TEMA 1: REPASO. ORIENTACIÓN A OBJETOS

ALGORITMOS Y ESTRUCTURAS DE DATOS

M. Colebrook Santamaría
J. Riera Ledesma
J. Hernández Aceituno
J. Molina Gil

Objetivos

- El lenguaje de programación C++
- Estructuras selectivas y repetitivas
- Estructuras de datos: vectores, estructuras, uniones
- Gestión dinámica de la memoria y punteros
- Subprogramas y Funciones
- Clases y objetos
- Funciones. Sobrecarga de funciones
- Abstracción y encapsulación
- Atributo, método, constructor, destructor

El lenguaje de programación C++

- El **lenguaje de programación C++** es una extensión del lenguaje C con el concepto adicional de las **clases.**
- Mantiene la eficiencia de C a **bajo nivel** (controladores, sistemas operativos, software empotrado).
- Añade la posibilidad de desarrollar aplicaciones de **alto nivel** (juegos, bases de datos, apps de escritorio).

Compilación de un fichero C++

- La extensión normal de un fichero en C++ es "**.cpp**".
- Por ejemplo, si tenemos este fichero **hola.cpp**:

```
#include <iostream>  // librería de entrada/salida
using namespace std; // espacio de nombres "std"

int main() {         // imprimimos por pantalla
  cout << "¡Hola mundo!" << endl;
  return 0;
}
```

- Compilamos con: `$ g++ hola.cpp -o hola`
- Ejecutamos con: `$ ./hola`

Entrada/Salida en C++

- La entrada/salida básica en C++ requiere la cabecera

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

- La salida a pantalla se hace mediante un *stream* ya definido, denominado `cout`:

```
cout << "¡Hola mundo!" << endl;
```

- `endl` equivale al carácter `'\n'` (salto de línea).
- La entrada se realiza mediante un *stream* ya definido, denominado `cin`:

```
int i;
cin >> i;
```

Comentarios y constantes

- Los comentarios de una línea comienza por `//`

```
int x;  // Puede aparecer después de código
```

- Los comentarios multilínea se escriben entre `/*` y `*/`

```
/* son útiles para documentar
clases y funciones */
```

- Las constantes en C se definen como macros:

```
#define MAX 10
```

- En C++, se definen con la etiqueta `const`:

```
const int MAX = 10;
```

- Hay más usos de `const` con otras expresiones que veremos a lo largo del curso.

Declaración de variables

```
int main() {            // Se puede declarar varias a la
  int v[10], s = 0;     // vez si son del mismo tipo base

  for (int i = 0; i < 10; i++)  // Puede hacerse en
    s += v[i];                  // cualquier lugar

  int s2 = 0;                   // Se puede reusar el
  for (int i = 0; i < 10; i++)  // nombre al cambiar
    s2 += v[i] * v[i];          // de ámbito

  int m = 1;
  for (int i = 0; i < 10; i++)
    m *= v[i];
}
```

Tipos de datos de las variables (1)

- Tipos de datos para números enteros:

Tipo	Tamaño	signed		unsigned	
`char`	1 byte	-128	127	0	255
`short`	2 bytes	-32768	32767	0	65535
`int`	4 bytes	-2^{31}	$2^{31} - 1$	0	$2^{32} - 1$
`long`	8 bytes	-2^{63}	$2^{63} - 1$	0	$2^{64} - 1$

- Por defecto todos los tipos de datos enteros son con signo. Añadir el modificador `unsigned` desplaza todo su rango a los valores positivos.
- `char` también se usa para almacenar caracteres (letras).

Tipos de datos de las variables (2)

- Tipos de datos para números reales (también llamados *en coma flotante* o *en punto flotante*):

Tipo	Tamaño	Rango	
`float`	4 bytes	$\pm 1.2 \cdot 10^{-38}$	$\pm 3.4 \cdot 10^{38}$
`double`	8 bytes	$\pm 2.2 \cdot 10^{-308}$	$\pm 1.8 \cdot 10^{308}$
`long double`	10 bytes	$\pm 3.4 \cdot 10^{-4932}$	$\pm 1.2 \cdot 10^{4932}$

- Existe un tipo de dato `bool` (1 byte) que sólo adquiere los valores booleanos `false` (0) y `true` (1).
- Aunque existen algunas convenciones básicas para los tipos de datos en C++, los tamaños y rangos pueden variar según el compilador y el sistema operativo utilizados.

Estructuras selectivas: `if-else`

- La estructura **`if`** permite ejecutar fragmentos de código de forma condicional. Los bloques **`else`** son opcionales.

```
if (x < 1) {
  cout << "x < 1" << endl;
} else if (x > 1) {
  cout << "x > 1" << endl;
} else {
  cout << "x == 1" << endl;
}
```

- Si un bloque condicional o un bucle contienen una sola línea, las llaves { } son opcionales.

```
if (x == 1)
  cout << "x == 1" << endl;
else
  cout << "x != 1" << endl;
```

Estructuras selectivas: `switch-case` (1)

- Equivalente a varios `if` de igualdad anidados.

```
switch (x) {
  case 0:
    cout << "A\n";
    break;
  case 1:
    cout << "B\n";
    break;
  default:
    cout << "C\n";
}
```

Es equivalente a:

```
if (x == 0)
  cout << "A\n";
else if (x == 1)
  cout << "B\n";
else
  cout << "C\n";
```

Estructuras selectivas: `switch-case` (2)

- Si no se incluye la instrucción **`break`** al final de un bloque, la ejecución continúa hasta encontrar un **`break`** o hasta llegar al final del **`switch`**. La última opción no necesita **`break`**.

```
switch (x) {
  case 'a': cout << "a or ";
  case 'b': cout << "b" << endl;
            break;
  case 'c': cout << "c" << endl;
}
```

Si `x == 'a'`, la salida será `"a or b"`.
Si `x == 'b'`, la salida será `"b"`.

- La etiqueta **`default`** es opcional. Si existe, debe ir al final.

Estructuras selectivas: ?:

- El operador ternario `?` permite evaluar condiciones dentro de otras instrucciones. Su sintaxis es la siguiente:

  `B ? V : F`

  donde `B` es una condición booleana y la expresión devolverá el valor `V` si `B` es cierta y el valor `F` si `B` es falsa.

- Por ejemplo,

```
y = x < 0.5 ? 0 : 1;
```

También podría escribirse como

```
x < 0.5 ? y = 0 : y = 1;
```

equivale a

```
if (x < 0.5)
  y = 0;
else
  y = 1;
```

Estructuras repetitivas (1)

- Un bucle **while** se repite mientras se cumpla su condición.

```
int i = 0;
while (i < 10) {
  cout << "i= " << i << endl;  // de 0 a 9
  i++;
}
```

- Un bucle **do-while** es equivalente, pero siempre se ejecuta al menos una vez.

```
int j = 0;
do {
  cout << "j= " << j << endl;  // de 0 a 9
  j++;
} while (j < 10);
```

Estructuras repetitivas (2)

- Los bucles **for** tienen la siguiente estructura:

```
for (inicialización; condición; actualización) {}
```

```
for (int k = 0; k < 10; k++) {
  cout << "k= " << k << endl;  // de 0 a 9
}
```

- Se puede usar más de una variable en el mismo bucle.

```
for (int k = 0, m = 0; k < 10; k++, m--) {
  cout << "k= " << k << ", m= " << m << endl;
}
```

Vectores o *arrays*

Run C++

- Un vector es una estructura de datos para almacenar una colección de valores del mismo tipo de datos.

```
int A[3];  // array de 3 enteros

// asignación de valores
A[0] = 1;
A[1] = 2;
A[2] = 3;

// forma alternativa de asignación
int B[] = { 1, 2, 3 };
```

- No es necesario indicar el tamaño si se puede deducir de la inicialización.

Matrices o *arrays* multidimensionales

- Un array puede contener a su vez otros arrays, creando una estructura multidimensional.

```
int C[2][2] = { { 0, 1 }, { 2, 3 } };

// La fila C[0] contiene el array { 0, 1 }
// La fila C[1] contiene el array { 2, 3 }

C[0][0] = 0;
C[0][1] = 1;  // otra forma de
C[1][0] = 2;  // inicialización
C[1][1] = 3;
```

Run C++

Run C++

Cadenas de caracteres

- Los arrays de **char** se conocen también como *cadenas de caracteres.* C/C++ proporciona herramientas para manejarlos de forma eficiente (librería `cstring`).
- Para que C/C++ pueda mostrar correctamente una cadena, su último carácter debe ser `'\0'` (equivalente al valor *numérico* `0`, no al *carácter* `'0'`).
- El carácter de terminación se añade automáticamente al inicializar una cadena o al leerla de teclado o fichero.

```
char c[] = { 'h', 'o', 'l', 'a', '\0' };
```
equivale a
```
char c[] = "hola";
```

Punteros (1)

- Un puntero es una variable que contiene una **dirección de memoria**.

```
int*   p;       // Puntero a un entero
int *p;  // Sintaxis alternativa
```

- El operador & devuelve la **dirección de memoria** de una variable.
- El operador * devuelve la **variable** a la que hace referencia un puntero.
- La constante NULL (definida en stdio.h → iostream) representa un **puntero nulo** y tiene valor 0 (C++11 introdujo el valor nullptr).

```
int             i               =               10;
p  =  &i;       //  p  toma  la  dirección  de  i
*p   =   20;        //   i   toma   el   valor   20
p = NULL;  // p toma el puntero nulo
```

Punteros (2)

Run C++

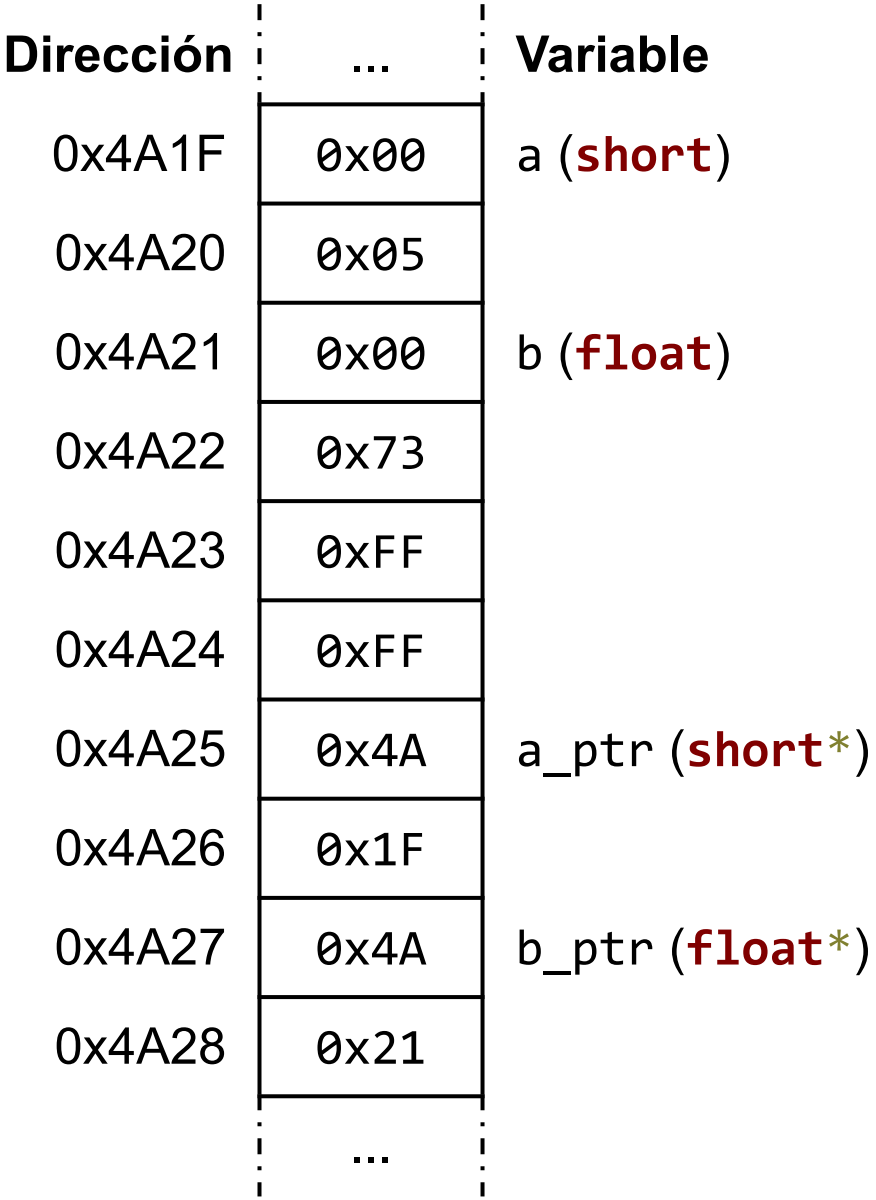
```
int main() {
  short a = 5;
  float b = 12.6;

  short *a_ptr = &a;
  float *b_ptr = &b;

  cout << "a=" << a << '\t';
  cout << "a_ptr=" << a_ptr;
  cout << "\t*a_ptr=";
  cout << *a_ptr << endl;

  cout << "b=" << b << '\t';
  cout << "b_ptr=" <<  b_ptr;
  cout << "\t*b_ptr=";
  cout << *b_ptr << endl;
}
```

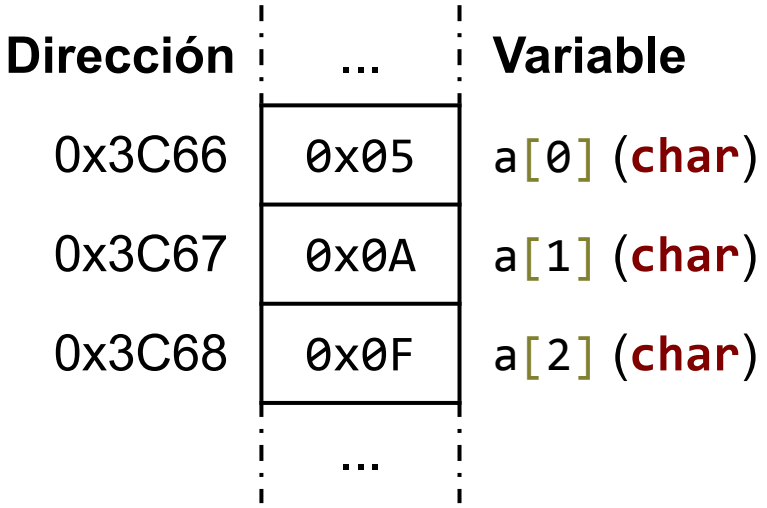
```
a=5      a_ptr=0x4a1f   *a_ptr=5
b=12.6   b_ptr=0x4a21   *b_ptr=12.6
```



Punteros (3)

- Las variables de tipo array son **punteros a zonas de memoria**. Su valor es el de la posición del primero de sus elementos.

```
char a[3];
a[0] = 5;
a[1] = 10;
a[2] = 15;
cout << "a[1]=" << a[1];
cout << endl << "a=" << a;
```




```
a[1]=10
a=0x3C66
```

Punteros (4)

Run C++ Run C++

- El nombre de un vector es un puntero **constante** al primer elemento, y se puede usar como cualquier puntero.

```
int vec[30], *ptr;

ptr=vec;        // equivale a ptr=&(vec[0])
ptr=vec+5;      // ptr apunta a vec[5]
*(vec+2)= 5;    // equivale a vec[2]=5
vec++;          // ILEGAL vec es constante
```

Punteros (5)

Run C++

```
int main() {
  int    a[] = {2, 3, 5, 7};
  double b[4];

  int    *a_ptr = a;
  double *b_ptr;
  b_ptr = b;

  for(int i = 0; i < 4; i++)
    b_ptr[i] = a_ptr[i] + 0.5;

  for(int i = 0; i < 4; i++)
    cout << setw(2) << a[i]
         << setw(3) << *(a_ptr + i)
         << setw(5) << b_ptr[i]
         << setw(5) << *(b + i) << endl;
}
```

- v+i ≡ &v[i]
- *(v+i) ≡ v[i]

```
 2  2  2.5  2.5
 3  3  3.5  3.5
 5  5  5.5  5.5
 7  7  7.5  7.5
```

Punteros (6)

Run C++

```
int a[5];
int *p = NULL;

p = a;        *p = 10;        //   p   → a[0]
p++;          *p = 20;        //   p   → a[1]
p = &a[2];    *p = 30;        //   p   → a[2]
p = a + 3;    *p = 40;        //   p   → a[3]
p = a;        *(p+4) = 50;    // p+4 → a[4]

for(int i = 0; i < 5; i++)
  cout << ' ' << a[i];
```

```
10 20 30 40 50
```

Gestión dinámica de la memoria

- Las peticiones de memoria en **tiempo de ejecución** se gestionan con los operadores **new** y **delete**:

```
int *p = NULL;
p = new int;       // Entero en memoria dinámica
delete p;          // Destrucción del entero
p = NULL;          // Es buena práctica ponerlo a NULL

p = new int[10]; // Vector de 10 enteros
delete[] p;        // Destrucción del vector
p = NULL;          // Es buena práctica ponerlo a NULL

int n;
cin >> n;          // n > 0
p = new int[n];  // Vector de n enteros
delete[] p;        // Destrucción del vector
p = NULL;          // Es buena práctica ponerlo a NULL
```

Referencias

- Las referencias nos permiten crear un nuevo nombre (alias) para una variable:

```
int i;
int *pi = &i;  // puntero a i
int &ri = i;   // referencia a i
int& ri = i;   // sintaxis alternativa
```

- Las siguientes instrucciones escriben en la misma posición de memoria y tienen el mismo efecto:

```
i = 10;      *pi = 10;      ri = 10;
```

Estructuras (`struct`)

- Permite crear registros con varios campos.

```
struct NumLetra {
  int n;
  char c;
};
int main() {
  NumLetra x = { 3, 'a' };  // inicialización
  x.n = 3;  // inicialización clásica
  x.c = 'a';
}
```

- Más adelante veremos la relación entre **`struct`** y **`class`**.

Punteros a estructuras

- El operador `->` permite acceder a los campos de un **`struct`** desde un puntero al mismo.

```
struct Num {
  float f;
};
int main() {
  Num *n = new Num;
  n->f = 3.14;  //  equivalente a (*n).f
  delete n;
}
```

- También se usa con punteros a objetos de tipo clase.

Uniones (union)

Run C++ Run C++

- Es igual al **struct**, pero todos los campos **comparten** la misma zona de memoria, por lo que permite almacenar **un solo valor**.

```
union Mix {
  unsigned char c[4];                    // 4 bytes
  struct { unsigned short hi, lo; } s;   // 4 bytes
  unsigned int i;                        // 4 bytes
};

int main() {
  Mix m;                 //     FF       00       F0       0F
  m.i = 0xFF00F00F;      // 11111111 00000000 11110000 00001111
  cout << hex << m.i << endl;      // FF00F00F
  cout << hex << m.s.lo << endl;   // FF00
  cout << hex << m.s.hi << endl;   //     F00F
  cout << hex << m.c[3] << endl;   // FF
  cout << hex << m.c[2] << endl;   //   00
  cout << hex << m.c[1] << endl;   //     F0
  cout << hex << m.c[0] << endl;   //       0F
}
```

Funciones (1)

- Una función es un fragmento de código que se puede invocar desde otro punto del programa. Pueden o no recibir parámetros (o argumentos) y/o devolver un valor.

```
void f(float a) {     // No devuelve un valor
    cout << "Se muestra " << a << endl;
}

float g() {                    // Devuelve float
                        return              5.0;
}

int                     main()                  {
   f(2 * g() + g());    // "Se muestra 15.0"
}
```

Funciones (2)

- Se pueden dar valores por defecto a los parámetros.

```
int f(int a, int b = 2, int c = 3) {
  return a + b + c;
}

void g() {
  f(1);          // 1 + 2 + 3 = 6
  f(1, 1);       // 1 + 1 + 3 = 5
  f(1, 1, 1);    // 1 + 1 + 1 = 3
}
```

- Todos los parámetros por defecto deben estar al **final** de la lista de parámetros.

Funciones (3)

- Hay varias formas de pasar parámetros a una función:

Paso por valor: Sólo se pasa una copia del valor, la variable original (si existe) no se modifica.	**Paso por dirección (o puntero):** Se pasa un puntero a una variable, su contenido puede ser modificado.	**Paso por referencia:** Se crea una referencia a la variable, su contenido puede ser modificado.

`void f(int p) {`
`  p = 2;`
`}`
`int a = 1;`
`f(a);`
`// a vale 1`
	`void f(int *p) {`
	`  *p = 2;`
	`}`
	`int a = 1;`
	`f(&a);`
	`// a vale 2`
		`void f(int &p) {`
		`  p = 2;`
		`}`
		`int a = 1;`
		`f(a);`
		`// a vale 2`

Funciones (4)

- La etiqueta **`inline`** *sugiere* al compilador que maneje una función como si fueran una macro, de forma eficiente y automática.

```
#define SQR(x)  (x)*(x)
int a = SQR(2);   // se expande como a=(2)*(2)=4
int b = SQR(2+3); // se expande como b=(2+3)*(2+3)=25
int c = SQR(a++); // se expande c=(a++)*(a++)=20, y a=6

inline int sqr(int x) { return x*x; }
int a = sqr(2);   // se expande como a=2*2=4
int b = sqr(2+3); // se expande como b=5*5=25
int c = sqr(a++); // se expande como c=a*a=16, y a=5

inline int f(int a) { int x = a; while (--a) x *= a; return x; }
// Muy compleja, el compilador podría ignorar la etiqueta inline
```

Sobrecarga de funciones

- Se puede reutilizar el nombre de una función siempre que sus parámetros sean diferentes.

```
void print(int);
void print(const char*);
void print(float);
```

- Esto no se aplica al valor devuelto, ya que no se podría identificar a qué función hace referencia una llamada.

```
void f(int);
int f(int); // ERROR
```

Clases y objetos (1)

- Una clase es una estructura que permite la descripción de las **características** (**atributos**) y **comportamiento** (**métodos**) de un conjunto de objetos. Un *objeto* es una variable de tipo *clase*.

```
[class | struct | union] NombreClase {
[private       |       public       |       protected]:
                                          Atributo1;
                                          Atributo2;
  ...
[private       |       public       |       protected]:
                                             Método1;
                                             Método2;
  ...
};
```

Clases y objetos (2)

```
class Rectangulo {

private:
  int x, y;

public:
  void setX(int a) { x = a; }
  void setY(int b) { y = b; }

  int getX() { return x; }
  int getY() { return y; }

  int area() { return x * y; }
};

Rectangulo r;
r.setX(2);
r.setY(5);
int a = r.area(); // a = 10
```

```
struct Rectangulo {

private:
  int x, y;

public:
  void setX(int a) { x = a; }
  void setY(int b) { y = b; }

  int getX() { return x; }
  int getY() { return y; }

  int area() { return x * y; }
};
```

No es conveniente usar **struct** como **class** porque es menos eficiente respecto a los métodos y al paso de parámetros.

Clases y objetos (3)

- Es posible definir los métodos de una clase fuera de la misma. Esto permite estructurar mejor el código.
- Una función miembro definida dentro de la definición de la clase se asume como función **inline**.

```
class C {
private:
  int r;
public:
  void setR(int R) { r = R; }
  int getR() { return r; }
};
```

```
class C {
private:
  int r;
public:
  void setR(int);
  int getR();
};


void C::setR(int R) {
  r = R;
}
int C::getR() {
  return r;
}
```

Clases y objetos (4)

- Por organización, se suele separar el código en varios ficheros.

C.hpp	C.cpp	main.cpp

`#pragma once`
 	
`class C {`
 	
`  private:`
`   int r;`
 	
`  public:`
`   void setR(int);`
`   int getR();`
`};`
	`#include "C.hpp"`
	
	`void C::setR(int R) {`
	`   r = R;`
	`}`
	
	`int C::getR() {`
	`   return r;`
	`}`
		`#include "C.hpp"`
		
		`int main() {`
		`   C obj;`
		`   obj.setR(5);`
		`   int a = obj.getR();`
		`}`

```
> g++ -c C.cpp → genera C.o
> g++ main.cpp C.o -o main
```

o también:

```
> g++ main.cpp C.cpp -o main
```

Clases y objetos (5)

- Cada atributo y función miembro tiene una visibilidad con respecto al resto de las clases y funciones:
  - `public`: todo el mundo puede acceder y modificar el atributo o método. Es la opción por **defecto** en `struct`.
  - `private`: sólo los miembros de la misma clase pueden modificar, leer los atributos y ejecutar los métodos. Es la opción por **defecto** en `class.`
  - `protected`: sólo miembros de la misma clase y clases derivadas pueden modificar, leer los atributos y ejecutar las funciones.
- Es de buen estilo de programación establecer explícitamente qué partes de la clase son **privada**, **pública** y **protegida**.

Tipos de métodos (1)

- **Selectores (o *getters*)**

  Evalúan el estado de un objeto. Son útiles para extraer el valor de los atributos privados.

- **Transformadores (o *setters*)**

  Alteran el estado del objeto. Son útiles para modificar el valor de los atributos privados.

- **Iteradores**

  Permiten recorrer el contenido del objeto. Normalmente se aplican a objetos que almacenan otros objetos (contenedores).

Tipos de métodos (2)

- **Constructores**

  Se ejecutan al crear (*instanciar*) un objeto.

  Tienen el mismo nombre que la clase a la que pertenecen y **no devuelven ningún tipo**, ni siquiera `void`.

  Suelen usarse para dar un valor inicial a los atributos.

  Un constructor sin parámetros o con todos los parámetros por defecto es un ***constructor por defecto***.

  Para poder definir vectores de objetos, deben pertenecer a una clase que tenga un constructor por defecto.

Tipos de métodos (3)

- **Destructores**

  Se ejecutan automáticamente cuando un objeto deja de existir.

  Tienen el mismo nombre que la clase a la que pertenecen, precedido de ~.

  No tienen parámetros, no devuelven nada y no pueden ser sobrecargados.

  Suelen usarse para liberar memoria dinámica si la clase tiene atributos de tipo puntero.

Tipos de métodos (4)

```
class Valor {
private:
  int x;
public:
  Valor(const int a) { x = a; }      // Constructor
  ~Valor() { }                       // Destructor
  int getX() { return x; } const;    // Getter
  void setX(const int a) { x = a; }  // Setter
};
int main() {
  Valor v(3);
  cout << v.getX();
}
```

Si un **método** (que no sea constructor ni destructor) nunca modifica los atributos del objeto, conviene declararlo **const**.

Si un **parámetro** no va a modificarse dentro de su función, es de buen estilo de programación declararlo **const**.

Constructores sobrecargados

- Un constructor puede ser **sobrecargado**.

```
class Rectangulo {

private:
  int x, y;

public:
  Rectangulo(const int a, const int b) { x = a; y = b; }
  Rectangulo(const int a = 1) { x = a; y = a; }
};
```

```
Rectangulo r(3,2);  // 3 x 2
Rectangulo s(5);    // 5 x 5
Rectangulo t;       // 1 x 1 (por defecto)
```

El puntero `this`

- Dentro de los métodos, el puntero **`this`** hace referencia a la clase a la que estos pertenecen.

```
class Clase {
private:
  float x;
public:
  void setX(const float x) {
    this->x = x;  // El atributo x de Clase recibe
  }               // el valor del parámetro x
};
```

Parámetros de tipo clase

- Para evitar copias potencialmente inestables de estructuras complejas, es conveniente que el paso de parámetros de tipo clase se haga siempre por referencia.

```
class Matriz {
  double m[1024][1024];
  ...
};

void transpuesta(const Matriz &src, Matriz &dst) {
  ...          // evita copiar 8Mb por cada parámetro
}
```

Operadores

- C++ permite sobrecargar la mayoría de los operadores (+, -, *, /, ++, --, <<, >>, [ ], ( ), <, >, =, ==, ...), definiéndolos como funciones o métodos de clase.

```
class A {
...
    bool operator<(const A &a) const
    { return x_ < a.x_; }
};

A operator+(const A &a1, const A &a2)
{ return A(a1.getX() + a2.getX()); }

ostream& operator<<(ostream &out, const A &a)
{ out << a.getX(); return out; }
```

```
A b, c;
if (b < c)
    cout << b+c;
```

Herencia (1)

- Se pueden crear clases a partir de otras anteriores mediante la herencia. Todos los miembros **public** y **protected** son visibles a la clase derivada.

```
class A {
    private: int i;
    protected: int j;
    public: int getI() { return i; };
};

class B: public A {
    private:
    public: int getJ() {
};
```

```
int main() {
    A x;
    B k;
    cout << x.getI();
    return j;
    cout << x.getJ();
```

Herencia (2)

- Se puede indicar el máximo nivel de visibilidad de los miembros heredados (por defecto, **private**).

```
class A {
  private:
   int i;
  protected:
   int j;
  public:
   int k;
};
```

```
class B: public A {        // B.i private
  ...                      // B.j protected
};                         // B.k public

class C: protected A {     // C.i private
  ...                      // C.j protected
};                         // C.k protected

class D: private A {       // D.i private
  ...                      // D.j private
};                         // D.k private
```

Funciones y clases `friend`

- Una función o una clase cuya cabecera aparece etiquetada como **`friend`** dentro de la declaración de una clase tiene permiso para acceder a los miembros **`private`** y **`protected`** de la misma, sin ser un método ni una clase derivada.

```
class A {
                                        private:
  double x;
       friend   double   func(const   A   &a);
};
double func(const A &a) { return a.x; }
```

Plantillas (`template`) (1)

- Las plantillas (**`template`**) permiten que una clase, función o variable trabaje con diferentes tipos de datos sin tener que reescribir el código para cada tipo.

```
template<class T>
void intercambio(T& a, T& b) {
  T tmp = a;  // Se pueden crear variables de tipo T
  a = b;
  b = tmp;
}

int a = 1, b = 2;
intercambio<int>(a, b);  // a = 2, b = 1
cout << "a = " << a << ", b = " << b << endl;
```

Plantillas (`template`) (2)

- Las plantillas de clases (***class templates***) permiten que los miembros puedan usar los parámetros de la template como tipos.

```
template<class T>
class punto_t {
 private:
  T x, y;
 public:
  punto_t(const T a, const T b): x(a), y(b) {}
  // punto_t(const T a, const T b) { x = a; y = b; }
  ostream& write(ostream& os) {
    return os << "x=" << x << ", y=" << y << endl;
  }
};
```

```
punto_t<int> i(3, 4);
i.write(cout); // x=3, y=4
```

```
punto_t<float> f(1.23, 5.67);
f.write(cout); // x=1.23, y=5.67
```

Espacios de nombres (1)

- Los espacios de nombres (**namespace**) se utilizan para evitar conflictos de nombres, permitiendo que las clases y funciones se agrupen bajo un **ámbito** (*scope*) separado.

```
namespace A {
    int x;
}

int main() {
    int x = 0;
    A::x = 1;
}
```

Espacios de nombres (2)

- **using** permite usar elementos de un espacio de nombres sin tener que indicar su prefijo. Por ejemplo, `cout` y `endl` de la librería `iostream` pertenecen al **namespace** `std`.

```
#include <iostream>
int main() {
    std::cout << "Texto" << std::endl;
}
```

equivale a

```
#include <iostream>
using namespace std;
int main() {
    cout << "Texto" << endl;
}
```

Referencias

- Olsson, M. (2018), “C++ 17 Quick Syntax Reference”, Apress. Disponible en PDF en la BBTK-ULL: link.springer.com/content/pdf/10.1007%2F978-1-4842-3600-0.pdf
- Stroustrup, B. (2002), “El Lenguaje de Programación C++”, Addison Wesley.
- Web C++ Reference, disponible en: cppreference.com
- Web sobre C++, disponible en: www.cplusplus.com
- C++ Syntax Highlighting (código en colores): tohtml.com/cpp