

# Segunda Sesión

## Metodologías y Técnicas de Programación II Programación Orientada a Objeto (POO)

**C++**

### Profesor:

José Luis Marina  
[jmarina@nebrija.es](mailto:jmarina@nebrija.es)

### Laboratorios:

José Luis Marina  
[jmarina@nebrija.es](mailto:jmarina@nebrija.es)  
Borland Builder 6.0

### Puntuación:

Prácticas Laboratorio:	<b>20%</b>
Diarias	20%
Trabajo Laboratorio	80%

Exámen Parcial	<b>15%</b>
----------------	------------

Exámen Final	<b>65%</b>
--------------	------------

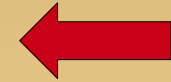
<b>Exámen Final Extraordinario</b>	<b>70%</b>
------------------------------------	------------

# 2.0 Estado del Programa

## Introducción a la POO

Historia de la Programación  
Conceptos de POO

C++  
Mi primera Clase



## Repaso de Conceptos

Estándares de Programación  
Punteros y Memoria

E/S  
Control y Operadores

## Clases y Objetos en C++

Uso y aplicación  
Constructores

Funciones Amigas  
Constantes e "inline"

## Sobrecarga

De Operadores

De Funciones

## Herencia.

Tipos de Visibilidad

Herencia Múltiple

## Polimorfismo

Funciones Virtuales

Polimorfismo y Sobrecarga.

## Plantillas

Contenedores

Iteradores

# 2.1 Repaso de clases anteriores.

## Buscamos la Reutilización Sistemática - (POO)

Queremos escribir sólo el código necesario, equivocarnos menos, poner y quitar funcionalidad de forma fácil.

### Todo es un Objeto:

- Un objeto es como una variable mejorada.
- Un objeto es una instancia de una clase determinada.

### Los objetos se comunican mediante mensajes:

- Los programas serán grupos de objetos enviándose mensajes entre sí.

### Características:

- Abstracción
- Encapsulación
- Herencia
- Polimorfismo

### C++:

- Language de Propósito General
- Lenguaje Orientado a Objeto.
- Eficiente y elegante
- 3.000.000 programadores (se usa)

Alumno
<pre>- nombre_ : string - apellido1_ : string - apellido2_ : string - dni_ : string - edad_ : int</pre>
<pre>+ nombre(string nom) + apellido1(string a1) + apellido2(string a2) + dni(string dni) + edad(int anios)  + nombre() : string + apellido1() : string + apellido2() : string + dni() : string + edad(): int</pre>

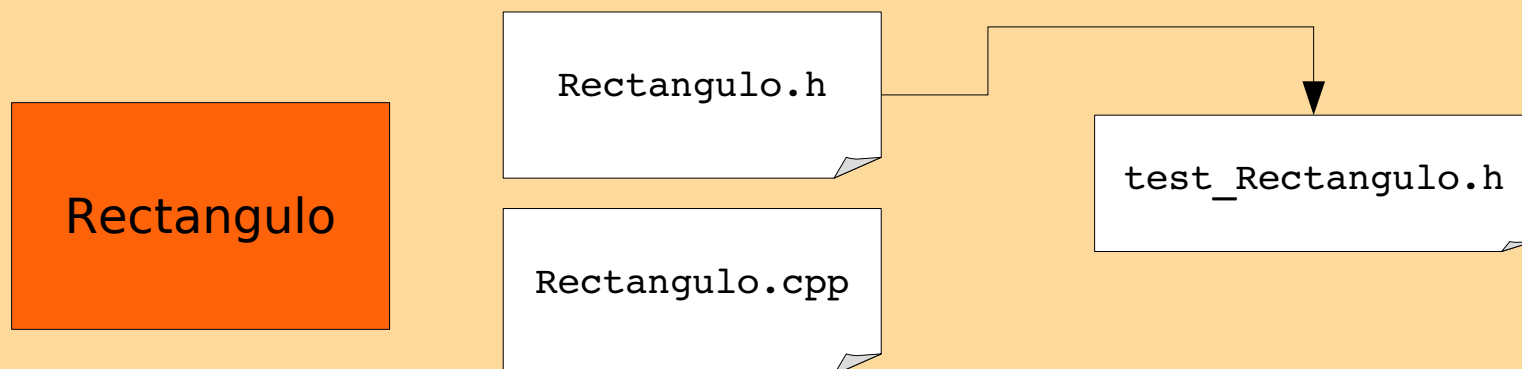
## 2.2 Organización de Ficheros

`nombre_clase.h` contiene la definición de la clase.

`nombre_clase.cpp` contiene la definición de las funciones y operadores de la clase.

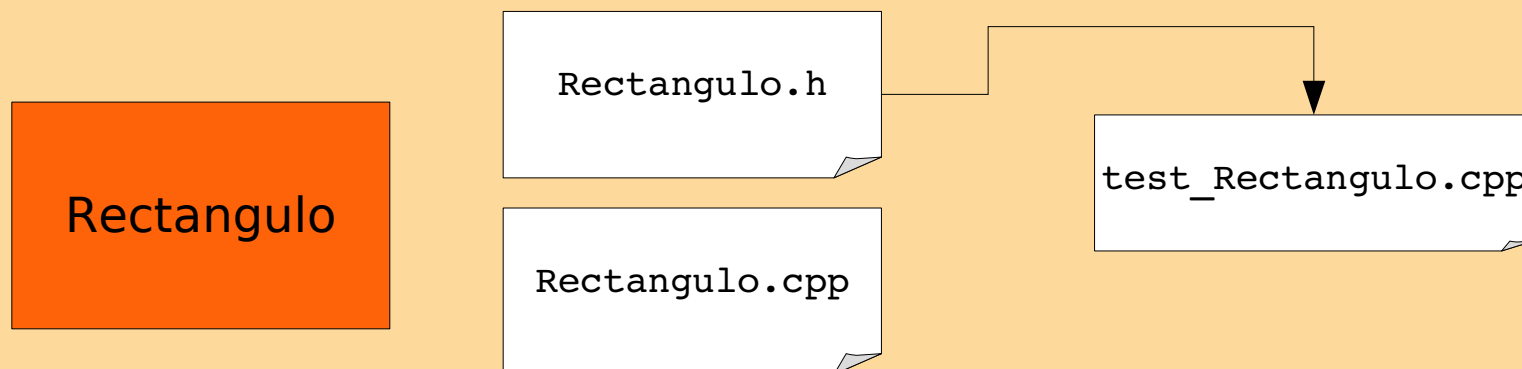
`programa.cpp` contiene un programa principal con algunas acciones sobre la clase.

Organización bastante habitual en C++



## 2.2 Organización de Ficheros

- Los usuarios de la clase tienen acceso al *header* o fichero de cabecera o fichero de declaración(nombre\_clase.h)
- La definición de las funciones y operadores (implementación de la clase) se hace en otro fichero fuente al cual no es necesario acceder. Los detalles de este código pueden quedar ocultos al usuario (cliente) de la clase .
- Me pueden dar acceso al “.h” y en lugar del “.cpp” al código objeto generado (”.o”, “.obj”, “.lib”. “.dll”).



## 2.2 Organización de Ficheros

- El fichero nombre\_clase.h se encierra en el siguiente código de preprocesador:

```
#ifndef NOMBRE_CLASE_H
#define NOMBRE_CLASE_H
    ...
#endif
```

- Las directivas de preprocesador anteriores evitan que el código entre `#ifndef` y `#endif` sea incluido si se ha definido el nombre `NOMBRE_CLASE_H`, es decir, si el encabezado ha sido incluido previamente en algún otro archivo.
- La inclusión (inadvertida) del mismo archivo sucede por lo general en programas grandes con muchos archivos de cabecera

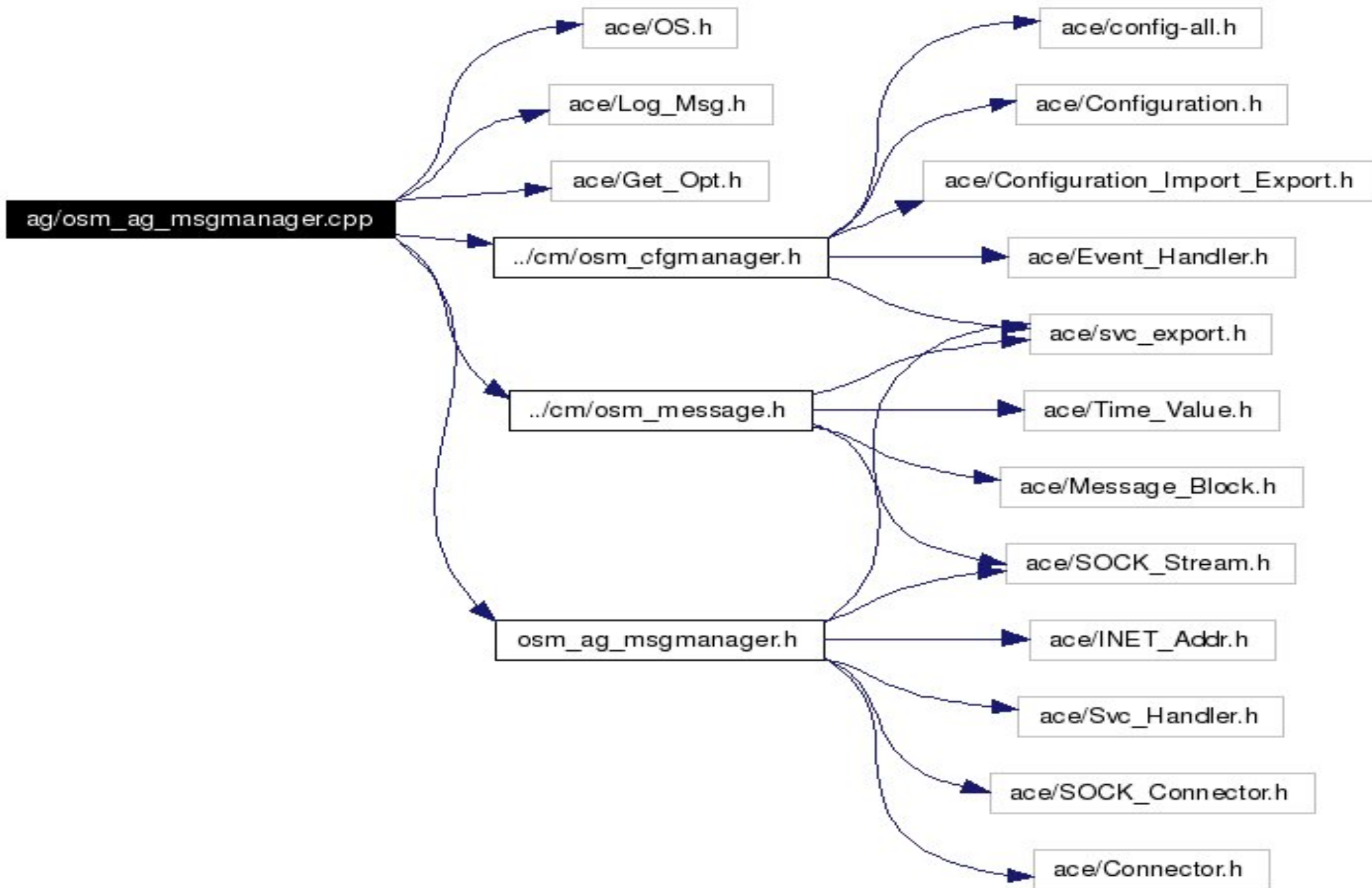
## 2.2 Organización de Ficheros

- Podemos tener 50 clases en nuestro programa que – recordemos – nos ayudan a ver mejor el problema que tratamos de solucionar.
- Estas clases estarán en 50 ficheros de cabecera y 50 de implementación.
- Unas clases harán uso de otras (coche y motor: Coche incluye Motor.h)

Queremos **Claridad** y poder **Reutilizar**



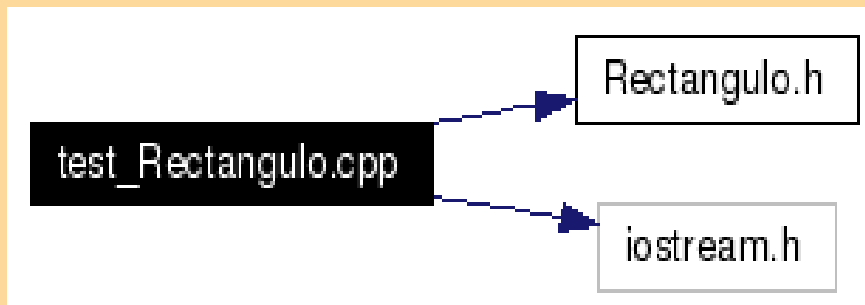
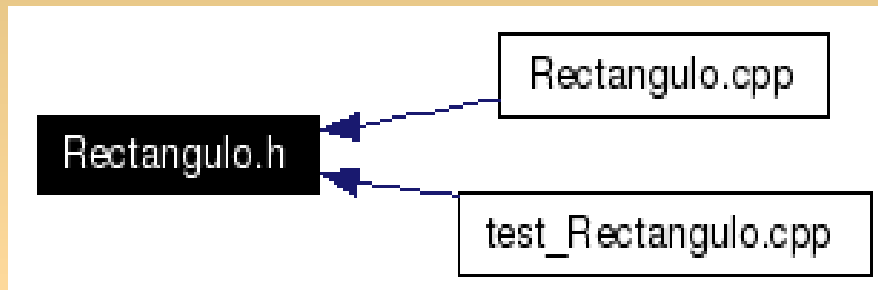
## 2.2 Organización de Ficheros



## 2.2 Organización de Ficheros

Ficheros de la Clase Rectangulo:

- Ver ficheros con el código fuente



Rectangulo
- altura_: int - anchura_: int
+ inicializar() + altura(): int + anchura(): int + area(): int
+ altura(int) + anchura(int)

## 2.3 Estándares

### Nombres de Clases, Objetos, atributos y métodos – Indexado y corchetes

```
class Rectangulo
{
private:
    int  altura_;
    int  anchura_;
public:
    void inicializar(void);
    int  altura(void);
    int  area(void);
    void altura(int alto);
};
```

```
int main(void)
{
    Rectangulo mi_rectangulo;
    Rectangulo rectangulo_ejemplo;

    Rectangulo* p_rec;

    .....
    return 0;
}
```

```
// -----
void Rectangulo::altura(int alto)
{
    if (alto > 0)
    {
        altura_ = alto;
    }
    else
    {
        altura_ = -1;
    }
}
```

## 2.3 Estándares

### Cabeceras de los Ficheros:

```
/**
 * @file test_Rectangulo.cpp
 * @brief Pruebas sobre la clase Rectangulo. Ejemplo Metodos II
 *
 * @author Jose Luis Marina <jlmarina@nebrija.es>
 * @date 19-FEB-2007
 */
```

### En las declaraciones de las Clases:

```
/**
 * @class Rectangulo
 *
 * @brief Clase Rectangulo para ejemplo de uso de estándares en Metodos II
 *
 * Podemos utilizar esta clase mas adelante para mostrar herencia y
 * polimorfismo.
 */
```

## 2.3 Estándares

Comentarios de documentación de atributos y metodos:

```
/// Altura del rectangulo en metros
int  altura_; // Comentario sobre la altura.
/**
 * Devuelve el valor del area.
 * @retval Area en metros cuadrados y -1 en caso de error.
 */
int area(void);

/**
 * Asigna valor a la altura del rectangulo.
 * Comprueba que el valor sea positivo.
 * @param alto Anchura en metros.
 */
void altura(int alto);
```

En los ficheros de cabecera que incluirá en resto (#include):

```
#ifndef RECTANGULO_H
#define RECTANGULO_H
... código .....
#endif /* RECTANGULO_H */
```

## 2.3 Estándares

### Ventajas de los Estándares

#### Ayudan a Reutilizar el código

El código realizado por dos programadores “se parece”. Es más fácil que entendamos el código del otro y lo haremos de forma más rápida.

#### Facilitan la Documentación

Documentar un sistema para que otros lo entiendan (o yo mismo dentro de un tiempo) es parte de la tarea de programación.

Un código incomprensible es poco:

- Mantenable
- Ampliable
- Reutilizable

#### Cambios en los Estándares

Sí, pero de manera consensuada y avisando al resto.

(\*) Ver ejemplo Doxygen y Rectángulo

## 2.4 Programación Estructurada

### Funciones

Encapsulan la complejidad en programación procedimental.  
Separamos la declaración de la implementación.

#### Declaración:

```
int translate(float x, float y, float z); // Correcto. La preferimos.  
int translate(float , float , float ); // Correcto.  
int translate(float x,y,z); // Incorrecto  
void funcion(void) ; // Sin argumentos. Equivalente a void funcion()
```

#### Implementación:

```
int translate(float x, float y, float z) // Correcto  
{  
    x = y = z;  
}  
int translate(float , float , float ) // Incorrecto.  
{  
}
```

## 2.4 Programación Estructurada

### Funciones: Valores de Retorno

Se debe especificar el tipo del valor devuelto. Si no devolvemos nada utilizamos **void**.

Declaración:

```
int    int f1(void); // Devuelve un entero, no tiene argumentos
int f2(); // igual que f1() en C++ pero no en C Standard
float f3(float, int, char, double); // Devuelve un float
void f4(void); // No toma argumentos, no devuelve nada
```

Para devolver valores usamos **return**

```
char cfunc(int i)
{
    if(i == 0)
        return 'a';
    if(i == 1)
        return 'g';
    if(i == 5)
        return 'z';
    return 'c';
}

int main() {
    cout << "Introduzca un entero: ";
    int val;
    cin >> val;
    cout << cfunc(val) << endl;
}
```

## 2.4 Programación Estructurada

### Funciones: Uso de funciones en librerías

Tenemos que incluir su declaración con ***include***

```
#include <iostream.h>    // Librería de entrada salida.
#include <mathlib.h>     // Librería con funciones matemáticas.
#include "lib_trigonometrica.h" // En path de proyecto o local.
.....
cout << "Estoy usando iostream" << endl;
x = sqrt(9);            // mathlib
y = funcion_coseno(180); // librería de funciones trigonometricas.
.....
```

Además con nuestro código se enlazará el código objeto adecuado:

```
.lib .a .o    -> Enlazado estático
.dll .so     -> Dinámico
```

## 2.4 Programación Estructurada

### Control de Flujo: If

Tiene dos formas:

```
if (expresion)
    sentencia
```

```
if (expresion)
    sentencia
else
    sentencia
```

Expresión se evalúa como **true** o **false**. False es equivalente a 0 y true a No\_cero.

```
cout << "type a number and 'Enter'" << endl;
cin >> i;
if(i < 10)
    if(i > 5) // "if" is just another statement
        cout << "5 < i < 10" << endl;
    else // ¿A que "if" pertenece este "else"?
        cout << "i <= 5" << endl;
else // Matches "if(i < 10)"
    cout << "i >= 10" << endl
```

## 2.4 Programación Estructurada

### Control de Flujo: While

En los bucles de control while, do-while, y for, una sentencia se repite hasta que la expresión de control sea **false**.

```
while(expresion) sentencia
```

La expresión se evalúa una vez al comienzo del bucle y cada vez antes de cada iteración de la sentencia

```
int main()
{
    int secreto = 15;
    int apuesta = 0;
    // "!=" Diferente de
    while(guess != secret)
    {
        cout << "Adivina el numero: ";
        cin >> apuesta;
    }
    cout << "Lo adivinaste!" << endl;
}
```

## 2.4 Programación Estructurada

### Control de Flujo: do-while

El **do-while** es diferente del **while** ya que la sentencia siempre se ejecuta al menos una vez, aun si la expresión resulta **false** la primera vez. En un **while** normal, si la condición es falsa la primera vez, la sentencia no se ejecuta nunca.

```
int main()
{
    int secreto = 15;
    int apuesta; // No hace falta inicializar porque ejecutamos el bucle.
    do {
        cout << "Adivina el numero: ";
        cin >> apuesta; // Se inicializa aqui.
    } while(apuesta != secreto);
    cout << "Lo adivinaste!!" << endl;
}
```

## 2.4 Programación Estructurada

### Control de Flujo: for

Un bucle **for** realiza una inicialización antes de la primera iteración. Luego ejecuta una evaluación condicional y, al final de cada iteración, efectúa algún tipo de “siguiente paso”. La estructura del bucle **for** es:

```
for(initializacion; condicional; paso)
    sentencia
```

```
int main()
{
    for(int i = 0; i < 128; i = i + 1)
    {
        if (i != 26) // ANSI Limpiar pantalla
            cout << " value: " << i
                << " character: "
                << char(i) // Conversion de entero a al carácter ASCII
                << endl;
    }
}
```

## 2.4 Programación Estructurada

### Control de Flujo: switch

Viene a ser un **if** anidado con una estructura más legible. Muy usado.

```
switch(selector)
{
    case valor-entero1 :
        sentencia; break;
    case valor-entero2 :
        sentencia; break;
    case valor-entero3 :
        sentencia; break;
    case valor-entero4 :
        sentencia; break;
    case valor-entero5 :
        sentencia; break;
        (...)
    default: sentencia;
}

int salir = 0;
while(!salir)
{
    cout << "Select a, b, c or q to quit: ";
    char response;
    cin >> response;
    switch(response)
    {
        case 'a' : cout << "you chose 'a'" << endl;
                    break;
        case 'q' : cout << "quitting menu" << endl;
                    quit = true;
                    break;
        default : cout << "Please use a or q!"
                    << endl;
    }
}
```

## 2.4 Programación Estructurada

### Operadores

Un operador se puede ver como un tipo especial de función.

Un operador recibe uno o más argumentos y devuelve un nuevo valor.

Los argumentos se pasan de forma diferente.

El valor es producido sin modificar los operandos excepto con los operadores de asignación (=), incremento (++) y decremento (--)

### Precedencia

En una misma sentencia puedo utilizar varios operadores:

```
A = X + Y - 2 / 2 + Z;
```

Como regla general:

- La multiplicación y la división se ejecutan antes que la suma y la resta.
- Si no se entiende bien usa los paréntesis.

```
A = X + (Y - 2) / (2 + Z);
```

## 2.4 Programación Estructurada

### Operadores matemáticos

Adición	: +	Sustracción	: -
Multiplicación	: *	División	: /
Resto	: % (Resto de división entera 17%3 es 2)		
C++ permite <code>x = x + 3;</code> ó <code>x += 3;</code>			

### Operadores Relacionales

Menor que	: <	Mayor que	: >
Menor o igual que	: <=	Mayor o igual que	: >=
Equivalente	: ==	No equivalente	: !=

Se pueden utilizar para todos los tipos predefinidos de C++.

Podemos definir su comportamiento para tipos o clases creadas por nosotros. Esto es sobrecarga de operadores.

```
if (rectangulo_a == rectangulo_b) ...
```

### Operadores Lógicos

AND	: &&	OR	:
-----	------	----	---

```
if ((i < j) && ((a == 5) || (a == 10)) ...
```

## 2.4 Programación Estructurada

### Operadores para bits

AND : &

Produce 1 en la salida si ambos operadores son 1. En otro caso 0.

OR : |

Produce 0 en la salida si ambos operadores son 0. En otro caso 1.

XOR : ^

Produce 1 en la salida si solo un operador es 1. En otro caso 0.

NOT : ~

Desplazamiento : >> derecha << izquierda

### Operadores Unarios

Not : - El más unario no hace nada ( x = (+a))

Incremento : ++ Decremento : --

Cuidado con los punteros!!

Dirección de : & Referencia : \* y ->

## 2.4 Programación Estructurada

### El Operador Ternario

Es verdaderamente un operador porque produce un valor, al contrario de la sentencia ordinaria if-else.

```
<expresion1> ? <expresion2> : <expresion3>;
```

```
a = --b ? b : (b = -99);
```

Primero decrementa b.

Si b decrementado es distinto de cero

```
a = b;
```

Si b decrementado es cero

```
b = -99;
```

```
a = (b = -99) -> true (1)
```

### El Operador coma

Además de separar nombres de variables en las listas de argumentos de funciones puede utilizarse como operador para separar expresiones.

Mejor lo utilizamos sólo para separar. Poco utilizado.

## 2.4 Programación Estructurada

### Operadores de cambio de tipo. **Casting** o Moldeado

El concepto es “Colocamos en un molde de un entero un long”.

El compilador cambiará automáticamente de un tipo a otro si esto tiene sentido. Insertará una función que hace el cambio de un tipo de datos al otro tipo.

Hay que utilizarlo con mucho cuidado y en pocas ocasiones. No tiene sentido cambiar de float a int constantemente; seguramente habrá que revisar el diseño de las estructuras de datos.

```
int i = 0x7fff; // Maximo valor= 32767
long l;
float f;
l = i; // Sin necesidad de cast
f = i; // Sin necesidad de cast
// También funciona
l = static_cast<long>(i);
f = static_cast<float>(i);

i = l; // Posible perdida de digitos
i = f; // Posible perdida info,
i = static_cast<int>(l); // no warning
i = static_cast<int>(f); // no warning
char c = static_cast<char>(i);
```

## 2.4 Programación Estructurada

### Operador *sizeof*

**sizeof** da información acerca de la cantidad de memoria utilizada para los datos.

Puede darnos:

- el número de bytes utilizado por una variable.
- el número de bytes utilizado por un tipo de datos.

Es un operador, no una función, y en el caso de una variable puede usarse sin paréntesis.

```
#include <iostream>
using namespace std;
int main(void)
{
    cout << "sizeof(double) = " << sizeof(double);
    cout << ", sizeof(char) = " << sizeof(char) << endl;
    int x;
    int i = sizeof x;
}
```

## 2.4 Programación Estructurada

### Tipos de Datos

Los tipos de datos definen el modo en que se usa el espacio (memoria) en los programas.

Especificando un tipo de datos, estamos indicando al compilador como crear un espacio de almacenamiento en particular, y también como manipular este espacio.

#### **Tipo de dato Predefinido:**

Es intrínsecamente comprendido por el compilador. Estos tipos de datos son casi idénticos en C y C++.

#### **Tipo de dato definido por el usuario:**

También se les llama tipos de datos abstractos y son por ejemplo las clases. El compilador aprende a manejar estos tipos de datos leyendo los ficheros que contienen las declaraciones de las clases.

# 2.4 Programación Estructurada

## Tipos de Datos: Básicos

Los valores máximo y mínimo que pueden albergar los distintos tipos de datos predefinidos se definen en los ficheros de cabecera de sistema ***limits.h*** y ***float.h***

```
#ifndef _LIBC_LIMITS_H_
#define _LIBC_LIMITS_H_ 1
/* Minimum and maximum values a `signed short int' can hold.  */
#  define SHRT_MIN      (-32768)
#  define SHRT_MAX      32767

/* Maximum value an `unsigned short int' can hold.  (Minimum is 0.)  */
#  define USHRT_MAX     65535

/* Minimum and maximum values a `signed int' can hold.  */
#  define INT_MIN       (-INT_MAX - 1)
#  define INT_MAX       2147483647

/* Maximum value an `unsigned int' can hold.  (Minimum is 0.)  */
#  define UINT_MAX      4294967295U
```

# 2.4 Programación Estructurada

## Tipos de Datos: Básicos

<b>char</b>	Para almacenar caracteres	8 bits (un byte)
<b>int</b>	Número entero	16 bits (2 bytes)
<b>float</b>	Número coma flotante precisión simple	utilizar sizeof
<b>double</b>	Número coma flotante precisión doble	utilizar sizeof.

Se pueden definir variables en cualquier sitio en un ámbito determinado, y podemos definir las e inicializarlas al mismo tiempo.

```
// Solo definicion:
char proteina;
int carbohidratos;
float fibra;
double grasa;
// Definicion e inicializacion:
char pizza = 'A', palomitas = 'Z';
int ganchitos = 100, almendras = 150, pipas = 200;
float chocolate = 3.14159;
// Notación exponencial:
double hamburguesa = 6e-4; // Seis por diez elevado a cuatro
```

## 2.4 Programación Estructurada

### Tipos de Datos: Especificadores

#### **long y short :**

Modifican máximos y mínimos.

#### **signed y unsigned :**

Como utilizar el bit de signo en enteros y caracteres.

Para enteros: short int, int , long int

Pueden ser del mismo tamaño si satisfacen mínimos y máximos.

Si la palabra del sistema es de 64 bits todos pueden ser de 64 bits.

Para números en coma flotante: float, double y long double. “long float”

```
char c;  
unsigned char cu;  
int i;  
unsigned int iu;  
short int is;  
short iis; // Lo mismo que short int
```

```
unsigned short int isu;  
unsigned short iisu;  
long iil; // Igual que long int  
long float; // No valido.  
long double ld;  
unsigned float ff; // No valido.
```