C++

ALGORITMOS Y ESTRUCTURAS DE DATOS AVANZADAS

M. Colebrook Santamaría
J.A. González Martínez

Objetivos

- Repaso de C++
  - Variables y constantes
  - Referencias y paso de parámetros
  - Sobrecarga de funciones
  - Argumentos por defecto
  - Funciones `inline`
  - Operadores `new` y `delete`
- Clases
  - Tipos de métodos y métodos `static`
  - Constructores y destructor
  - Funciones `friend`
- Sobrecarga de operadores
- Herencia y clientela
- Polimorfismo y vinculación dinámica
- Plantillas (`template`) y manejo de excepciones

Compilación modular (1)

p1.h

```
extern void g(int);
```

p2.h

```
extern void z(int);
```

p1.cpp

```
#include <iostream>
#include "p1.h"

using namespace std;

static int f(int);

int f(int x)
{ return x*x;
}

void g(int i)
{ cout << f(i) << endl;
}
```

p2.cpp

```
#include <iostream>
#include "p2.h"

using namespace std;

static int f(int);

int f(int x)
{ return x / 2;
}

void z(int i)
{ cout << f(i) << endl;
}
```

Compilación modular (2)

main.cpp

```
#include <iostream>
#include "p1.h"
#include "p2.h"

int main()
{
  g(3);
  z(8);
  return 0;
}
```

makefile

```
main: p1.cpp p1.h p2.cpp p2.h main.cpp
	g++ p1.cpp p2.cpp main.cpp -o main
```

Comentarios y constantes

```
// comentario en C++
/* comentario en C y C++ */

// definición de constante en C
#define MAX 10
// lo que introduce C++
const int MAX = 10;


// macro útil para depurar variables usando #define
#define TRAZA(t) cout << (#t) << "=" << (t) << endl


int i = 10;
TRAZA(i), TRAZA(i*i), TRAZA(2*i);
// i=10, i*i=100, 2*i=20
```

Uso de `assert`

```
#include <assert.h>

...
assert(expresión booleana);
...
```

- Si la condición es `FALSE`, el programa se **detiene** después de mostrar un **mensaje de error** que incluye el **nombre del fichero fuente** en C++ y el **número de línea**.
- Definiendo el símbolo `#define NDEBUG` por **encima** del `#include <assert.h>`, convierte todos los `assert` en sentencias nulas.

Más ejemplos de `const`

```
// contenido constante, pero no el puntero
const char *s1 = "ABC"; // equivale a char const *s1 = "ABC";
s1++;       // OK: s -> 'B'
*s1 = 'X';  // ERROR de compilación

// puntero constante, pero no el contenido
char * const s2 = "ABC";
s2++;      // ERROR de compilación
*s2 = 'X'; // OK: s = "XBC";

// puntero constante, contenido constante
const char * const s3 = "ABC"; // equivale a char const * cont s3 =
"ABC"
s3++;      // ERROR de compilación
```

Declaración de variables en cualquier lugar

```
void main()
{ int v[10];
  int s = 0;
  for (int i = 0; i < 10; i++)
    s += v[i];
  int s2 = 0;
  for (int i = 0; i < 10; i++)
    s2 += v[i] * v[i];
  int m = 1;
  for (int i = 0; i < 10; i++)
    m *= v[i];
}
```

Referencias

```
int i;

// declara un puntero a i
int *pi = &i;

// declara una referencia (alias) a i
int &ri = i;

// las siguientes instrucciones tienen el mismo efecto
i = 10, *pi = 10, ri = 10;
```

Paso de parámetros a función (1)

Paso por valor	Paso por dirección (puntero)	Paso por referencia

`void f(int param)`
`{ param = 10;`
`}`
 	
`int arg = 1;`
`f(arg);`
	`void f(int *param)`
	`{ *param = 10;`
	`}`
	
	`int arg = 1;`
	`f(&arg);`
		`void f(int &param)`
		`{ param = 10;`
		`}`
		
		`int arg = 1;`
		`f(arg);`

Paso de parámetros a función (2)

```
void f(int a, int *b, int &c)
{ a = 3;   // no tiene efecto en el exterior
  *b = 3;  // se modifica el objeto al que apunta b
  b = &a;  // no tiene efecto en el exterior
  c = 4;   // c es a todos los efectos la variable pasada
  b = &c;  // es posible tomar la dirección de la variable pasada
  *b = 5;
}

int main()
{ int a1 = 1, b1 = 2, c1 = 2;

  f(a1, &b1, c1);
  // resultado: a1 = 1, b1 = 3, c1 = 5

  return 0;
}
```

La referencia como valor devuelto de una función

```
int& min(int &a, int &b)
{ return (a < b ? a : b);
}

int main()
{ int i = 1, j = 2;

  min(i, j) = 3; // ¿es esto posible?
  return 0;
}
```

Argumentos con el modificador `const`

```
void f(const int a, const int *b, const int &c)
{ a = 3;  // ERROR: no se puede modificar una constante
  *b = 3; // ERROR: no es const el puntero, sino el contenido
  b = &a; // OK: se puede direccionar una constante
  c = 4;  // ERROR
  b = &c; // OK
  *b = 5; // ERROR
}
```

Sobrecarga de funciones

```
// utilidad de la sobrecarga
void print(int);
void print(const char*);
void print(float);

// CUIDADO al sobrecargar funciones
void f(int);
int f(int); // ERROR: no hay diferencia por el valor
            // devuelto
```

Argumentos por defecto

```
void f(const int i = 0)
{ ...
}

f();  // igual que f(0)
f(3);

void f(const int a, const float b, const int c = 4, const int d = 5, const int e = 6);
f(1, 2.0, 3); // equivale a f1(1, 2.0, 3, 5, 6)
f(1, 2);      // f(1, 2, 4, 5, 6)

// CUIDADO con la sobrecarga de funciones y los argumentos por defecto
void f(const int a);
void f(const int a, int b = 0);
void g()
{ f(1, 2); // segunda f
  f(1);    // ¿a cuál de las dos se refiere?
}
```

Funciones `inline`

```
#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++ = 16, y a = 6 (ERROR)

// el problema anterior se resuelve usando funciones inline
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 = (2+3)*(2+3) = 25
int c = sqr(a++); // se expande como c = a*a = 16, y a++ = 5 (OK!)

// MAL EJEMPLO de función inline
inline int bucle(int a) { int f = a; while (--a) f *= a; return f; }
```

Operadores `new` y `delete`

```
int *p;

p = new int;      // creación de un entero en memoria dinámica
delete p;         // destrucción del entero

p = new int[10]; // creación de un vector de 10 enteros
delete[] p;       // destrucción del vector anterior

const int n = 30;
p = new int[n];  // creación de un vector de n enteros
delete[] p;
```

Clases en C++

- **Clase**: descripción de las características y comportamiento de un conjunto de objetos.

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

```
class miClase
{
  int Dato;
public:
  int Valor() { return Dato; }
}
```

```
struct miClase
{
private:
  int Dato;
public:
  int Valor() { return Dato; }
}
```

Fichero cabecera y clases

- La **interfaz** de cada clase debe estar en un fichero con la extensión `.h` (o `.hpp`) . La interfaz sería la declaración de la clase.
- La **implementación** de cada clase debe ponerse en un fichero con extensión `.cpp` con el mismo nombre.
- En el fichero donde se almacena la **interfaz** se establece una **macro** para evitar que se redeclare la clase.

```
#ifndef __MICLASE__
#define __MICLASE__

class MiClase
{ ...
};

#endif
```

Clases en C++

- 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 la parte privada, pública o protegida de la clase.
- Dentro de los métodos se usa el puntero `this` para acceder a los atributos y a las funciones miembro del mismo objeto.

```
return Dato; // equivale a return this->Dato;
```

Métodos

- Los métodos son las **funciones miembro** de la clase.
- Los métodos pueden ser **públicos** (pueden activarse interna y externamente) o **privados** (sólo pueden activarse por otros métodos internos a la clase).
  - Métodos **constructores**: alteran el estado actual del objeto o lo inicializan.
  - Métodos **selectores**: evalúan y exploran el estado actual del objeto.
  - Métodos **transformadores**: alteran el estado del objeto, modificando alguno de sus atributos.
  - Métodos **iteradores**: permiten recorrer todas las partes del objeto, normalmente aplicados a objetos que almacenan otros objetos (contenedores).
- Una función miembro definida dentro de la definición de la clase se asume como función `inline`.

Miembros `static` de las clases

```
class miClase
{
public:
  static int objetos;
  miClase() { objetos++; }
  static int getObjetos() { return objetos; }
};

int miClase::objetos = 0;

int main()
{ miClase a, b;
  cout << miClase::getObjetos() << endl;
  return 0;
}
```

Constructores y destructor (1)

- Los constructores son funciones con el mismo nombre de la clase que no devuelven **ningún** tipo, ni siquiera `void`.
- Un constructor puede ser **sobrecargado**.
- Cuando se **crea** un objeto en memoria global, local (pila), o dinámica (*heap*) se llama al constructor.
- Un constructor **sin parámetros** o con todos los parámetros por defecto es un **constructor por defecto**.
- Cuando un objeto acaba su vida útil, se emplea una función especial llamada **destructor**, que tiene el mismo nombre de la clase precedido de '~'. **No tiene parámetros**, **no devuelve nada**, y **no puede ser sobrecargado**.
- Para poder definir **vectores de objetos**, es necesario declarar el constructor por defecto.

Constructores y destructor (2)

```
#define DEF_SIZE 1
typedef int TDATO;
class Vector
{
private:
  TDATO *V;
  int first, last;
public:
  Vector(const int = DEF_SIZE);
  ~Vector() { delete[] V; }
  void init(const TDATO&);
  int size() const
  { return last - first + 1; }
  int getFirst() const { return first; }
  int getLast() const { return last; }
  void print(void);
};
```

```
Vector::Vector(const int n)
{ first = 0, last = n - 1;
  V = new TDATO[n];
  init(0);
}

void Vector::init(const TDATO& v)
{ for (int i = getFirst(); i <= getLast(); i++)
    V[i] = v;
}

void Vector::print()
{ for (int i = getFirst(); i <= getLast(); i++)
    cout << V[i] << " ";
  cout << endl;
}
```

Conversión usando constructores

- Un constructor que acepta **un único parámetro** especifica una conversión desde el tipo del argumento al tipo de su clase.

```
class X
{ ...
public:
  X(int);
  X(char *s);
  ...
};
```

```
void f(const X& param)
{ X a = 1;       // X a = X(1); ó X a(1);
  X b = "algo"; // X b = X("algo");
  a = 2;         // a = X(2);
  f(3);          // f(X(3));
  ...
}
```

Clases y funciones `friend`

- Una función **`friend`** de una clase es aquella que, sin ser miembro, se le permite acceder a los elementos **privados** y **protegidos** de ésta.
- Si la declaración `friend` se aplica sobre una clase, **todas** las funciones miembro de la clase se convierten en amigas.

```
class X
{ int v;
  // declarada en cualquier parte
  // de la clase
  friend void f_amiga(X&, int);
public:
    void f_miembro(int i)
    { v = i; }
};
```

```
void f_amiga(X& x, int i)
{ x.v = i;
}


int main()
{ X x;
  f_amiga(x, 10);
  x.f_miembro(10);
}
```

Entrada/Salida en C++ (1)

```
class X
{ ...
public:
  ...
  // declaración de la función friend
  friend ostream& operator<<(ostream&, const X&);
}


// cuerpo de la función
ostream& operator<<(ostream& os, const X& x)
{ // visualiza de forma personalizada el objeto ‘x’
  // NO se usa "cout", sino “os”

  // ¿por qué hay que retornar siempre os?
  return os;
}
```

Entrada/Salida en C++ (2)

```
class X
{ ...
public:
  ...
  // declaración de la función friend
  friend istream& operator>>(istream&, X&);
}


// cuerpo de la función
istream& operator>>(istream& is, X& x)
{ // forma personalizada de entrada de datos en el objeto ‘x’
  // NO se usa "cin", sino “is”

  // ¿por qué hay que retornar siempre is?
  return is;
}
```

Sobrecarga de operadores (1)

- Los únicos operadores que **no** se pueden sobrecargar son:

  `.  .* :: ?: sizeof`

- **No** se pueden **inventar** nuevos operadores.
- Si el operador se define como **miembro** de una clase, el operando izquierdo es el objeto al que se le aplica la función, el otro operando será el parámetro pasado a la función. Además, sólo se puede aplicar **conversores automáticos** al segundo operando.
- En el caso de que se define como **función libre**, los parámetros serán los operandos (normalmente se establecen como `friend`). Se puede aplicar conversores a ambos operandos.
- En caso de operadores unarios se tendrían funciones miembro sin parámetros y funciones libres con un único parámetro.

Sobrecarga de operadores (2)

```
// problemas al definir operador como
// función miembro
class Complejo
{ float im, re;
public:
  Complejo(const float r = 0, const float i = 0)
    { im = i; re = r; }
  friend Complejo operator+(const Complejo&,
                            const Complejo&);
  Complejo operator*(const Complejo&);
};

Complejo operator+(const Complejo& c1,
                   const Complejo& c2)
{ return Complejo(c1.re + c2.re,
                  c1.im + c2.im);
}
```

```
Complejo Complejo::operator*(const Complejo& c)
{ return Complejo(re * c.re - im * c.im,
                  re * c.im + im * c.re);
}

int main()
{ Complejo c1, c2(1.1), c3(2), c4(2.1, 3.2);
  c1 = c2 * c3 + (c3 + c2 + 8) * c4;
  c2 = 10 + c1;
  c4 = 10 * c2; // ERROR de compilación
  return 0;
}
```

Operador de asignación (1)

- Seguimos con la clase Vector de la diapo #20.

```
int main()
{ Vector a(3);
  a.init(1);
  a.print(); // 1 1 1

  Vector *b = new Vector(7);
  b->init(2);
  b->print(); // 2 2 2 2 2 2 2

  a = *b;   // core dumped!!
  delete b;

  return 0;
}
```

Operador de asignación (2)

- ¿Por qué la instrucción `a = *b` es tan **nefasta**? Porque lo único que hace es **una copia de los atributos de un objeto en el otro**. Es decir:
  - Se pierde el puntero de `*V` dentro de `a`.
  - `a` y `*b` **comparten el mismo puntero `*V`** y dejan de ser independientes.
  - La instrucción `delete b` llama al destructor, el cual **libera la zona de memoria de `*V`**. Cuando se destruye `a`, se intenta liberar la **misma zona de memoria**.
- **Solución**: sobrecarga del **operador de asignación**.

Operador de asignación (3)

```
Vector& Vector::operator=(const Vector& x)
   { if (size() < x.size()) {
       delete[] V;
       first = x.getFirst(), last = x.getLast();
       V = new int[x.size()];
     }
     // pasamos todos los elementos desde 'x'
     for (int i = first; i <= last; i++)
       V[i] = x.V[i];
     // ¿por qué hay que retornar siempre *this?
     return *this;
   }
```

Operadores para convertir clases

- Es posible establecer operadores (como funciones miembro) de **conversión de clase** a la que pertenece a otra u otro tipo predefinido.

```
class X
{ ...
public:
  operator int();   // conversor de X a int
  operator char*(); // conversor de X a char*
};

void f(X& a)
{ int i = a;     // i = a.operator int();
  char c[100];
  strcpy(c, a); // strcpy(c, a.operator char*());
}
```

Paso de objetos a funciones y valor devuelto por una función: Constructor de copia (1)

- Seguimos con la clase Vector de la diapo #20.

```
Vector f(Vector v)
{ v.init(1);
// nuevo vector dinámico
  Vector *pv = new Vector(3);
  pv -> init(2);
  pv -> print();
// operador de asignación: OK
  v = *pv;
// eliminamos vector dinámico
  delete pv;

  return v;
}
```

```
int main()
{ Vector v1, v2(5);

  v1 = f(v2); // core dumped!!
  v1.print();

  return 0;
}
```

Paso de objetos a funciones y valor devuelto por una función: Constructor de copia (2)

- ¿Por qué es **errónea** la instrucción `v1 = f(v2)`?
  - `v2` y el parámetro `p` comparten el mismo vector `*V`, dejan de ser independientes.
  - Cuando termina la ejecución de `f`, se llama automáticamente al destructor de `v`, el cual libera el vector `*V`. Cuando se destruye `v2`, una vez terminada la ejecución del `main()`, se intenta liberar la **misma** zona de memoria.
  - Cuando una función devuelve un objeto, se crea un **objeto temporal**, sobre el que se vuelca dicho objeto devuelto. Por tanto, los dos objetos comparten el mismo vector `*V`.
  - Cuando se llama al destructor del objeto devuelto en la función `f`, se libera el vector `*V` que usa `v`. Cuando se destruye automáticamente el objeto temporal, una vez terminado el `main()`, se intenta liberar la **misma** zona de memoria.
- **Solución**: constructor de copia.

Paso de objetos a funciones y valor devuelto por una función: Constructor de copia (3)

```
// constructor de copia
Vector::Vector(const Vector& x)
{ first = x.getFirst(), last = x.getLast();
  V = new int[x.size()];
  for (int i = first; i <= last; i++)
    V[i] = x.V[i];

  // ¿podría reemplazar todo este código por esta línea?
  // *this = x;
}
```

Herencia (1)

- La **herencia** es la capacidad de **definir clases nuevas** a partir de otras ya establecidas de forma que presenten las mismas características de aquellas, más otras nuevas.
- La creación de una especialización de una clase existente se denomina **subclase o clase derivada**, y la clase existente es la **clase base** de la nueva.
- Las clases derivadas **heredan los atributos y métodos de su clase base**, y además, pueden añadir nuevas características y métodos que son apropiados para objetos más especializados, o **redefinir** características heredadas.

Herencia (2)

- **Modificadores de acceso**: en C++ los elementos públicos y protegidos se pueden heredar de forma `public` (públicos y protegidos en la clase derivada) o `private` (se vuelven privados).
- Los **constructores no se heredan**, pero se llaman automáticamente antes del constructor de la clase derivada (también usando "`Derivada(...): Base(...)`").

```
class ClaseDerivada: [public, private] ClaseBase1,
                     [public, private] ClaseBase2, ...
{ ...
};
```

Herencia (3)

```
class A
{ ...
public:
  A(int, int = 1);
  ...
};

class B: public A
{ int i;
  float f;
public:
  B(int x, float y): A(x), f(y)
  { ...
  }
};
```

Herencia (4)

- **Sobrecarga y redefinición de métodos**: se llama **redefinición** de un método al proceso de volver a definir en una clase derivada un método que ya se ha definido en la clase base.
- El método sobrecargado de la clase derivada **oculta** al de la clase base, y **no existen simultáneamente** al mismo nivel de acceso. Sin embargo se puede acceder al método de la clase base usando el operador de ámbito (`ClaseBase::Método`).

Polimorfismo y vinculación dinámica (1)

- El **polimorfismo** representa la capacidad de los objetos de adoptar formas diferentes. El **enlace dinámico** es el mecanismo que permite el polimorfismo.
- La **sobrecarga de funciones** se resuelve en **tiempo de compilación**. El **polimorfismo** es una decisión en **tiempo de ejecución**.
- La llamada a una función usando enlace dinámico es **más lenta** que la de una que emplee enlace fijo.

```
class A
{
public:
  void f(void)
  { cout << "A::f()" << endl; }
  void g(int i)
  { cout << "A::g()" << endl; }
};
```

```
class B: public A
{
public:
  void f(void)
  { cout << "B::f()" << endl; }
  void g(char *c)
  { cout << "B::g()" << endl; }
};
```

```
int main()
{ B b;
  b.f(); // B::f()
  b.A::f(); // A::f()
  b.g("Hola"); // B::g()
  b.g(3); // ERROR
}
```

Polimorfismo y vinculación dinámica (2)

```
class A
{
public:
  void f(void)         { cout << "A::f() + "; g(); }
  virtual void g(void) { cout << "A::g()" << endl; }
};

class B: public A
{
public:
  virtual void g(void) { cout << "B::g()" << endl; }
};

int main()
{ A a;
  B b;
  a.f(); // A::f() + A::g()
  b.g(); // B::g()
  b.f(); // A::f() + B::g()
}
```

Métodos virtuales puros

- Un método definido como:

```
virtual void <nombre_método>(<params>) = 0;
```

  se denomina un **método virtual puro**, y la clase que lo contiene es una **clase virtual pura (o abstracta)**.
- **No** se pueden **crear objetos** de clases virtuales puras.
- Un clase dejará de ser virtual pura tan pronto como todos sus métodos virtuales puros sean redefinidos con métodos reales.

Bibliografía

- Bjarne Stroustrup (1994), *The C++ Programming Language*, Addison-Wesley.
- Syntax Highlighting with Google Docs' add-on: Code Pretty