valor de la variable puede cambiar en cualquier momento fuera de la secuencia del programa. Por lo tanto, el compilador no debe realizar ningún tipo de optimización con estas variables. Pueden ser locales y globales.

Casting:

- La conversión de un tipo de menor precisión a uno de mayor es automática (promoción).
- Se denomina casting a la operación que fuerza la conversión de restricción (coerción) de un tipo a otro.
- Se realiza:
  (<nuevo tipo>) <expresión>
  <nuevo tipo>(<expresión>)
- La conversión de tipos de mayor precisión a uno de menor implica la pérdida de las posiciones altas.

2.- Variables, operadores, constantes y expresiones.

2.3. Operadores y expresiones.

- **Operando**: cada una de las constantes, variables o expresiones que intervienen en una expresión.

- **Operador**: cada uno de los símbolos que indican las operaciones a realizar sobre los operandos.

- **Expresión**: cualquier conjunto de operadores y operandos, que dan como resultado un valor.

- Clasificamos los operadores según el número de operandos a los que afecta:

  - unarios: afectan a un sólo operando.
  - binarios: afectan a dos operandos.
  - ternarios: afectan a tres operandos.

- A continuación los clasificamos dependiendo del tipo de objeto sobre los que actúan.

2.- Variables, operadores, constantes y expresiones.

2.3. Operadores y expresiones.

2.3.1. Operadores aritméticos.

`+ <expresión>`	Signo positivo (unario)
`- <expresión>`	Signo negativo (unario)
`<expresión> + <expresión>`	Suma
`<expresión> - <expresión>`	Resta
`<expresión> * <expresión>`	Producto
`<expresión> / <expresión>`	División
`<expresión> % <expresión>`	Resto de la división entera
`<variable> ++`	Post-incremento (unario)
`++ <variable>`	Pre-incremento (unario)
`<variable> --`	Post-decremento (unario)
`-- <variable>`	Pre-decremento (unario)

2.- Variables, operadores, constantes y expresiones.

2.3. Operadores y expresiones.

2.3.2. Operadores de asignación.

<variable> <operador de asignación> <expresión>

donde <operador de asignación> puede ser alguno de los siguientes:

`=` `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `^=` `|=`

2.3.3. Operador coma.

- Separa elementos de una lista de argumentos de una función.
- Separa variables del mismo tipo en su declaración.
- Separa elementos en la inicialización de arrays.
- Separa elementos en la declaración de tipos de datos enumerados.
- Otros casos especiales, que veremos en su contexto.

2.- Variables, operadores, constantes y expresiones.

2.3. Operadores y expresiones.

2.3.4. Operadores de comparación.

- Comparan dos operandos, resultando un valor booleano: **`true`** ó **`false`**:

<expresión1> == <expresión2>	igualdad
<expresión1> != <expresión2>	desigualdad
<expresión1> < <expresión2>	menor que
<expresión1> > <expresión2>	mayor que
<expresión1> <= <expresión2>	menor o igual que
<expresión1> >= <expresión2>	mayor o igual que

2.3.5. Operadores lógicos.

- Relacionan operaciones lógicas, resultando una expresión lógica: **`true`** ó **`false`**:

`<expresión1> && <expresión2>`	**`and`** (y)
`<expresión1> || <expresión2>`	**`or`** (o)
`! <expresión>`	**`not`** (no)

2.- Variables, operadores, constantes y expresiones.

2.3. Operadores y expresiones.

2.3.6. Operador sizeof.

Este operador tiene dos usos diferentes:

1. sizeof (<expresión>)
2. sizeof (nombre_de_tipo)

Devuelve una constante entera que indica el tamaño en bytes del espacio de memoria usada por el operando (determinado por su tipo).

2.3.7. Operadores de referencia e indirección.

Referencia: **`& <expresión>`** devuelve la dirección de memoria del operando.
Indirección: **`* <puntero>`** devuelve el contenido de una dirección de memoria.

Los veremos en profundidad cuando estudiemos los Punteros.

2.- Variables, operadores, constantes y expresiones.

2.3. Operadores y expresiones.

2.3.8. Operadores . y ->

Operador . : **<variable>.<nombre>** accede a miembros dentro de una estructura u objeto.
Operador -> : **<puntero>-><nombre>** selecciona miembros dentro de una estructura u objeto referenciadas con punteros.

Los veremos en profundidad cuando estudiemos los Punteros.

2.3.9. Operador de preprocesador.

El operador **#** sirve para dar órdenes o directivas al compilador.

Preprocesador: programa auxiliar, que forma parte del compilador, y que procesa el fichero fuente antes de que sea compilado.

```
#define <identificador_de_macro> <secuencia>
#include <nombre_de_fichero_de_cabecera>
#include "nombre_de_fichero_de_cabecera"
#include identificador_de_macro
```


2.- Variables, operadores, constantes y expresiones.

2.3. Operadores y expresiones.

2.3.10. Operadores de manejo de memoria.

- Operador new: **new <tipo> [(inicialización)]**

  La inicialización, si aparece, se usará para asignar valores iniciales a la memoria reservada.

  La memoria reservada con **new** será válida hasta que se libere con **delete**.

  Si la reserva de memoria no tuvo éxito, **new** devuelve un puntero **NULL**.

- Operador delete: **delete [<expresión>]**

  Se usa para liberar la memoria dinámica reservada con **new**.

  Existe peligro de pérdida de memoria si se ignora esta instrucción.

- Operador de ámbito: <clase>::<miembro>

  Se usa para eliminar la ambigüedad en determinados contextos.

2.- Variables, operadores, constantes y expresiones.

2.3. Operadores y expresiones.

2.3.11. Operadores de bits.

Trabajan con las expresiones que manejan manipulándolas a nivel de bit, y únicamente se pueden aplicar a expresiones enteras. Son los siguientes:

`<expresión>&<expresión>`	Operación lógica AND
`<expresión>^<expresión>`	Operación lógica OR exclusivo (XOR)
`<expresión>|<expresión>`	Operación lógica OR
`~<expresión>`	Operación lógica NOT
`<expresión><<<expresión>`	Desplazamiento de bits a la izda. del valor de la izda.
`<expresión>>><expresión>`	Desplazamiento de bits a la dcha. del valor de la izda.

2.3.12. Operador condicional.

`<E1> ? <E2> : <E3>`	E1, E2, E3: expresiones

Se evalúa E1, si es `true` se evalúa E2, sino se evalúa E3.

2.- Variables, operadores, constantes y expresiones.

2.3. Operadores y expresiones.

2.3.13. Precedencia de operadores.

Operadores	Asociatividad
`() [] -> .`	Izquierda a derecha
`! ~ + - ++ -- & * sizeof new delete`	Derecha a izquierda
`.* ->*`	Izquierda a derecha
`* / %`	Izquierda a derecha
`+ -`	Izquierda a derecha
`<< >>`	Izquierda a derecha
`< <= > >=`	Izquierda a derecha
`== !=`	Izquierda a derecha
`&`	Izquierda a derecha
`^`	Izquierda a derecha
`|`	Izquierda a derecha
`&&`	Izquierda a derecha
`||`	Izquierda a derecha
`?:`	Derecha a izquierda
`= *= /= %= += -= &= ^= |= <<= >>=`	Derecha a izquierda
`,`	Izquierda a derecha

2.- Variables, operadores, constantes y expresiones.

2.4. Sentencias: bucles, selección y salto.

```
if (<condición>)
   <bloque_de_sentencias>
[else <bloque_de_sentencias>]

switch (<expresión_entera>) {
   [case <expresión_constante1>: [<bloque_de_sentencias>[break;]]
   [case <expresión_constante2>: [<bloque_de_sentencias>[break;]]]
   ...
   [case <expresión_constanten>: [<bloque_de_sentencias>[break;]]]
   [default: [<bloque_de_sentencias>]]
}

while (<condición>)                do
   <bloque_de_sentencias>             <bloque_de_sentencias>
                                   while (<condición>);

for ( [<inicialización>]; [<condición>]; [<variación>] )
   <bloque_de_sentencias>

break;                             continue;
```


3.-Tipos de datos estructurados.

3.-Tipos de datos estructurados.

3.1. Arrays.

Un array es una agrupación de datos del mismo tipo, que reciben un nombre común, y al los que se acceden por medio de índices.

3.1.1. Declaración de arrays.

```
<tipo> <identificador>[<número-elementos>][[número-elementos]...];
```

<número-elementos> debe ser una constante entera. NUNCA usar una variable para definir el tamaño de un array (algunos compiladores lo permiten, pero está fuera de la norma).

Los índices varían SIEMPRE de `0` a **<número-de-elementos> - 1**

3.1.2. Arrays unidimensionales.

```
<tipo> <identificador>[<número-elementos>];
```

3.1.3. Arrays multidimensionales.

```
<tipo> <identificador>[<#elementos>][<#elementos>]...[<#elementos>];
```


3.-Tipos de datos estructurados.

3.1. Arrays.

3.1.4. Inicialización de arrays.

```
<tipo> <identificador>[<#elementos>] = {<elto1>,<elto2>,...,<elton>};
<tipo> <identificador>[] = {<elto1>,<elto2>,...,<elton>};
<tipo> <identificador>[<índice>] = <valor>;
```

3.1.5. Asignación de elementos a/de un array.

```
<identificador>[<índice>] = <valor>;
<variable> = <identificador> [<índice>];
```

3.2. Strings.

- Son un tipo especial de arrays: arrays unidimensionales de tipo char terminados en nulo.
- Declaración: **`char <identificador> [<longitud-máxima>];`**
- La asignación directa sólo está permitida junto con la declaración.
- Sí podemos asignar caracteres individuales a posiciones individuales del String.
- El último carácter debe ser \0.
- Librería estándar de C: <string>
- C++ también tiene una biblioteca de clases para String.

3.-Tipos de datos estructurados.

3.3. Estructuras.

- Nos permiten agrupar varios tipos de datos bajo un nombre, permitiendo manipularlos todos juntos.

```
struct [<identificador>] {
   [<tipo> <nombre-campo>[,<nombre-campo>,...]];
} [<objeto-estructura>[,<objeto-estructura>,...]];
```

- Las estructuras pueden referenciarse completas, usando su nombre.

- O podemos acceder a los elementos de ella usando el operador de selección (.) .

- Una vez definida la estructura, podemos crear objetos cuyo tipo sea esta estructura:

```
[struct] <identificador> <objeto-estructura>
   [,<objeto-estructura>...];
```

- Para inicializar una estructura, operamos igual que con arrays, salvo matices.

- Se permite la asignación de estructuras del mismo tipo.

- Aunque no es muy común, las estructuras se pueden anidar.

3.-Tipos de datos estructurados.

3.4. Punteros.

- Cada byte de la memoria de un ordenador tiene una dirección: @memoria.
- La unidad básica de información con la que trabajan los microprocesadores: palabra/word.
- Según el procesador la palabra será de 1, 2, 4, 8 o 16 bytes: plataformas de 8, 16, 32, 64 o 128 bits.
- Cuando las palabras sean de 32 bits accederemos a @memoria con múltiplos de 4.
- Los objetos > 1byte se almacenarán en posiciones consecutivas de memoria y la dirección de la primera posición de memoria será la dirección de ese objeto.
- Un puntero es un tipo especial de variable que contiene un valor entero (en hexadecimal) con la dirección de memoria de un objeto: char, int, float,...
- Al declarar un puntero hay que decirle a qué tipo de objeto va a apuntar.
- El valor inicial del puntero, podría ser una @memoria cualquiera o incluso inexistente.
- NUNCA usar un puntero que no haya sido inicializado correctamente.

3.4.1. Declaración de punteros.

Se declaran igual que el resto de tipos, precediendo el identificador con *

<tipo> *<identificador>;

Existen dos cualificadores para indicar el direccionamiento de la memoria:

- near: puntero cercano, en el mismo segmento de memoria. Solo almacena el desplazamiento dentro del sector de memoria.
- far: puntero lejano, almacena el segmento además del desplazamiento.

3.-Tipos de datos estructurados.

3.4. Punteros.

3.4.2. Uso de punteros.

Al apuntar a un objeto, lo primero que tenemos que conocer es a qué dirección apunta.

```
<tipo> <variable>;
<tipo> *<identificador>;
<identificador> = &<variable>;
```

<identificador> sería un puntero a **<tipo>** que apunta a la dirección donde se almacena el valor de **<variable>**

¿Cómo obtenemos el objeto referenciado por un puntero? Accediendo a ***<identificador>**

Es bueno siempre inicializar los punteros a **NULL**.

NULL es una constante definida en varias librerías como **0**, aunque **0** no significa **NULL**.

31

3.-Tipos de datos estructurados.

3.4. Punteros.

3.4.3. Punteros y arrays.

Existe una relación muy estrecha entre punteros y arrays.

Cuando declaramos un array:

1. Declaramos un puntero del mismo tipo que los elementos del array.
2. Reservamos memoria para todos los elementos del array.
3. Estos elementos se almacenan en posiciones consecutivas de memoria.
4. Se inicializa el puntero a la @memoria del primer elemento del array.

Pero:

1. El identificador del array es un puntero constante, no podemos hacerle apuntar a otra @memoria.
2. El compilador asocia, de forma automática, una zona de memoria para los elementos del array, cosa que no hace para los elementos apuntados por un puntero corriente.

C ++ permite el uso de punteros que apuntan a punteros, que estos, a su vez, apuntan a datos (o incluso a otros punteros). La sintaxis simplemente requiere un asterisco (*) para cada nivel de direccionamiento indirecto en la declaración del puntero:

```
int toArray** = &myArray;
```


32

3.-Tipos de datos estructurados.

3.4. Punteros.

3.4.4. Operaciones con punteros.

- Para asignar un puntero a otro, usamos la asignación normal.
- Si sumamos un entero a un puntero, el puntero saltará a una dirección de memoria calculada en función del tipo de dato al que apunte.
- Si los dos operandos son punteros, sólo tiene sentido la resta: el resultado es un entero.
- No es posible comparar punteros de tipos diferentes, ni aunque sean ambos nulos.
- Es posible declarar punteros sin especificar a qué tipo de objeto apuntan:

```
void *<indentificador>;
```

- Así podemos usarlo para apuntar a distintos tipos de objetos haciendo:

```
(<tipo> *)<identificador>;
```

- Los punteros nos permiten manejar arrays dinámicos con la ayuda de **`new`** y **`delete`**.

3.4.5. Punteros a estructuras.

- Para referirnos a cada elemento de la estructura usamos el operador **`->`** en lugar del **`.`**

`puntero->miembro` equivale a **`(*puntero).miembro`**

3.-Tipos de datos estructurados.

3.5. Uniones.

- Son un tipo de estructuras que permiten almacenar elementos de distintos tipos en las mismas posiciones de memoria, no simultáneamente.

```
union [<identificador>] {
    [<tipo> <nombre-variable>[,<nombre-variable>,...]];
} [<variable-union>[,<variable-union>,...]];
```

- La particularidad de las uniones es que cada elemento se almacenará comenzando en la misma posición de memoria.

- Las uniones se pueden referenciar completas o por elementos, usando el operador **`.`**

- Pueden declararse más objetos del tipo de la unión en cualquier parte:

```
[union] <identificador-de-unión> <variable>[,<variable>...];
```

- Las uniones sólo pueden ser inicializadas en su declaración mediante su primer miembro.

3.-Tipos de datos estructurados.

3.6. Enumeraciones.

- Nos permite definir conjuntos de constantes enteras.
- Las variables declaradas de este tipo, sólo podrán tomar valores definidos en la declaración.

```
enum [<identificador-de-enum>] {
   <nombre> [=<valor>,...]
}<identificador>[,<indentificador2>[,<identificador3>]...];
```

- Varios **<nombre>** pueden tomar el mismo **<valor>**.
- Cada **<identificador>** sólo puede usarse en un tipo enumerado.
- **<nombre>** debe considerarse como "etiquetas" que sustituyen a enteros.
- Para el compilador, internamente, son enteros.
- Sólo se usan para una mejor comprensión de los programas.
- Si no se indica **<valor>** para un **<nombre>** , tomará el **<valor>** del anterior **<nombre>** incrementado en 1.

3.7. Definición de tipos.

- Puede ser útil definir tipos de datos que nos faciliten la tarea de declarar variables.

```
typedef <tipo> <identificador>;
```

- **<tipo>** puede ser cualquier tipo de dato C++, fundamental o derivado.

3.-Tipos de datos estructurados.

3.8. Constantes.

define <nombre_contante> [<valor>]

Con el define indicamos al precompilador que sustituya dentro del código fuente el nombre de la constante por el valor indicado.

const <declaración_de_variable>

Declaramos la constante como una variable, con todas las características de estas salvo la posibilidad de modificar su valor.

El define se utiliza como un nombre nemotécnico que ni siquiera llega a la fase de compilación.

La const se comporta como una variable pero solo reservara espacio de memoria cuando se utilice su dirección (aunque solo sea para pasársela como parámetro).

Una const tiene ámbito global dentro del fichero donde este definida y no se pueden declarar dentro de la definición de una clase.

4.-Funciones.

avante
¿hasta dónde quieres llegar?

37

avante
¿hasta dónde quieres llegar?

4.-Funciones.

- Una función es un conjunto de instrucciones que realizan una tarea específica.
- Toman ciertos valores de entrada: parámetros, y proporcionan un valor de salida o retorno.

4.1. Prototipos de funciones.

- Un prototipo es una declaración adelantada de una función que le damos al compilador:

```
[extern|static] <tipo-valor-retorno> [<modificadores>] <identificador>(<lista-parámetros>);
```

- Los prototipos se escriben antes de la función main.

4.2. Definición de funciones.

La definición contiene además, las instrucciones que forman parte de ella.

```
[extern|static] <tipo-valor-retorno> [modificadores] <identificador>(<lista-parámetros>) {
    [sentencias]
}
```

Esta definición suele hacerse después de la función main.

4.3. Parámetros por valor y por referencia.

- Un parámetro por valor no modifica la variable externa cuando se modifica dentro de la función.
- Un parámetro por referencia modifica la variable externa cuando se modifica dentro de la función.
- Para pasar parámetros por referencia debemos indicarlo en la declaración de la función anteponiendo un & a cada parámetro de la función que queramos pasar por referencia.

38

4.-Funciones.

4.4. Parámetros con valores por defecto.

- Es posible que para ciertos parámetros sus valores se repitan con frecuencia.
- En esos casos usamos parámetros con valores por defecto.
- A esos parámetros les asignamos valor en los prototipos ó en las declaraciones de la función.
- Si en un prototipo hay parámetros con valores por defecto y sin ellos, los parámetros que llevan valores por defecto deben de ser los últimos.

4.5. Funciones con número de argumentos variable.

- Es posible crear funciones con un número indeterminado de argumentos.
- Los parámetros desconocidos se sustituyen por 3 puntos:

```
<tipo-valor-retorno> <identificador>(<lista-parámetros-conocidos>, ...);
```

4.6. Funciones inline.

- Si se declara una función como inline, el compilador sustituye cada vez que se llama a la función por el código de la función.
- No siempre ocurre así, ya que no es una obligación para el compilador, sino una recomendación.

4.-Funciones.

4.7. Punteros a funciones.

- Declaración de un puntero a una función:

```
<tipo> (*<identificador>)(<lista-de-parámetros>);
```

- Mucho cuidado con los `()` y los `*` , porque el significado cambia radicalmente.
- Los punteros a funciones son útiles cuando se personalizan ciertas funciones de biblioteca.
- Podríamos diseñar una función de biblioteca que admita como parámetro una función, que debe crear el usuario, para que la función de biblioteca complete su funcionamiento.
- Una vez declarado este puntero, funciona como una variable cualquiera.
- Al asignar una función a un puntero NO es necesario indicar el `&` antes del nombre de la función.
- Para invocar a la función usando el puntero, sólo hay que usar el identificador del puntero como si se tratase de una función al uso.

```
int (*pFn)(char* str);
// …
pFn = strlen;
int len = (*pFn)("Mi cadena");
```

- A fines prácticos, el puntero se comporta como un alias de la función a la que apunta.

4.-Funciones.

4.8. Sobrecarga de funciones.

- Una función sobrecargada se da cuando definimos varias funciones con el mismo nombre, con la única condición de que el número y/o el tipo de parámetros sean distintos.
- El compilador es el que decide cuál de las versiones existentes usará después de analizar el número y el tipo de los parámetros.
- Si no puede decidir, se aplican las reglas implícitas de conversión de tipos.
- A este proceso se le llama resolución de sobrecarga.

4.9. La función main().

- Las funciones no tienen que seguir un orden determinado en un programa C. El compilador necesita saber cual es la función por la que empezar la ejecución del programa.
- La ejecución empieza por la función main y es obligatorio insertarla una única vez por cada fuente ejecutable.
- La función main() acepta argumentos como cualquier otra función. Los argumentos se le pasan en la línea de comandos. Hay dos argumentos especiales ya incorporados:
  - argc: es un entero que contiene el numero de parámetros, al menos vale 1 ya que el nombre del programa cuenta también.
  - argv: es un puntero a un array de punteros a caracteres que contiene los parámetros de la línea de comandos. Todos los argumentos son cadenas.

4.-Funciones.

4.10. Estructura de un Programa

```
/*
  Descripción del fichero:
        Nombre: (del fichero)
        Autor:
        Fecha:
        Comentario:
*/

#  // Ordenes al precompilador

// Definición de constantes y variables globales.

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

        // Modulo principal

}

// Resto de procedimientos.
```

4.-Funciones.

4.11. Espacios de nombres

Un espacio de nombres es una región declarativa que proporciona un ámbito a los identificadores (nombres de tipos, funciones, variables, etc.) de su interior.
Los espacios de nombres se utilizan para organizar el código en grupos lógicos y para evitar conflictos de nombres que pueden producirse, especialmente cuando la base de código incluye varias bibliotecas.

```
namespace ContosoData {
    class ObjectManager {
    public:
        void DoSomething() {}
    };
    void Func(ObjectManager) {}
}
```

Todos los identificadores del ámbito del espacio de nombres son visibles entre sí sin calificación, fuera del espacio de nombres se requiere el nombre completo para cada identificador:

```
ContosoData::ObjectManager
```

La directiva using permite utilizar todos los identificadores de un espacio de nombres sin tener que cualificarlos con el nombre de su espacio de nombres.

```
using namespace ContosoData;
```

Todos los tipos de la biblioteca estándar de C++ y las funciones se declaran en el espacio de nombres std o espacios de nombres anidado dentro de std.

4.-Funciones.

Directivas del preprocesador: #define

#define <nombre de macro> <nombre de reemplazo>

Permite configurar el lenguaje al reemplazar un identificador en todas sus apariciones en el código fuente por un texto:

```
#define FALSO 0
#define VERDADERO !FALSO
#define AND &&
#define OR ||
```

El nombre de macro puede tener argumentos, cada vez que el precompilador encuentra el nombre de macro, los argumentos reales encontrados en el programa reemplazan los argumentos asociados con el nombre de la macro.

```
#define MIN(a,b) (a < b) ? a : b
```

Directivas del preprocesador: #undef

Se usa #undef para quitar una definición de nombre de macro que se haya definido previamente.
El formato general es:

#undef <nombre de macro>

El uso principal de #undef es permitir localizar los nombres de macros sólo en las secciones de código que los necesiten.

4.-Funciones.

Directivas del preprocesador: Condicional

Los comandos de compilación condicional permiten que se compilen selectivamente partes del código fuente.

```
#define PAIS_ACTIVO EU
#if PAIS_ACTIVO == JAP
char moneda[]="yen";
#elif PAIS_ACTIVO == EU
char moneda[]="euro";
#else
char moneda[]="dolar";
#endif
```

La condición puede ser la existencia o inexistencia de una definición previa o paso de parámetros al compilador:

```
#ifdef DEBUG
…
#else
…
#endif
#ifndef MI_LIB
…
#define MI_LIB ok
#endif
```


4.-Funciones.

Directivas del preprocesador: #include

La directiva del preprocesador #include instruye al precompilador para incluir otro archivo fuente que esta dado con esta directiva y de esta forma compilar otro archivo fuente.

El archivo fuente que se leerá se debe encerrar entre comillas dobles o paréntesis de ángulo:

```
#include <archivo>
#include "archivo"
```

Cuando se indica <archivo> se le dice al precompilador que los busque donde están los archivos de cabecera estándar del sistema.

Si se usa la forma "archivo" se busca en el directorio actual, es decir, donde el programa esta siendo ejecutado.

En C las extensión .h es obligatoria, en C++ es opcional.

4.-Funciones.

Standard headers

<assert.h>	<stdio.h>	<signal.h>
<locale.h>	<errno.h>	<string.h>
<stddef.h>	<setjmp.h>	<limits.h>
<ctype.h>	<stdlib.h>	<stdarg.h>
<math.h>	<float.h>	<time.h>

Conversiones

<CTYPE.H> <cctype>

```
int     isalnum(int c);
int     islower(int c);
int     isalpha   (int c);
int     isprint   (int c);
int     isascii   (int c);
int     ispunct(int c);
int     iscntrl   (int c);
int     isspace(int c);
int     isdigit   (int c);
int     isupper(int c);
int     isgraph(int c);
int     isxdigit  (int c);

int     tolower(int ch);
int     toupper(int ch);
```

<STDLIB.H> <cstdlib>

```
int     atoi(const char *s);
long    atol(const char *s);
double  atof(const char *s);
int     rand (void);
```


4.-Funciones.

Entrada/salida

<STDIO.H> <cstdio>

```
FILE    *fopen    (const char *filename, const char *mode);
int     fclose    (FILE *stream);
int     feof      (FILE *stream);
size_t  fread     (void *ptr, size_t size, size_t n, FILE *stream);
size_t  fwrite    (const void *ptr, size_t size, size_t n, FILE *stream);
int     fseek     (FILE *stream, long offset, int whence);
void    rewind    (FILE *stream);

int     fgetc     (FILE *stream);
int     fputc     (int c, FILE *stream);
int     getc      (FILE *stream);
int     putc      (int c, FILE *stream);

char    *gets     (char *s);
int     puts      (const char *s);
char    *fgets    (char *s, int n, FILE *stream);
int     fputs     (const char *s, FILE *stream);
```

4.-Funciones.

Entrada/salida con formato

<STDIO.H> <cstdio>

```
int     fscanf    (FILE *stream,  const char *format [,address, ...]);
int     scanf     (const char *format [, address,...]);
int     sscanf    (const char *buffer, const char *format[, address, ...]);
int     vfscanf   (FILE *stream,  const char *format, va_list arglist);
int     vscanf    (const char *format, va_list arglist);
int     vsscanf   (const char *buffer, const char *format, va_list arglist);

int     fprintf   (FILE *stream, const char *format [,argument, ...]);
int     printf    (const char *format [, argument,...]);
int     sprintf   (char *buffer, const char *format [,argument, ...]);
int     vfprintf  (FILE *stream, const char *format,va_list arglist);
int     vprintf   (const char *format, va_list arglist);
int     vsprintf  (char *buffer, const char *format, va_list arglist);
```


4.-Funciones.

Cadenas

<STRING.H> <cstring>

```
size_t    strlen    (const char *s);
char      *strcpy   (char *dest, const char *src);
char      *strcat   (char *dest, const char *src);
int       strcmp    (const char *s1, const char*s2);
                    <0 s1<s2, 0 s1==s2, >0 s1>s2
char      *strchr   (const char *s, int c);
char      *strstr   (const char *s1, const char *s2);
```

Memoria

<STDLIB.H> <cstdlib>

```
void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);

void abort (void);
void exit    (int status);
```

5.- Asignación dinámica de memoria.

avante
¿hasta dónde quieres llegar?

51

avante
¿hasta dónde quieres llegar?

5.- Asignación dinámica de memoria.

El C++ permite crear tipos de datos y estructuras de cualquier tamaño de acuerdo a las necesidades que se tengan en el programa.

La función malloc es empleada comúnmente para intentar obtener una porción contigua de memoria:

```
void *malloc(size_t size);
```

Devuelve un apuntador del tipo void (puede ser convertido a cualquier tipo), con el inicio en memoria de la porción reservada de tamaño size. Si no puede reservar esa cantidad de memoria la función regresa un apuntador nulo o NULL.

El operador sizeof() puede ser usado para encontrar el tamaño de cualquier tipo de dato, variable o estructura:

```
dim = (miTipo *)malloc(sizeof(miTipo));
```

La función free() toma un apuntador como argumento y libera la memoria a la cual el apuntador hace referencia.

52

5.- Asignación dinámica de memoria.

Cuando se usa la función malloc() la memoria no es inicializada (a cero) o borrada. La función calloc inicializa la memoria y permite indicar el número de elementos deseados (nmemb) y el tamaño del elemento (size), como dos argumentos individuales.

```
void *calloc(size_t nmemb, size_t size);
```

La función realloc intenta cambiar el tamaño de un bloque de memoria previamente asignado. Si el bloque se hace más grande, entonces el contenido anterior permanece sin cambios y la memoria es agregada al final del bloque.

Si el tamaño se hace más pequeño entonces el contenido sobrante permanece sin cambios.

Si el tamaño del bloque original no puede ser redimensionado, entonces realloc intentará asignar un nuevo bloque de memoria y copiará el contenido anterior.

```
void *realloc(void *ptr, size_t size);
```


6.- Entrada y salida.

6.- Entrada y salida.

- Las operaciones de E/S no forman parte de C++. Se hacen mediante bibliotecas externas.
- En C++ se dispone de varias clases (<iostream> <fstream>): **`streambuf`**, **`ios`**, **`istream`**, **`ostream`** y **`fstream`**.
- Dejar estas operaciones fuera del lenguaje permite:
  1. Ser independiente de la plataforma: cada compilador posee una versión de la biblioteca.
  2. Encapsulación: pantalla, teclado o ficheros se tratan del mismo modo.
  3. *Buffering*: las operaciones de E/S se agrupan, haciéndolas en memoria. Y las físicas se hacen por bloques ahorrando mucho tiempo.
- Un stream es una abstracción para referirse a cualquier flujo de datos entre una fuente y un destino. Se encargan de convertir cualquier tipo de dato a texto legible por el usuario.
- C++ posee varias clases estándar para el manejo de *streams*:
  `streambuf`: manipulación de *buffers*.
  `ios`: E/S, incluye un objeto de la clase **`streambuf`**.
  `istream`: derivada de **`ios`**, clase especializada de entradas.
  `ostream`: derivada de **`ios`**, clase especializada en salidas.
  `iostream`: derivada de **`istream`** y **`ostream`**, encapsula las funciones de E/S por teclado y pantalla.
  `fstream`: E/S desde ficheros.

6.- Ficheros: Stream.

- Los streams facilitan el acceso a ficheros en disco.
- Cuando creemos un stream para un fichero, podremos trabajar con él igual que hacemos con cin o cout.
- Existen dos tipos de ficheros: ficheros de texto y ficheros binarios.
- Los ficheros de texto son tratados de forma diferente por diferentes S.O., usaremos los binarios.
- Los ficheros binarios permiten almacenar estructuras complejas, en las que se mezclen datos de cadenas con datos numéricos.
- En cuanto al modo de acceso distinguimos entre acceso secuencial y acceso aleatorio.
- Para trabajar con un stream simultáneamente en entrada y salida usaremos la clase fstream.
- Una de las principales ventajas de trabajar con streams es que nos permiten sobrecargar los operadores << y >> para realizar salidas y entradas de nuestros propios tipos de datos.
- Además existen varios flags de estado que podemos usar para comprobar el estado en el que se encuentra un stream.

6.- Ficheros: Stream.

6.1.1.- Abrir un archivo.

La primera operación que generalmente se realiza en un objeto stream es asociarlo a un archivo real y establecer el modo de trabajo con el.

Este procedimiento es conocido como abrir un archivo y usamos su función miembro open: donde una cadena representa el nombre del archivo que se abrirá y unos flag establecen el modo de trabajo:

```
ofstream outFile;
outFile.open ("example.bin", ios::out | ios::app | ios::binary);
ifstream inFile;
inFile.open ("example.txt");
```

Los modos son:

ios::in	Abierto para operaciones de entrada (por defecto).
ios::out	Abierto para operaciones de salida.
ios::binary	Abrir en modo binario (por defecto es texto).
ios::ate	Establecer la posición inicial al final del archivo.
ios::app	Todas las operaciones de salida se realizan al final del archivo, agregando el contenido al contenido actual del archivo.
ios::trunc	Trunca el fichero y empieza a escribir desde el principio.

Para comprobar que el fichero se ha abierto correctamente:

```
if (outFile.is_open()) { /* ok, proceed with output */ }
```


6.- Ficheros: Stream.

6.1.2.- Buffers y Sincronización.

Cuando operamos con flujos de archivos, estos están asociados a un objeto de tipo de búfer interno que representar un bloque de memoria que actúa como un intermediario entre la secuencia y el archivo físico para optimizar las operaciones de entrada/salida a un medio muy lento.

Cuando se vacía el búfer, todos los datos contenidos en él se escriben en el medio físico (si se trata de una secuencia de salida). Este proceso se llama sincronización y se produce:

- Cuando el archivo se cierra: antes de cerrar un archivo, todos los buffers que aún no se han vaciado se sincronizan y todos los datos pendientes se escriben o se leen en el medio físico.
- Cuando el búfer está lleno: los búferes tienen un tamaño determinado y cuando está lleno se sincroniza automáticamente.
- Explícitamente, con manipuladores: cuando se usan ciertos manipuladores (flush y endl) tiene lugar una sincronización explícita.
- Explícitamente, con la función miembro sync (): al llamar a la función sync() se provoca una sincronización inmediata.

Por ello es fundamental cerrar el flujo cuando hayamos terminado con nuestras operaciones de entrada y salida en un archivo: se vacía el búfer, se notifica al S.O. y se liberan los recursos. No cerrar un flujo de salida puede provocar perdidas de datos.

```
outFile.close();
```

6.- Ficheros: Stream.

6.1.3.-Entrada/Salida.

istream	
operador >>	Extraer entrada formateada
gcount	Obtener recuento de caracteres
get	Obtener caracteres
getline	Obtener línea
ignore	Extraer y descartar caracteres
peek	Mira el siguiente caracter
read	Leer bloque de datos
readsome	Leer datos disponibles en el búfer
putback	Devolver el carácter
unget	Unget carácter
tellg	Obtener la posición en la secuencia de entrada
seekg	Establecer posición en secuencia de entrada
sync	Sincronizar el búfer de entrada

ostream	
operador <<	Insertar salida formateada
put	Poner carácter
write	Escribir bloque de datos
tellp	Obtener posición en la secuencia de salida
seekp	Establecer posición en la secuencia de salida
flush	Vaciar el buffer de flujo de salida

6.- Ficheros: Stream.

6.1.4.-Clases base.

ios	
good	Comprueba si el estado de la transmisión es bueno
eof	Comprueba si se establece eofbit
fail	Comprueba si se establece failbit o badbit
bad	Comprueba si badbit está establecido
rdstate	Obtener banderas de estado de error
setstate	Establecer indicador de estado de error
clear	Establecer banderas de estado de error
copyfmt	Copiar información de formato
fill	Obtener/establecer el carácter de relleno
exceptions	Obtener/establecer máscara de excepciones
imbue	Configuración local
tie	Obtener/establecer flujo atado
rdbuf	Obtener/establecer búfer de flujo
narrow	Carácter estrecho
widen	Carácter Widen

ios_base	
flags	Obtener/establecer marcas de formato
setf	Establecer marcas de formato específico
unsetf	Borrar marcas de formato específico
precision	Obtener/Establecer precisión decimal en coma flotante
width	Obtener/establecer ancho de campo
getloc	Obtener la configuración regional actual
xalloc	Obtener nuevo índice para la matriz extensible [estática]
iword	Obtener elemento entero de matriz extensible
pword	Obtener el elemento puntero de la matriz extensible
register_callback	Registrar función de devolución de llamada de evento
sync_with_stdio	Alternar la sincronización con streams cstdio [estática]

6.- Ficheros: Stream.

6.1.5.- Flags.

Podemos modificarlos con las funciones setf() y unsetf()

FLAG	ESTADO	SIGNIFICADO
skipws	SI	Ignorar caracteres blancos en la entrada
left	NO	Salida justificada a la izquierda
right	NO	Salida justificada a la derecha
internal	NO	Usar 'padding' después del signo o de la indicación de base. Esto quiere decir que el signo se pone justificado a la izquierda y el número justificado a la derecha. El espacio del medio se llena con el carácter de relleno.
dec	SI	Base en decimal. Equivalente a la función dec
oct	NO	Base en octal. Equivalente a la función oct
hex	NO	Base en hexadecimal. Equivalente a la función hex
showbase	NO	Usar indicador de base en la salida. Por ejemplo si hex está ON, un número saldrá precedido de 0x y en hexadecimal. En octal saldría precedido únicamente de un 0
showpoint	NO	Poner en la salida el punto decimal y ceros a la derecha del punto (formato de coma) aunque no sea necesario para números reales
uppercase	NO	Usar mayúsculas para los números hexadecimales en la salida y la E del exponente
showpos	NO	Poner un '+' a todos los números positivos en la salida
scientific	NO	Usar notación exponencial en los números reales
fixed	NO	Usar notación fija en los números reales
unitbuf	NO	Volcar todos los streams después de la salida
stdio	NO	Volcar cout y cerr después de la salida

6.- Ficheros: Stream.

6.1.6.- Manipuladores.

MANIPULADOR	SENTIDO	ACCIÓN
dec	E/S	Pone la base a decimal hasta nueva orden
oct	E/S	Lo mismo a octal
hex	E/S	Lo mismo a hexadecimal
ws	E	Extrae los caracteres blancos del stream. Se usa cuando skipws está a 0
endl	S	Inserta una nueva línea. La única diferencia con el carácter '\n' es que endl vuelca el stream
ends	S	Inserta un carácter nulo. Completamente equivalente a ' \ 0 ' y vuelca el string
flush	S	Vuelca el stream
setw(int)	E/S	Pone el ancho del campo. (Equivalente a width()). Tope máximo para la entrada y mínimo para la salida
setfill(char)	S	Pone el carácter de relleno. Por defecto es el carácter espacio ' '. (Equivalente a fill())
setprecision(int)	S	one la precisión a N dígitos después del punto decimal. (Equivalente a precision())
setbase(int)	E/S	Pone la base (0, 8, 10, 16). 8 10 y 16 son equivalentes a oct, dec y hex. 0 implica dec en la salida y en la entrada significa que podemos usar las convenciones del C++ para introducir los números en distintas bases

6.- Ficheros: Stream.

6.1.7.- Escribir.

```
ofstream outFile("example.txt");
if (outFile.is_open()) {
        cout << "SALIDA:\t\t";
        for (int i = 0; i < 30; i++) {
                int val = rand() % 100 + 1;
                outFile << val << endl;
                cout << val << " ";
        }
        cout << endl;
        outFile.close();
} else cout << "Error de apertura";
```

6.1.8.- Leer.

```
ifstream inFile("example.txt");
if (inFile.is_open()) {
        char bufer[1024];
        cout << "ENTRADA:\t";
        while (inFile.getline(bufer, 1024))
                cout << bufer << " ";
        cout << endl;
        inFile.close();
} else cout << "Error de apertura";
```


6.- Ficheros: Flujos.

6.2.1.- Flujos.

Los flujos son una forma flexible y eficiente para leer y escribir datos.

Existe una estructura interna de C, FILE, que representa a todos los flujos y esta definida en stdio.h.

Por lo tanto simplemente se necesita referirse a la estructura para realizar entrada y salida de datos.

La estructura FILE mantiene el manipulador del fichero que se obtiene con la función fopen.

El flujo de E/S usa BUFFER, área temporal de la memoria (intermedia), para minimizar las lecturas/escrituras físicas a disco, por lo que es obligatorio cerrar con fclose todos los ficheros si no se quiere perder la memoria.

Hay 3 flujos predefinidos (en stdio.h):

- stdin: Salida estándar (consola).
- stdout: Entrada estándar (teclado).
- stderr: Salida estándar de errores (consola).

Todas ellas usan texto como método de E/S.

Los flujos stdin y stdout pueden ser usadas con archivos, programas, dispositivos de E/S como el teclado, la consola, etc.

El flujo stderr siempre va a la consola o la pantalla.

6.- Ficheros: Flujos.

6.2.2.- fopen.

FILE *fopen(const char *filename, const char *mode);

Donde mode especifica el tipo de acceso solicitado para el archivo:

- "r" Abre para lectura. Si el archivo no existe o no se encuentra, la llamada de fopen falla.
- "w" Abre un archivo vacío para escritura. Si el archivo especificado existe, se destruye su contenido.
- "a" Abre para escritura al final del archivo (anexo) sin eliminar el marcador de fin de archivo (EOF) antes de que se escriban nuevos datos en el archivo. Crea el archivo si no existe.
- "r+" Abre para lectura y escritura. El archivo debe existir.
- "w+" Abre un archivo vacío para lectura y escritura. Si el archivo existe, se destruye su contenido.
- "a+" Se abre para lectura y anexado. La operación de anexado incluye la eliminación del marcador EOF antes de que los nuevos datos se escriban en el archivo. El marcador EOF no se restablece una vez completada la escritura. Crea el archivo si no existe.

6.- Ficheros: Flujos.

6.2.3.- Entrada/salida de texto.

Carácter a carácter:

```
int   fgetc      (FILE * flujo);
int   fputc      (int c, FILE * flujo);
```

Cadenas:

```
char  *fgets (char *s, int max, FILE * flujo);
int   fputs      (const char *s, FILE * flujo);
```

Cadenas con formato:

```
int fprintf(FILE *flujo, const char *formato, args …);
int fscanf(FILE *flujo, const char *formato, args …);
```

6.2.4.- Formato.

%c	Carácter	%s	Cadena
%d %i	Entero decimal	%o	Entero octal
%x %X	Entero hexadecimal	%u	Entero sin signo
%f	Real	%e %E	Real en notación científica
%g %G	Real en %f o %e, la mas compacta	%%	Carácter porciento

Entre el % y el carácter de formato se puede aparecer:

- Signo menos para justificar a la izquierda

número entero para el ancho del campo o *m.d* en donde m es el ancho del campo y d es la precisión de dígitos después del punto decimal o el número de caracteres de una cadena.

6.- Ficheros: Flujos.

6.2.5.- Entrada/salida binario.

size_t fread(void *buffer, size_t size, size_t count, FILE *stream);
size_t fwrite(const void *buffer, size_t size, size_t count, FILE *stream);
buffer: Ubicación del destino de los datos o Puntero a los datos que se van a escribir.
size: Tamaño del elemento en bytes.
count: Número máximo de elementos que se va a leer o escribir.
stream: Puntero a la estructura FILE.
Devuelve el número de elementos completos leídos o realmente

6.2.6.- Estado del flujo.

Volcar buffer y cerrar flujo
int fclose (FILE *stream);
Prueba de fin de archivo en flujo
int feof (FILE *stream);
Volcar buffer en el fichero
int fflush (FILE *stream);
Mover posición de archivo a la ubicación dada
int fseek (FILE *stream, long offset, int whence);
Mover la posición de archivo al principio de la flujo
void rewind (FILE *stream);

6.- Ficheros: Flujos.

6.2.7.- Escribir.

```
FILE *stream;
int list[30];
int  i, numwritten;
stream=fopen("fread.out", "w");
if(stream != NULL ) {
      for ( i = 0; i < 30; i++ ) list[i] = 100+i;
      numwritten = fwrite( list, sizeof(int), 30, stream );
      printf("Escritos %d elementos.\n", numwritten);
      fclose(stream);
}
```

6.2.8.- Leer.

```
FILE *stream;
int list[10];
int  i, numread;
stream=fopen("fread.out", "r");
if(stream != NULL ) {
      while(!feof(stream)) {
            numread = fread(list, sizeof(int), 10, stream );
            printf( "Leídos = %d\n", numread );
            for ( i = 0; i < numread; i++ )  printf( "%d\n", list[i] );
      }
      fclose( stream );
}
```

7.- El lenguaje C++ y la POO.

avante
¿hasta dónde quieres llegar?

69

avante
¿hasta dónde quieres llegar?

7.- El lenguaje C++ y la POO.

Surge para permitir la reutilización del código, mejorando la calidad y abaratando los coste.

Existen 4 principios básicos que cualquier sistema O-O debe incorporar:
abstracción, *encapsulamiento*, *herencia* y *polimorfismo*.

Abstracción: que el objeto realice su trabajo, informe, cambie su estado y se comunique con otros objetos del sistema sin revelar cómo se implementan estas características.

Encapsulamiento: reunir a todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción. Aumenta la cohesión de los componentes.

Herencia: la relación de jerarquía que existe entre las clases. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen. Facilita el encapsulamiento.

Polimorfismo: comportamientos diferentes, asociados a objetos distintos, pueden compartir el mismo nombre. Al llamarlos por ese nombre se utilizará el comportamiento correspondiente al objeto que se esté usando.

Además, la sobrecarga de operadores y la creación personalizada de tipos que se comporten como tipos fundamentales, son particularidades de C++.

70

7.- El lenguaje C++ y la POO.

7.1. Terminología

POO: paradigma de programación cuya misión es agrupar los datos y los procedimientos para manejarlos en una única entidad (el objeto). Cada programa es un objeto, que a su vez está formado de objetos que se relacionan entre ellos.

Objeto: una unidad que engloba en sí misma los datos y la funcionalidad necesaria para el tratamiento de esos datos.

Atributo: Variables internas al objeto que almacenan los datos dentro del mismo.

Método: Funciones internas al objeto que le dotan de comportamiento.

Mensaje: Es la petición a un objeto para que realice una operación, es el modo en el que se comunican e interrelacionan los objetos entre si.

Clase: Es la definición de las características para un número arbitrario de objetos, es el patrón para construir objetos.

Instancia: Cada representación concreta del objeto creada a partir de una clase.

Interfaz: la parte del objeto que es visible para el resto de los objetos. Es decir, el conjunto de métodos de que dispone un objeto para comunicarse con él.

Jerarquía: definición del orden de subordinación de un sistema de clases.

7.- El lenguaje C++ y la POO.

7.2.- Herencia.

Concepto: Es la habilidad de una clase para definir su comportamiento y sus estructuras de datos en función de otra clase.

Subclase: Se dice que una clase es una subclase o clase derivada cuando hereda parcial o totalmente de otra clase.

Superclase: Se dice que una clase es una superclase o clase base cuando otra clase hereda de ella.

Superclases abstractas: Una superclase abstracta es aquella clase que se crea para factorizar el conocimiento, sin entidad propia, cuya finalidad es que hereden su subclases.

Herencia múltiple: Es la habilidad de una clase para heredar de más de una superclase. Subclase y superclase.

Polimorfismo: es la posibilidad de utilizar el mismo nombre para diferentes métodos si residen en clases diferentes, tiene múltiples significados.

Sobrecarga: es la posibilidad de crear varias versiones de un método de una clase para disponer de múltiples significados dependiendo del tipo de sus argumentos

Sobrecarga de operadores: ampliar el significado de un operador con nuevas funcionalidades.

Sobreescritura: es la posibilidad que tienen las clases derivadas de sustituir un método heredado por su propia versión.

Dinamic Binding: es el enlazamiento dinámico, donde el enlazamiento se realiza durante la ejecución del programa.

Late Binding: es el enlazamiento estático, donde el enlazamiento se realiza en el momento de la compilación.

Persistencia: Cantidad de tiempo durante el cual un objeto tiene asignada memoria y es accesible.

Framework: Es un esqueleto de programa que hay que rellenar para construir una aplicación. Son diseños reutilizables.

8.- Clases y objetos.

8.- Clases y objetos.

8.1.- Los objetos.

Los objetos son entidades que combinan ***estado***, ***comportamiento*** e ***identidad***.

El **estado** está compuesto de datos, será uno o varios atributos a los que se habrán asignado unos valores concretos.

El **comportamiento** está definido por los procedimientos o métodos con que puede operar dicho objeto, es decir, qué operaciones se pueden realizar con él.

La **identidad** es una propiedad de un objeto que lo diferencia del resto, dicho de otra forma, es su identificador.

La POO expresa un programa como un conjunto de objetos, que colaboran entre ellos para realizar tareas. Esto permite hacer los programas más fáciles de escribir, mantener y reutilizar.

8.- Clases e Instancias.

8.2.- Las clases.

Una clase es un patrón que define las características comunes de un conjunto de elementos. Son los moldes con los que creamos las distintas instancias.

```
class <identificador-de-clase> [<:lista-clases-base>] {
   <lista-de-miembros>
} [<lista-identificadores-de-objetos>];
```

<lista-clases-base> se usa para derivar clases, de momento sólo haremos clases base.

<lista-de-miembros> será una lista de datos y funciones (prototipos, generalmente).

Para definir dichas funciones, fuera de la definición de la clase, precedemos al nombre de la función con el nombre de la clase utilizando :: como separador:

```
<tipo-retorno-función> <id-clase>::<id-función>(<par-función>) {
   <sentencias-cuerpo-función>
}
```

8.- Clases y objetos.

8.2.- Las clases.

Dentro de **<lista-de-miembros>**, cada miembro puede tener diferentes niveles de acceso.

```
class <identificador-de-clase> {
   public:
        <lista-de-miembros>
   private:
        <lista-de-miembros>
   protected:
        <lista-de-miembros>
}
```

Acceso privado (**private**): sólo son accesibles por los propios miembros de la clase, pero no desde funciones externas o desde clases derivadas.

Acceso público (**public**): accesible desde cualquier parte donde sea accesible el propio objeto.

Acceso protegido (**protected**): con respecto a las funciones externas se comporta como si fuera privada, pero con respecto a las clases derivadas se comporta como pública.

Por defecto, los miembros de una clase son privados.

8.- Clases y objetos.

8.2.- Las clases.

struct: Los elementos son por defecto públicos.

struct <nombre_clase> {
public:
...
lista de elementos públicas
...
private:
...
lista de elementos privados
...
};

union: Los elementos son por defecto públicos.

union <nombre_clase> {
public:
...
lista de elementos públicas
...
private:
...
lista de elementos privados
...
};

8.- Clases y objetos.

8.3.- Constructores.

- Son funciones miembro especiales que sirven para inicializar un objeto al mismo tiempo que éste se declara.
- Tienen el mismo nombre que la clase a la que pertenecen.
- No tienen valor de retorno.
- No pueden ser heredados.
- Deben ser SIEMPRE públicos.

```
class <id-clase> {
   public:
        <id-clase>(<lista-param>) [:<lista-constructores>] {
                <código-constructor>
        }
        ...
}
```

- Si no creamos uno, el compilador crea uno por defecto, sin parámetros que no hará nada.
- El constructor será llamado SIEMPRE que se declare un objeto de esa clase.
- Si lleva parámetros es OBLIGATORIO dárselos en la llamada.
- Si no lleva parámetros, en la declaración del objeto no se deben escribir los ().

8.- Clases y objetos.

8.3.- Constructores.

- **Sobrecarga de constructores**: pueden definirse varios constructores para cada clase. La única limitación es que no pueden declararse varios constructores con el mismo número y el mismo tipo de parámetros.

- **Constructores con argumentos por defecto**: al igual que con las funciones, podemos establecer parámetros por defecto en constructores ahorrándonos la declaración de alguno de ellos.

- **Constructor copia**: crea un objeto a partir de otro objeto existente. Sólo tienen un argumento, que es una referencia a un objeto de su misma clase:

```
<tipo-clase>::<tipo-clase>(const <tipo-clase> &<id-objeto>);
```

- En este caso el compilador también crea uno por defecto, si no lo hacemos nosotros.

8.- Clases y objetos.

8.4.- Destructores.

- Funciones miembro especiales que sirven para eliminar un objeto de una determinada clase.
- Su finalidad es liberar: la memoria dinámica, recursos usados, ficheros, dispositivos, etc.
- Tienen el mismo nombre que la clase a la que pertenecen, anteponiendo el símbolo **~**.
- No devuelven ningún valor.
- No tienen parámetros.
- No pueden ser heredados.
- Deben ser públicos.
- NO pueden ser sobrecargados.
- El destructor es llamado automáticamente cuando se abandona el ámbito donde fue definido.
- Si el objeto fue creado dinámicamente con **`new`**, hay que eliminarlo con **`delete`**.

8.- Clases y objetos.

8.5.- El puntero `this`.

- Para cada objeto declarado de una clase, todos mantienen una copia de sus datos. PERO todos comparten la misma copia de las funciones de esa clase.
- Esto redunda en un ahorro considerable de memoria y de tamaño de los programas, pero plantea un problema serio: **¿cómo sabe la función el dato de qué objeto tiene que modificar?** La respuesta es: usando el puntero **`this`**.
- Es un puntero especial que tiene asociado cada objeto y que apunta a sí mismo.
- Este puntero permite resolver conflictos de ámbito y pasar/devolver un puntero a si mismo.

8.6.- Sistema de protección.

- Los miembros privados de una clase no son accesibles para funciones y clases exteriores. Esta es la finalidad del ***encapsulamiento***.
- En ocasiones querremos tener acceso a algunos miembros de un objeto de una clase desde otros objetos de clases diferentes, sin perder ese encapsulamiento para el resto del programa, es decir, manteniendo esos miembros como privados.
- C++ proporciona un mecanismo para bordear el sistema de protección.

8.- Clases y objetos.

8.6.- Sistema de protección.

- Declaraciones **`friend`**: el modificador **`friend`** puede aplicarse a clases o funciones para inhibir el sistema de protección.

- *La amistad no puede propagarse*: si A es amigo de B, y B es amigo de C, **no implica** que A sea amigo de C.

- *La amistad no puede heredarse*: si A es amigo de B, y C es una clase derivada de B, **A no es amigo de C**.

- *La amistad no es simétrica*: si A es amigo de B, **B no tiene por qué ser amigo de A**.

- Lo más sencillo es una relación de amistad con una función externa.

- Estas amistades son especialmente útiles cuando sobrecargamos algunos operadores.

- Lo más común es aplicar relaciones de amistad a clases completas.

8.- Clases y objetos.

8.7.- Modificadores para miembros.

- A la hora de definir algunos miembros de las clases, tenemos varias alternativas.
- Los modificadores afectan al modo en que se genera el código de ciertas funciones y datos, o al modo en que se tratan los valores de retorno.
- **Funciones miembro constantes (`const`):** aquellas clases que no deben modificar el valor de ningún dato de la clase. Podemos saltarnos este "deber" pero el compilador avisará con un error.
- **Valores de retorno constantes**: usados especialmente al devolver punteros miembro de la clase.
- **Miembros estáticos de una clase** (**`static`**): tienen algunas propiedades especiales.
- Cuando un dato miembro es **`static`**, sólo existirá una copia que compartirán todos los objetos de la misma clase.
- Es necesario declarar e inicializar los miembros **`static`** de la clase porque:

  1.- los miembros **`static`** deben existir aunque no exista ningún objeto de la clase,
  2.- si no lo hacemos, al declarar objetos de esa clase, los valores de los miembros estáticos serían indefinidos.

- En el caso de las funciones miembro, no pueden acceder a los miembros de los objetos, sólo a los datos miembro de la clase que sean **`static`** = no tienen acceso al puntero **`this`**, y además deben ser usadas con su nombre completo (usando el operador de ámbito.)

8.- Clases y objetos.

8.8.- Sobrecarga de operadores.

- Cuando se sobrecargan operadores en el interior de las clases, se asume que el primer operando es el propio objeto de la clase donde se define el operador.

```
<tipo> operator<operador-binario>(<tipo> <identificador>);
```

- Sobrecargar el operador de asignación: ¿para qué?

- En ocasiones hay ciertas estructuras de datos en las que la asignación directa es perfectamente válida. Pero en otras ocasiones la asignación directa sólo significa asignación de punteros.

- Esto último provocaría que dos punteros apuntaran a la misma posición de memoria, con lo cual, si el contenido de esa dirección es modificada por alguien, sería modificado para todos los que apuntan a ella.

- Suelen darse errores muy frecuentemente cuando se pasa por alto este hecho al usar cadenas de caracteres y otros arrays.

9.- Herencia.

9.- Herencia.

9.1.- Jerarquía, clases base y clases derivadas.

- Clase derivada: nueva clase obtenida mediante herencia desde otra clase (clase base).
- Cuando una clase derivada lo es de más de una clase base, hablamos de herencia múltiple.
- Esto nos permite crear una jerarquía de clases todo lo compleja que queramos.
- ¿Qué ventajas ofrece la herencia o derivación de clases?
- Nos permite encapsular diferentes partes de cualquier objeto y vincularlo con objetos más elaborados del mismo tipo básico, que heredarán todas sus características.
- Las jerarquías de clases ofrecen toda su potencia en la resolución de problemas complejos. Normalmente no tendrás que recurrir a ellas para resolver problemas sencillos.
- Una jerarquía de clases cohesionada proporciona todo el significado de la POO, es una muy potente y efectiva herramienta, PERO es necesario un análisis y diseño previos exahustivos, ya que un error de diseño con el proyecto avanzado es extremadamente costoso de reparar y suele ser más barato desechar el proyecto y empezar de cero.

9.- Herencia.

9.2.- Derivar clases.

- La sintaxis general de declarar clases derivadas es:

```
class <clase-derivada>:
   [public|private] <base1> [,[public|private] <base2>] {};
```

- Si no se especifica el tipo de acceso, se asume que es **`private`**.
  - **`public`**: los miembros heredados de la clase base conservan su tipo de acceso.
  - **`private`**: los miembros heredados de la clase base pasan a ser privados en la clase derivada.
- Cuando creamos estructuras jerárquicas de clases suele interesar que las clases derivadas tengan acceso a los datos miembros de las clases base: definirlos como **`protected`**.
- De este modo, protegemos a los datos de accesos externos a nuestras clases, pero a la vez que sean accesibles desde nuestras clases derivadas.

9.- Herencia.

9.3.- Constructores de clases derivadas.

- Cuando se crea un objeto de una clase derivada, primero se invoca al constructor de la clase base y a continuación al constructor de la clase derivada. Si la clase base es a su vez una clase derivada, el proceso se repite recursivamente.

- Este concepto puede parecer sencillo, pero cuando usamos constructores sobrecargados o constructores con parámetros, la operación no es tan simple.

- Para inicializar una clase base desde una clase derivada, usamos el constructor de la derivada:

```
<clase-derivada>(<lista-param>) : <clase-base> (<lista-param>) {}
```

9.4.- Destructores de clases derivadas.

- Cuando se destruye un objeto de una clase derivada, primero se invoca al destructor de la clase derivada, si existen objetos miembro a continuación se invoca a sus destructores y finalmente al destructor de la clase o clases base. Si la clase base es a su vez una clase derivada, el proceso se repite recursivamente.

9.- Herencia.

9.5.- Redefinición de funciones en clases derivadas.

- En una clase derivada podemos definir una función que ya existía en la clase base. Esta redefinición oculta la definición previa en la clase base (aunque esté sobrecargada).

- En caso de querer acceder a la función de la clase base debemos escribir su nombre completo (::).

9.6.- Polimorfismo.

- El polimorfismo en C++, llega a su máxima expresión cuando la usamos junto con punteros o con referencias.

- C++ nos permite acceder a objetos de una clase derivada usando un puntero a la clase base.

- Sólo podremos acceder a datos y funciones que existan en la clase base, los datos y funciones propias de los objetos de clases derivadas serán inaccesibles.

9.- Herencia.

9.7.- Funciones virtuales.

- Cuando en una clase declaramos una función como **`virtual`**, y la superponemos en alguna clase derivada, al invocarla usando un puntero de la clase base, se ejecutará la versión de la clase derivada.

- Una vez que una función es declarada como **`virtual`**, lo seguirá siendo en las clases derivadas.

- Si la función **`virtual`** no se define exactamente con el mismo tipo de valor de retorno y el mismo número y tipo de parámetros que en la clase base, no se considerará como la misma función, sino como una función superpuesta.

- Este mecanismo sólo funciona con punteros y referencias, usarlo con objetos no tiene sentido.

- Si en una clase existen funciones virtuales, **el destructor DEBE ser `virtual`**.

- Los constructores no pueden ser virtuales, por ello el constructor copia no hará siempre lo que se espera que haga. NO debemos usar el constructor copia cuando usemos punteros a clases base.

- Si un constructor llama a una función **`virtual`**, ésta será SIEMPRE la de la clase base: porque el objeto de la clase derivada aún no ha sido creado.

9.- Herencia.

9.8.- Herencia múltiple.

- Se trata de crear clases derivadas que hereden de varias clases base. Las clases así derivadas, heredan los datos y funciones de TODAS las clases base.

```
<clase-derivada>(<lista-param>):
   <clase-base1>(<lista-param>)
    [, <clase-base2>(<lista-param>)] {}
```

- Cuando varias clases base tienen una función con el mismo nombre, la clase derivada creada con herencia múltiple tiene que redefinir esa función para evitar la ambigüedad, o bien acceder a la función que necesite con su nombre completo, usando el operador de ámbito (`::`).

- El constructor de la clase derivada deberá llamar a los constructores de las clases base cuando sea necesario.

9.9.- Funciones virtuales puras.

- Es aquella que no necesita ser definida: puede resultar útil en algunas ocasiones.

- El modo de declarar este tipo de funciones es asignándole el valor cero.

```
virtual <tipo> <nombre-funcion>(<lista-param>) = 0;
```

9.- Herencia.

9.10.- Clases abstractas.

- Son aquellas clases que poseen al menos una función virtual pura.

- No se pueden crear objetos de una clase abstracta: sólo se usan como clases base para la creación de clases derivadas.

- Las funciones virtuales puras serán aquellas que siempre se definirán en las clases derivadas, de modo que no será necesario definirlas en la clase base.

- Siempre hay que definir todas las funciones virtuales de una clase abstracta en sus clases derivadas, no hacerlo así implica que la nueva clase derivada será también asbastacta.

10.- Tratamiento de excepciones.

avante ¿hasta dónde quieres llegar?

10.- Tratamiento de excepciones.

- Las excepciones son errores que se producen durante la ejecución de un programa.

- Si no capturamos y tratamos dichas excepciones el programa terminará de forma inesperada.

- Que el programa termine de forma inesperada implica: pérdida de datos en buffers que escriben en ficheros, que no se cierren tales ficheros, objetos sin destruir, fugas de memoria, etc.

- Tenemos, en la medida de lo posible, que prever dónde de pueden producir excepciones y estar preparados para cuando se produzcan.

- El tratamiento de excepciones consiste en transferir la ejecución del programa desde el punto donde se produce la excepción a un manejador que coincida con el motivo de la excepción.

```
try {
    [<código-erróneo>] | [throw <id-excepción>];
}
catch (<tipo-excepción>) {
    <tratamiento-para-tipo-excepción>
}
catch (...) {
    <tratamiento-para-tipo-excepción-no-tratada>
}
```


10.- Tratamiento de excepciones.

Si dentro de una función detectamos un error lanzaremos una excepción poniendo la palabra throw y un parámetro de un tipo determinado, es como si ejecutáramos un return de error de un objeto (una cadena, un entero o una clase definida por nosotros).

Por ejemplo:

```
f() {
 ...
 int *i;
 if ((i= new int) == NULL)
   throw "Error al reservar la memoria"; // no hacen falta paréntesis,
                                         // es como en return
 ...
}
```

Si la función f() fue invocada desde g() y esta a su vez desde h(), el error se irá propagando entre ellas hasta que se recoja.

11.- Plantillas

avante
¿hasta dónde quieres llegar?

97

avante
¿hasta dónde quieres llegar?

11.- Plantillas

Una función genérica es realmente como una plantilla de una función, lo que representa es lo que tenemos que hacer con unos datos sin especificar el tipo de algunos de ellos.

Por ejemplo una función máximo se puede implementar igual para enteros, para reales o para complejos, siempre y cuando este definido el operador de relación <.

La idea de las funciones genéricas es definir la operación de forma general, sin indicar los tipos de las variables que intervienen en la operación.

Una vez dada una definición general, para usar la función con diferentes tipos de datos la llamaremos indicando el tipo (o los tipos de datos) que intervienen en ella.

En realidad es como si le pasáramos a la función los tipos junto con los datos.

Al igual que sucede con las funciones, las clases contenedor son estructuras que almacenan información de un tipo determinado, lo que implica que cada clase contenedor debe ser reescrita para contener objetos de un tipo concreto.

Si definimos la clase de forma general, sin considerar el tipo que tiene lo que vamos a almacenar y luego le pasamos a la clase el tipo o los tipos que le faltan para definir la estructura, ahorraremos tiempo y código al escribir nuestros programas.

98

11.- Plantillas.

11.1.- Funciones genéricas.

Para definir una función genérica se utiliza template con una lista de nombres de genéricos (precedidos de la palabra class) y se define la función utilizando los nombres de los tipos genéricos.

```
template <class T>  T max (T a, T b) { // función genérica del máximo
    return (a > b) ? a : b;
}
```

Las funciones genéricas se pueden sobrecargar siempre que la firma difiera.

Los tipos se resuelven en la invocación.

```
int i = 9, j = 12;
cout << max (i, j); // Genera la version int max(int a, int b)
cout << max ("HOLA", "ADIOS"); // Genera la version const char* max(const
    char* a, const char* b)
```

Todos lo modificadores de una función (inline, static, etc.) van después de template < ... >.

11.2.- Clases genéricas.

Se definen de forma similar a las funciones genéricas:

```
template <class T> // podríamos poner más de un tipo
class vector {
 T* v;
public:
T& operator[] (int); // el operador devuelve objetos de tipo T
 ...
}
vector<int> v(100);
```


Ejercicios.

12.- Ejercicios.

- Numero mágico (rand(), srand(time(NULL)))
  - Básico
  - Registro de jugadores y récords
  - Persistencia
- Aritmética de punteros:
  - Mínimo, máximo, media, …
- Cadenas
  - Invertir, contar, comparar
- Calculadora
  - Básica
  - Con fichero de entrada
  - Con fichero de salida
- Estructuras:
  - Empleado: id, nombre, apellidos, edad
  - Sobreescribir operador de igualdad
  - Añadir gestión de salario.
- Juego del Ajedrez
  - Con jugadores
  - Con registro de jugadas

101

avante
¿hasta dónde quieres llegar?

Sevilla
c/ Torricelli 26 - PCT Cartuja
CP 41092 - Sevilla

Contacta con nosotros
Tel: 902 117 902
Tel: 954 186 540
formacion@avante.es

Síguenos en las redes
@AvanteFormación
@AvanteTIC
Avante
Avante Formación

102