Ampliación de Programación Avanzada - C/C++ -

J.J. Sánchez / J.A. Medina
Ciencias de la Computación
Universidad de Alcalá

COMENTARIOS

C++

- **C++** es un lenguaje de programación diseñado a mediados de los 1980 por **Bjarne Stroustrp**.
- Comenzó extendiendo al C con mecanismos propios de la **programación orientada a objetos**.
- Posteriormente añadió facilidades para la **programación genérica**.
- Existe un estándar denominado **ISO C++**.
- *C*++ es un lenguaje muy potente (permite trabajar tanto a alto como a bajo nivel).
- C++ permite **redefinir los operadores** (sobrecarga de operadores) y crear nuevos tipos que se comporten como tipos fundamentales.

Bibliografía

- Stroustrup, B. 1997. “The C++ programming language”. 3rd Ed.. Addison-Wesley.
- Eckel, B. 2003 “Thinking in C++” Volumenes 1&2. 2nd Ed. Prentice-Hall.
  (Disponibles en formato PDF en la página web http://mindview.net/Books/TICPP/ThinkingInCPP2e.html)

Software

- En el laboratorio se empleará el entorno de Bloodshed Software Dev-C++.
- Puede descargarse libremente en:
  http://www.bloodshed.net/devcpp.html

RUDIMENTOS

Un clásico: Hello world!

```
#include <iostream.h>
#include <stdlib.h>

int main(){
    std::cout << "Hello brave new world!
\n";
    return 0;
}
```

Tipos

- Booleanos (*bool*)
- Caracteres (p.e. *char*)
- Enteros (p.e. *int*)
- Coma flotante (p.e. *double*)
- Enumeraciones (*enum*)
- Punteros (p.e. *int**)
- Arrays (p.e. *char[]*)
- Clases

Booleanos (bool)

- Toma dos valores: `true` y `false`
- Conversiones (casting) frecuentes:
  - a enteros: **`true`** toma el valor de 1 y **`false`** el de 0.
  - desde enteros: 0 va a **`false`** y resto de valores a **`true`**.
  - desde punteros: un puntero nulo va a **`false`** y el resto a **`true`**.

Caracteres (*char*)

- Almacenan un carácter del juego de caracteres de la implementación (suele ser variante de ISO-646, p.e. ASCII).
- Normalmente tienen 8 bits de longitud (pueden almacenar 256 valores).
- Lo más seguro es asumir un juego típico de 127 caracteres.
- Pueden tener signo (*signed char*), o no (*unsigned char*); en las conversiones a enteros:
  - Con signo se interpretan -127 a 127
  - Sin signo de 0 a 255
- Otro tipo asociado (*wchar_t*) almacena caracteres de juegos mayores como Unicode (depende de las implementaciones).

Enteros (int, long)

- Pueden tener signo (`signed`) o no tenerlo (`unsigned`).
  - El tipo **`unsigned`** es sinónimo de **`unsigned int`**, y **`signed`** lo es de **`signed int`**.
- Dependiendo de su capacidad de almacenamiento, pueden ser cortos (`short`) o largos (`long`):
  - El tipo `long int` es sinónimo de `long`, y `short int` lo es de `short`.
- Los enteros sin signo se usan para almacenar **arrays de bits**.

Coma flotante (float, double)

- Admite tres tamaños:
  - `float`
  - `double` (defecto en los literales)
  - `long double`
- El tamaño exacto **depende de la implementación** concreta.

Literales

- Los caracteres literales entre comillas simples: `'a'`, `'b'`,...
- Las constantes enteras en decimal (0,45,…), hexadecimal (0x3f, 0x78,…) o en octal (012, 023,…).
- Lo números en coma flotante: 1.23, 1.2e10, 1.23e-15, 3.141592f (para forzar `float`, tb. con F),…

```
bool a = true;
bool b = false;
bool c = a+b; //true
char ch = 'a';
unsigned int num = 015;
float fc = 1.2e-40;
```

Void

- Sintácticamente es un tipo fundamental.
- **¡No hay objetos de dicho tipo!**
- Sus usos principales son:
  - indicar que una función no devuelve valor
  - servir de tipo base para punteros a objetos de tipo desconocido.

Enumeraciones (enum)

- Almacena un conjunto de valores especificado por el programador.
- El rango de una enumeración va desde 0 (si el más pequeño no es negativo) hasta la potencia de 2 más cercana (por la derecha).

```
enum flag { x=1, y=2, e=8};
flag f1 = flag(5);
```

Punteros

- **Un puntero almacena la dirección de otro objeto.**
- Pueden apuntar exclusivamente a objetos de un tipo concreto (p.e. `char*`) o de cualquier tipo (`void*`).
- El operador de referencia **`&` devuelve la dirección de un objeto**, mientras que el operador de derreferencia **`*` devuelve el objeto apuntado por un puntero**.

```
char c = 'a';
char* p = &c; //p la dirección de c
char c2 = *p; // c2=='a'
```

Arrays

- Un **array** es un conjunto de valores de un mismo tipo indexado mediante un natural (comienza en 0).
- Ejemplos de declaraciones típicas:

```
float f[3];
int n[2][3][4];
```

- Puede declararse el tamaño mediante los valores iniciales

```
char c[3] = { 'a', 'b', 'c' };
```

- También son válidas declaraciones del tipo:

```
int v[4]= { 1, 2, 3}; // el último es cero
char p[] = "Hello world!" ; // tiene 13 elementos!
```

Punteros a arrays

- El nombre de un array se puede usar como un puntero al primer elemento.

```
int v[] = {1, 2, 3, 4};
int* p1=v; //puntero al primer elemento
int* p2=&v[0]; //puntero al primer elemento
int* p3=&v[3]; //puntero al último elemento
```

- Los punteros se pueden usar para recorrer un arrays (aritmética de punteros)

```
void f(char v[]){
    for(char* p=v; *p!=0; p++)
        do_something(*p);
}
```

Constantes

- Las constantes se declaran mediante la palabra reservada ***const***.
- *Deben inicializarse al ser declaradas.*

Estructuras

- Una estructura es un agregado de distintos tipos como

```
struct name {
    char* firstname;
    char* lastname;
};
name jd = { “John”, “Smith”};
```

- Los elementos de le estructura se acceden empleando ‘.’, un punto :

```
name.lastname = "John2";
```

Consejos

- Evitar aritmética de punteros **no trivial**.
- No escribir más allá de los límites de un array.
- Emplear `0` antes que `NULL`.
- Usar cadenas (`Strings`) antes que arrays de caracteres terminados en 0.
- Evitar `*void` excepto en código de bajo nivel.
- Evitar literales no triviales.

I/O sencilla

- La entrada salida en C++ se realiza mediante flujos.
- Hay dos flujos predefinidos:
  - std::in para entrada teclado
  - std::out para salida consola
- La sintáxis para enviar a un flujo es << y para recibir de un flujo >>

```
char c;
std::cin >> c; //leemos
std::cout << c; //escribimos
```

Selección

- Las instrucciones encargados de la selección son los siguientes:
  - if (condition) statement
  - if (condition) statement else statement
  - switch (condition) statement

if (condition) statement else statement

```
if (a<=b)
    max = b;
else
    max = a;
```

switch (condition) statement

```
enum keyword {ASM, AUTO, BREAK};
void f(keyword key)
{
switch (key){
case ASM:
//hacer algo
break;
case BREAK:
//hacer otra cosa
break;
}
}
```

Iteración

- while (condition) statement
- do statement while (expression);
- for (init-statement ; condition; expression) statement

while (condition) statement

```
…
int i = 1;
while(i <= 20) {
cout << i;
i++;
}
…
```

for (init ; cond; expr) statement

- **Init** es la inicialización, **cond** es la condición de permanencia en el bucle y **expr** es la instrucción que da la iteración.

```
for ( i = 0; i < n; i++)
for ( j = 0; j < m; j++) if (nm [i] [j] == a)
    goto found;
// not found
// …
found:
//nm [i] [j] ==a
//…
```

do statement while (expression);

- **Atención: el cuerpo del bucle se ejecuta al menos una vez. Puede haber errores si se necesita alguna condición para su ejecución.**

Operadores aritméticos

Operator Name	Syntax	Operator Name	Syntax
Plus	**+**a	Assignment by Subtraction	a **-=** b
Addition (Sum)	a **+** b	Multiplication (Product)	a * b
Prefix increment	**++**a	Assignment by Multiplication	a ***=** b
Postfix increment	a**++**	Division (Quotient)	a **/** b
Assignment by Addition	a **+=** b	Assignment by Division	a **/=** b
Unary Minus (Negation)	**-**a	Modulus (Remainder)	a **%** b
Subtraction (Difference)	a **-** b	Assignment by Modulus	a **%=** b
Postfix decrement	**--**a		

Operadores comparación/ lógicos

Operator Name	Syntax	Operator Name	Syntax
Less Than	a **<** b	Not Equal To	a **!=** b
Less Than or Equal To	a **<=** b	Equal To	a **==** b
Greater Than	a **>** b	Logical Negation	**!**a
Greater Than or Equal To	a **>=** b	Logical AND	a **&&** b
Logical OR	a **||** b		

Operadores de ***bits***

Operator Name	Syntax	Operator Name	Syntax
Bitwise Left Shift	a **<<** b	Assignment by Bitwise AND	a **&=** b
Assignment by Bitwise Left Shift	a **<<=** b	Bitwise OR	a **|** b
Bitwise Right Shift	a **>>** b	Assignment by Bitwise OR	a **|=** b
Assignment by Bitwise Right Shift	a **>>=** b	Bitwise XOR	a **^** b
Bitwise One's Complement	**~**a	Assignment by Bitwise XOR	a **^=** b
Bitwise AND	a **&** b		

Consejos

- Usar la librería estándar antes que otras…
- Evitar expresiones complicadas.
- Usar paréntesis en caso de duda (precedencia).
- Evitar conversiones explícitas.
- Evitar **goto**.
- Evitar **do-statement**.
- No declarar las variables hasta que se tiene un valor para ello.

Ejemplo: Test primalidad

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>

int main(){
int num, det=1, div;
cout << "Introduzca un numero:
";
cin >> num;
div = num - 1;
if (num > 3){
while(div > 1){
det = det * (num % div);
div--;
}
if (det == 0)
cout << "El numero no es
primo." << endl;
else
cout << "El numero es primo."
<< endl;
}
else
cout << "El numero es primo."
<< endl;
system("PAUSE");
return 0;
}
```

Ejemplo: Burbuja

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#define T 10

void ordenar(int[]);
void main() {
int vector[T];
for (int i=0; i < T; i++) {
cout << "Introduzca el numero "
<< (i+1) << ":\t";
cin>> vector[i];
}
ordenar(vector);
cout<< "Los elementos ordenados
son: ";
for (int m=0; m < T; m++)
cout<< vector[m] << " ";
cout << endl;
system("PAUSE");
return 0;
}
void ordenar(int vect[]) {
for (int i=0; i < T; i++) {
for (int k=0; k < T-1; k++) {
if(vect[k] > vect[k+1]) {
int a = vect[k];
vect[k] = vect[k+1];
vect[k+1] = a;
}
}
}
}
```

Ejemplo: Números aleatorios

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <cstdlib>

using namespace std;
int main()
{
srand((unsigned)time(0));
int random_integer = rand();
cout << random_integer << endl;
system("PAUSE");
return 0;
}
```

Funciones

- La declaración de una función proporciona:
  - El nombre de la función
  - El tipo del valor de retorno (si lo hay)
  - El número y el tipo de los argumentos
- La semántica del paso de argumentos es igual al de la inicialización.
- En la llamadas se comprueban los tipos de los argumentos y se llevan a cabo conversiones implícitas si se necesita.

Funciones (y2)

- Toda función debe declararse en algún lugar.
- Una definición es una declaración que incluye el cuerpo de la función.
- En las declaraciones no es necesario indicar el nombre de los argumentos.
    - En las definiciones tampoco si no se usan (previsión o cuestiones de compatibilidad).
- Pueden declararse *inline* para que el compilador genere código y no una llamada a la función.
- Las variables *static* son compartidas por todas las llamadas de una función.

Ejemplos

```
extern void swap(int*, int*, char*) //declaración
void swap(int* p, int* q, char* ) //definición
{
int t = *p;
*p = *q;
*q = t;
}
inline int sum(int a, int b)
{
return a+b;
}
```

Paso de argumentos

- La semántica del paso de argumentos es la misma que la de la inicialización.
- El tipo de los argumentos se comprueba y, si es necesario, se llevan a cabo las conversiones.
- Los argumentos pueden pasarse por valor y por referencia.
- **Si se pasan por referencia conviene declararlos como constantes (modificador const) para evitar que se modifiquen (y evitar problemas). Idem si se pasan punteros.**
- Los arrays sólo pueden pasarse por referencia (se interpretan como punteros al tipo del array).
- Pueden definirse argumentos por defecto (entre los finales únicamente).

Ejemplos

```
void f(int value, int& reference)
{
value++; //incrementa copia local
reference++; //incrementa variable original
}
void g(const int& parameter) //el parámetro no puede …
//cambiarse
void h(const char* array) //pasamos una array
… //que no puede //modificarse
void i(int a, int b=10) //se puede llamar con i(8) para
//que b tome el valor por defecto
```

Valores

- Una función devuelve un valor si y sólo si no ha sido declarada como *void*.
- No deben devolverse referencias a variables locales (!su espacio se reutiliza al salir de la función!).

```
void f() {…return i;… } // ¡Mal!
int& g() {…return local;…} // ¡Mal!
int* h() {…return &local;…} // ¡Fatal!
```

Sobrecarga de funciones

- Varias funciones pueden compartir el mismo nombre.
- El compilador decide cuál es la más apropiada en función de los tipos de las expresiones y los de los argumentos:
  - Coincidencia exacta o con conversiones triviales (p.e. nombre de un array a puntero)
  - Coincidencia usando promociones (p.e. bool a int, char a int, float a double,…)
  - Coincidencia usando conversiones estandar (p.e. int a double, int* a void*,…)
  - Coincidencia basadas en conversiones definidas por el usuario.
  - Coincidencias gracias al uso de la elipsis.
- Los valores de retorno no se tienen en cuenta en las sobrecargas.

Algunas cosas más

- Se pueden declarar funciones con un número indeterminado de argumentos.
- Se pueden declarar punteros a funciones y utilizar estos para ejecutar las funciones en cuestión.
- Se pueden usar macros, sobre todo para compilaciones condicionales.

Organización del código

- **.h:** ficheros a incluir con la directiva #include, deben contener las declaraciones, las definiciones de los tipos de datos, de las clases,…
  - **#include <stdio.h>** para ficheros de la librería general…
  - **#include "my.h"** para ficheros del directorio actual.
- **.c/.cpp**: ficheros con el código, definiciones de las funciones,…

Algunos consejos

- Sospechar de argumentos pasados por referencia que no son constantes.
- Evitar las macros.
- Evitar funciones con un número indeterminado de argumentos.
- No devolver **jamas** referencias a variables locales.

Espacios de nombres (namespace)

- Es un mecanismo para expresar unión lógica entre varias entidades.
- Conviene separar la declaración del *namespace* (el interfaz) de la definición.
- No se pueden definir nuevos miembros del *namespace* fuera del mismo.
- Se pueden definir alias para los *namespace*.
- Se pueden combinar varios *namespace* en uno.
- Se puede acceder sólo a parte de un *namespace.*

Ejemplos

```
namespace Sample {
double f(float); //declaramos f
}
namespace Sample = Sample_Namespace;
double Sample::f(float f) //definimos f
{
Using Sample2::g; //vamos a usar g del Sample2
…
g();
…
}
```

Ejemplos (y2)

```
namespace Sample1 {
…
}
namespace Sample2 {
…
}
namespace Sample3 {
double f(); //una función
…
}
```

```
namespace MyFinalSample {
using namespace Sample1
using namespace Sample2
using Sample3::f
…
}
```

Gestión de errores

- La gestión de errores se puede llevar a cabo vía excepciones.
- La idea es que la función que puede detectar el error no sabe tratarlo y lanza una excepción para que su director lo trate.
- La construcción es la estándar:

```
try {…} catch {…}
```

Consejos

- Usar namespace para dar estructura lógica.
- Colocar todo nombre no local en un namespace.
- Utilizar alias para acortar nombres largos.
- Utilizar excepciones para el tratamiento de errores.

CLASES

Clases

- Son tipos definidos por el usuario.
- Tiene propiedades (atributos) y métodos (acciones que pueden llevar a cabo).
- Tienen constructores que permiten inicializar los objetos de dicha clase (sus propiedades, …).
- Pueden tener propiedades y/o métodos *static* compartidos por todos los objetos de la clase.

Privacidad

- En una clase pueden declararse públicos tanto métodos como propiedades.
- Basta usar el modificador **“public:”** antes de lo que deseemos hacer público (da la interfaz del objeto).
- Lo que se encuentra antes del modificador es privado (sólo puede usarse por funciones miembro).

Constructores

- Las clases poseen un método especial llamado **constructor** que inicializa el **objeto**.
- Los constructores tienen exactamente el mismo nombre que la clase.
- Suelen facilitarse varios constructores para adaptarse a distintas circunstancias.
- Puede ser interesante emplear valores por defecto en los constructores (en vez de definir varios constructores).

Componentes estáticos

- Las clases pueden tener métodos estáticos, añadiendo el modificador **static** a la definición.
- Los métodos estáticos pertenecen más a la clase que al objeto: no hace falta hacer referencia al objeto sino a la clase al ejecutarlos.
- Ocurre los mismo con las propiedades, puede haber propiedades estáticas.

Funciones constantes

- Al igual que pueden pasarse argumentos constantes, las funciones miembro pueden declararse como constantes (modificador **const** depués de los argumentos).
- Así, se declarara que una función no modificará el valor del objeto.
- Algunas propiedades pueden cambiarse aún desde funciones constantes si se declaran mutables con el modificador **mutable**.

Ejemplo

```
class Date{
int year;
int month;
int day;
public:
void print() const;
void Date();
}
void Date::print() {…}
```

Autoreferencia

- En funciones miembro que, lógicamente no deben devolver ningún valor, se puede devolver una referencia al objeto para poder encadenar acciones sobre el mismo.
- Las funciones no estáticas permiten referenciar al objeto con el que se invoco la función utilizando **this**.

Clases vs. estructuras

- Las estructuras (**struc**) son clases en las que todos los miembros son públicos por defecto.
- Para hacerlos privados, basta añadir "**private:"** antes de los miembros que se desee sean privados.

Ejercicios

- Implementar una clase Date par almacenar fechas con las siguientes características:
  - Un constructor que tome 3 parámetros enteros (day, month y year), que se pueda llamar con 0, 1, 2 ó 3 argumentos. Los valores ausentes se deben inicializar con los valores de una fecha por defecto. Dicha fecha por defecto debe ser común a toda los objetos y no debe poderse modificar.
- Funciones para actualizar y obtener los distintos elementos de la fecha (el día, el mes o el año).
- Una función para decidir si una fecha es más reciente que otra.

Ejercicios

- Implementar una clase Person par almacenar los datos de una persona con las siguientes características:
  - Un constructor que tome 3 parámetros (FirstName, LastName y Birthday), que se pueda llamar con 2 ó 3 argumentos (si no se especifica la fecha se asignará una por defecto). Dicha fecha por defecto debe ser común a toda los objetos y no debe poderse modificar.
- Funciones para actualizar y obtener los distintos valores.
- Una función para decidir si una persona es más joven que otra.

Bibliografía

- Meyer, B. 1997. "Object-oriented software construction". 2nd Ed.. Prentice-Hall.

- **"Epoch-making"** *Journal of O-O Programing*
- **"Destined to become the comprehensive and definitive reference"** *Software Development*
- **"The ultimate O-O guide"** *Unix Review*
- **"Arguably the best work on the subject"** John Dvorak in *PC Week*
- **"Read this book and you'll immediately be a better programmer"** David Wall at Amazon.com
- **"If you program computers, you need to read this book"** Official Amazon.com review

OBJETOS

Creación de objetos

Hay varias formas de crear objetos:

- Como variables locales,
- Como variables globales,
- Como miembros de otras clases,
- …

Creación de objetos

Hay varias formas de crear objetos:

- Como variables locales,
- Como variables globales,
- Como miembros de otras clases,
- Dinámicamente
- Temporalmente al evaluar una expresión
- …

Constructores y destructores

- Los constructores inicializan un objeto, su nombre coincide con el de la clase.
- Los destructores destruyen un objeto, liberando la memoria y llevando a cabo las tareas necesarias (cerrar ficheros,…). Su nombre es el de la clase precedido de ~.
- Las clases tiene constructores por defecto.

Constructores y destructores (y2)

- Es necesario un constructor en:
  - Clases sin constructor específico,
  - Clases con constantes,
  - Clases con referencias.
- Los objetos locales se inicializan cuando el hilo de ejecución alcanza su declaración y se destruyen al salir de la función, el bloque,…

Operadores *new* y *delete*

- ***new*** es un operador que se encarga de crear un objeto dinámico, devuelve una referencia al mismo.
- ***delete*** se encarga de destruir un objeto liberando la memoria (dinámica) por el ocupada.
- Tb. existen ***new[]*** y ***delete[]*** (para arrays).

Sobrecarga de operadores

- En C++ los operadores pueden sobrecargarse.
  - Cambiar su significado en función de los tipos de los operandos
- En las sobrecargas, uno de los operandos debe tener un tipo definido por el usuario (al menos).
  - No se puede redefinir la suma de int, p.e..

Operadores sobrecargables

```
+ * / % ^ & | ~ ! < > += = *= /=
%= ^= &= |= << >> >>= <<= == !=
     <= >= && || ++ >* , >
   = [ ] ( ) new new[] delete
            delete[]
```

Ejemplo

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

class Integer {
int i;
public:
Integer(int ii) : i(ii) {}
const Integer
operator+(const Integer& rv) const {
cout << "operator+" << endl;
return Integer(i + rv.i);
}
Integer&
operator+=(const Integer& rv) {
cout << "operator+=" << endl;
i += rv.i;
return *this;
}
};

int main() {
cout << "built-in types:" << endl;
int i = 1, j = 2, k = 3;
k += i + j;
cout << "user-defined types:" << endl;
Integer ii(1), jj(2), kk(3);
kk += ii + jj;
} ///:~
```

Operadores particulares

- Los operadores del tipo:

  `X :: operator T (),`

definen conversiones de tipos.

- El operador [] permite construir arrays asociativos,…

Clases derivadas

- Partiendo de una clase se pueden definir clases derivadas que *heredan* de la primera.
  - La clase de la que heredan suele llamarse superclase (base) y la que hereda subclase.
  - La clase derivada puede usar los miembros públicos (***public:***) y los protegidos ***(protected:)***, pero no los privados ***(private:)***.

Ejemplo

```
class Employee {
string first_name,family_name;
public:
void print()const;
/ / ...
};
class Manager : public Employee{
/ /
public:
Voidprint() const;
/ / ...
};
```

Constructores y destructores

- Si una clase base tiene un constructor, este debe llamarse desde las clases derivadas.
- Se puede llamar implícitamente, pero si se necesitan argumentos, hay que hacer una llamada explícita.

Ejemplo

```
Employee::Employee(const string& n , int d ) :
familily_name(n), department(d)
{
//
{
// ...
}
Manager::Manager(const string& n , int d, int lvl)
:Employee(n ,d ), // inicializa base
level(lvl) // inicializa level con lvl
{
// ...
}
```

Jerarquía de clases

- La relación de derivación da estructura de grafo a la jerarquía de clases.
- En C++ hay herencia múltiple:

```
class Tsec:public Temporary, public Secretary {...};
class Consultant:public Temporary, public
Manager{ ...};
```

Funciones virtuales

- Permiten reescribir funciones en clases derivadas.
- Deben definirse en alguna clase derivada.
- Se indican mediante el modificador ***virtual.***

```
virtual void print () const;
```

- Hay funciones puramente virtuales (dan interfaces):

```
virtual void rotate (int) = 0;
```

Sirven para definir clases abstractas.

PLANTILLAS

Ejemplo

```
#include "fibonacci.h"
#include "../require.h"
#include <iostream>

using namespace std;

class IntStack {
        enum { ssize = 100 };
        int stack[ssize];
        int top;
    public:
        IntStack() : top(0) {}
        void push(int i) {
            require(top < ssize, "Too many push()es");
            stack[top++] = i;
        }
    int pop() {
        require(top > 0, "Too many pop()s");
        return stack[--top];
    }
};
```

```
int main() {
    IntStack is; for(int i = 0; i < 20; i++)
    is.push(fibonacci(i));
    for(int k = 0; k < 20; k++)
    cout << is.pop() << endl;
}
```

Ejemplo(y2)

```
#include "../require.h"
#include <iostream>
using namespace std;

template<class T> class Array {
        enum { size = 100 };
        T A[size];
    public:
        T& operator[](int index) {
            require(index >= 0 && index < size,
                "Index out of range");
        return A[index];
        }
};
```

```
int main() {
    Array<int> ia;
    Array<float> fa;
    for(int i = 0; i < 20; i++) {
        ia[i] = i * i;
        fa[i] = float(i) * 1.414;
    }
    for(int j = 0; j < 20; j++)
        cout << j << ": " << ia[j]
            << ", " << fa[j] << endl;
}
```

Ejemplo(y3)

```
template<class T>
class Array {
enum { size = 100 };
T A[size];
public:
T& operator[](int index);
};

template<class T>
T& Array<T>::operator[](int index) {
require(index >= 0 && index < size,
"Index out of range");
return A[index];
}

int main() {
Array<float> fa;
fa[0] = 1.414;
}
```

Plantillas

- Soportan la programación genérica en C++.
- Permiten pasar parámetros como tipos.
- Se usan ampliamente en la librería estándar (STL).

Ejemplo

```
template<class C> class String{
        struct Srep;
        Srep *rep;
    public:
        string();
        string(const C*);
        C read(int i) const;
…
}
…
template<class C> String<C>::read(int i)…
```

Algunos comentarios generales

- Una clase generada a partir de una plantilla es una clase común.
- El depurado de plantillas es complicado…suele costar la abstracción.
- Sólo se pueden sobrecargar las funciones.
- La especialización permite proporcionar implementaciones alternativas.
- El tipo proporcionado como argumento debe facilitar la interfaz asumida por la plantilla.

Algunos comentarios generales (y2)

- El proceso de generar una declaración de clase a partir de la plantilla se llama **instanciación de plantilla**.
  - Para cada tipo de argumentos se debe instanciar una versión de la plantilla.
- Las plantillas pueden tomar varios parámetros:

```
template<class T, T defval> class Cont {};
```

- Dos instanciaciones con los mismos argumentos son del mismo tipo.

Algunos comentarios generales (y3)

- Las plantillas pueden tener argumentos por defecto:

```
template<class T, class C=Cmp<T>>…
```

- Se pueden derivar clases de plantillas, plantillas de clase,…
- Se pueden definir plantillas para funciones.

```
template<class T> void sort(vector<T>&);
…
template<class T> void sort(vector<T>& v)
{…}
```

- Las plantillas de función se pueden sobrecargar.

Resolución de sobrecarga

- Encontrar el conjunto de plantillas de funciones que tienen parte en la resolución de la sobrecarga.
- Si se pueden aplicar dos plantillas de función, aplicar la más especializada.
- Llevar a cabo la resolución para este conjunto de funciones así como de cualquier otra función común.
- Si una función y una especialización se ajustan, se prefiere la función.
- Si no se encuentra una función o se encuentra más de una se produce un error (llamada ambigua).

Especializaciones (Clases)

- Las plantillas se pueden especializar para optimizar su funcionamiento:

```
template<> class Vector<void*>{…}
```

Suponemos dada la plantilla Vector y especializamos para Vectores de punteros.

- Se pueden especializar parcialmente (distinguiendo entre si son punteros o no sus argumentos):

```
Template<class T> class Vector<T*>:private
                  Vector<void*>
```

- Al resolver se prefiere a las versiones más especializadas.

Especializaciones (Funciones)

- Las plantillas de funciones se pueden especializar de igual manera.
- Se pueden proporcionar especializaciones parciales de las funciones (especializando parcialmente los argumentos).

Organización del código

- Definir la plantilla en un .h e incluir este haya donde se necesite.
- Declarar la plantilla en un .h, definirla en un .cpp (**extern**) e incluir el .h haya donde se neceite.

Ejemplo

```
#ifndef STACKTEMPLATE_H
#define STACKTEMPLATE_H
#include "../require.h"

template<class T> class StackTemplate {
        enum { ssize = 100 };
        T stack[ssize];
        int top;
    public:
        StackTemplate() : top(0) {}
        void push(const T& i) {
            require(top < ssize, "Too many push()es");
            stack[top++] = i;
        }
        T pop() {
            require(top > 0, "Too many pop()s");
            return stack[--top];
        }
        int size() { return top; }
};
#endif
```

Consejos

- Usar plantillas para expresar algoritmos que se aplican a numerosos argumentos.
- Usar plantillas para definir contenedores, especializando a punteros.
- Declarar la forma general de la plantilla antes que sus especializaciones.
- Declarar especializaciones antes de usarlas.
- Utilizar especialización y sobrecarga para dar un único interfaz.
- Utilizar interfaces simples para casos simples y tratar los complejos con sobrecarga.

Consejos (y2)

- Depurar en casos concretos antes de generalizar a la plantilla.
- Hay que exportar las definiciones de las plantillas **(export).**

Tratamiento de errores

- El autor de una librería conoce cuando se ha producido el error.
- El usuario de la librería sabe cómo recuperarse del mismo.
- Alternativas tradicionales:
  - Terminar el programa.
  - Devolver un valor representando el error.
  - Devolver un valor legal y dejar el programa en un estado ilegal.
  - Llamar a una función suministrada por el usuario.

Ninguna es especialmente recomendable.

Manejo de excepciones

- Actualmente se utiliza el manejo de excepciones:
  - La librería detecta un error y *lanza* un excepción con información sobre el error.
  - El usuario *captura* el error y lo trata.
  - La librería indica de alguna manera que excepciones va a lanzar.
  - El usuario indica que excepciones va a tratar.

Excepciones

- En C++ las excepciones se representan mediante Clases.
- Hay una jerarquía de excepciones (forman un grafo y se puede usar herencia múltiple).
- Cuando se lanza una excepción se recorre la pila de llamadas hasta encontrar una función que la trate.

Captura de excepciones

```
void f()
{
try{
throw E();
}
catch (H){
// ¿qué hacemos?
}
}
```

El manejador se llama…

- Si H es del mismo tipo que E.
- Si H es, sin ambigüedades, una clase base de E.
- Si H y E son de tipo punteros y lo anterior es válido para sus tipos base.
- Si H es una referencia y alguno de los dos primeros casos se tiene para el tipo al que refiere H.

Ejemplo

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

class Rainbow {
public:
  Rainbow() { cout << "Rainbow()" << endl; }
  ~Rainbow() { cout << "~Rainbow()" << endl; }
};

void oz() {
  Rainbow rb;
  for(int i = 0; i < 3; i++)
    cout << "there's no place like home" << endl;
  throw 47;
}

int main() {
  try {
    cout << "tornado, witch, munchkins..." << endl;
    oz();
  } catch(int) {
    cout << "Auntie Em! I had the strangest dream..."
         << endl;
  }
} ///:~
```

Captura de excepciones (y2)

- El manejador puede *relanzar* la excepción incluyendo **`throw`** de nuevo entre su código.

`...throw;...`

- Se pueden capturar todas las excepciones escribiendo **`catch(...)`** .
- Se pueden incluir varios manejadores (el orden es significativo):

```
try {...} catch (e1) {...} catch (e2) {...}...;
```

Especificación de excepciones

- Se pueden indicar las excepciones que va a lanzar una función:

```
void f(int) throw (e2,e6);
```

- Si una función lanza una excepción que no ha indicado se llama a `std::unexpected()` (que suele llamar a `std::terminate()` y esta a `abort()`).
- Se puede modificar para que se ejecute un código particular.

Jerarquía de clases

Nos vamos a centrar en:

- Herencia múltiple
- Control de acceso
- Identificación en tiempo de ejecución
- Punteros a miembros
- Empleo de la memoria dinámica

Herencia múltiple

- Ya hemos visto que una clase puede heredar de varias.
  - Cuando un objeto se utiliza como una superclase, están disponibles los métodos de dicha superclase (no los del resto de superclases).
  - Si hay conflicto con el nombre de un método (existe en dos superclases), lo mejor es definir dicho método en la subclase.
  - Uno puede incluir la directiva **using** en la **declaración** de la clase para indicar que funciones desea de las superclases. **Using** no debe aparecer en la **definición** de la clase.

Ejemplo

```
class A {
public:
int f(int);
char f(char);
//…
};
class B {
public:
double f(double);
//…
};
Class AB: public A, public B {
Public:
using A::f; using B::f;
char f(char); // ocultamos
A::f(char)
AB f(AB);
};
```

```
void g(AB &ab)
{
ab.f(1); // A::f(int)
ab.f(´a´); // AB::f(char)
ab.f(2.0); // B::f(double)
ab.f(ab); // AB::f(AB)
}
```

Herencia múltiple (y2)

- Una clase **A** puede heredar de dos superclases **B** y **C** que heredan de **D**.
  - En **A** hay dos copias de **D** (la heredada de **B** y la heredada de **C**).
  - Para hacer referencia a la parte **D** heredada de **C** se usa **C::D…** y para hacer referencia a la heredada de **B** se usa **B::D…**
  - Las funciones virtuales en una subclase como **A** sobrescriben a las de **B** y **C.**
  - Se puede evitar este duplicidad empleando clases base virtuales.

Ejemplo

```
class D {
//…
};
class B: public virtual D {
//…
};
class C: public virtual D {
//…
};
class A: public B, public C {
//¡Sólo una copia de D!
};
```

Control de acceso

Hemos visto que el acceso a los elementos de la clase puede ser:

- Privado: sólo realizable por los elementos de la clase:
- Protegido: realizable por los elementos de la clase y sus derivadas.
- Público: realizable por todo el mundo.

Control de acceso (y2)

Supongamos que **D** deriva de **B**:

- Si **B** es una base privada, sus miembros públicos y protegidos son accesibles por las funciones miembro y *amigas* de **D**. Sólo las *amigas* y los miembros de **D** pueden convertir una **D*** en una **B***.
- Si **B** es una base protegida, sus miembros públicos y protegidos son accesibles por las funciones miembro y *amigas* de **D** y de clases derivadas de **D**. Sólo las *amigas* y los miembros de **D** y de sus clases derivadas pueden convertir una **D*** en una **B***.
- Si B es una base pública, todos sus miembros públicos pueden usarse por cualquier función, sus miembros protegidos pueden accederse por las funciones *amigas* y los miembros de **D** y de sus clases derivadas.

Información en tiempo de ejecución

- Cuando se usa polimorfismo, un objeto pierde su tipo en favor de uno de una clase base.
- Para recuperar esta información se usa
  - `dynamic_cast<A*>(p)`
- (¿es el objeto apuntado por p del tipo A?).
- `Dynamic_cast` devuelve un puntero válido si es así y 0 en caso contrario.

Información en tiempo de ejecución (y2)

- Para recuperar información de tipo de referencias se usa
  - `dynamic_cast<A&>(r)`
- (el objeto referenciado por r es del tipo A).
- `Dynamic_cast` lanza una excepción `bad_cast` si no es así.

Información en tiempo de ejecución (y3)

- Para obtener información sobre el tipo de un objeto sin asumir pertenencia a una clase se usa `typeid()`.
- `Typeid()` devuelve:
  - La información del tipo de la expresión que se ha pasado como argumento, o
  - Si se pasa un nombre, la informaciín del tipo correspondiente a ese nombre.
- Devuelve objetos de tipo `type_info`.
- No está garantizado que haya un único type_info para un tipo concreto (problemas con librerías dinámicas,...), luego para las comprobaciones hay que usar == y no =.

UN VISTAZO A C++ STL

Funcionalidad de la STL

- Proporciona soporte para características del lenguaje: gestión de memoria, información en tiempo de ejecución,...
- Proporciona información sobre características dependientes de la implementación: tamaño máximo de los float,...
- Proporciona funciones que no pueden implementarse de manera óptima en el propio lenguaje: sqrt(), memmove(),...
- Proporciona facilidades no-primitivas para la programación en el lenguaje: vectores, listas...
- Proporciona soporte para características de I/O,...
- Proporciona un punto de partida para otras librerías.

Principios de diseño

- Debe resultar esencial para todo estudiante, programador, ...
- Debe ser usada directa o indirectamente por todo programador.
- Debe ser eficiente para que nadie quiera utilizar alternativas.
- No debe implementar políticas concretas: uno puede ordenar colecciones de objetos de diversas maneras, la librería debe permitir cualquiera de ellas,...
- Debe ser matemáticamente consistente y primitiva.
- Debe ser completa en sus tareas.
- Debe ser eficiente (para que nadie proporcione alternativas ad-hoc) y razonablemente segura.
- Debe soportar los estilos de programación más frecuentes.
- Extensible para soportar tipos del usuario de manera similar a los tipos primitivos.

Organización

Contenedores:

- <vector>: vectores unidimensionales de tipo T
- <list>: listas doblemente enlazadas de tipo T
- <deque>: dobles colas de tipo T
- <queue>: colas de tipo T, incluye priority_queue
- <stack>: conjuntos de tipo T
- <map>: vectores asociativos de tipo T, incluye multimap
- <set>: conjuntos de tipo T, incluye multiset
- <bitset>: vectores de booleanos

Organización (y2)

Utilidades generales

- <utility>: operadores y pares
- <functional>: objetos función
- <memory>: gestión de memoria para contenedores
- <ctime>: fechas y tiempo estilo C.

Iteradores:

- <iterator>: iteradores

Algoritmos:

- <algorithm>: algoritmos generales
- <cstdlib>: bsearch(), qsort()...

Oraginización (y3)

Diagnóstico:

- <exception>: clases relacionadas con las excepciones
- <stdexcept>: excepciones estándar
- <cassert>: macro assert
- <cerno>: manejo de errores tipo C

Cadenas:

- <string>: cadenas de tipo T
- <cctype> clasificación de carcateres
- <cwtype>: clasificación de caracteres (ampliados)
- <cstring>: funciones de cadenas tipo C
- <cwchar>: funciones de cadena para caracteres tipo C
- <cstdlib>: funciones de cadenas tipo C

Organización (y4)

Entrada/Salida

- **<iosfwd>: declaraciones de facilidades de I/O**
- **<iostream>: objetos y operadores**
- **<ios>: base de iostream**
- **<streambuf>: buffers de flujos**
- **<istream>: flujos de entrada (plantilla)**
- **<ostream>: flujos de salida**
- **<iomanip>: manipuladores**
- **<sstream>: flujos de cadenas**
- **<cstdlib>: funciones de clasificación de caracteres**
- **<cstdio>: familia de funciones printf(),...**
- **<cwchar>: familia de funciones printf(),...para caracteres ampliados.**

Organización (y5)

Localización

- <locale>: representación de diferencias culturales
- <clocale>: representación de diferencias culturales al estilo de C.

Números:

- <complex>: operaciones y números complejos
- <valarray>: vectores numércios y operaciones
- <numeric>: operaciones numéricas generalizadas
- <cmath>: librería de funciones matemáticas estándar
- <cstdlib>: números aleatorios al estilo de C

Organización (y6)

Soporte del lenguaje

- <limits>: límites numéricos
- <climits>: límites numéricos al estilo de C
- <cfloat>: límites numéricos de los float estilo C
- <new>: gestión de la memoria dinámica
- <typeinfo>: información en tiempo de ejecución
- <exception>: excepciones
- <cstddef>: soporte de la librería de C
- <cstdarg>: paso de parámetros
- <csetjmp>: relacionada con la pila de llamadas
- <cstdlib>: terminación del programa
- <ctime>: reloj del sistema
- <csignal>: manejo de señales al estilo de C

Diseño de los contenedores

- Deben ser suficientemente generales (¡plantillas!).
- Deben proporcionar iteradores (objetos que recorren contenedores de manera secuencial).
- Deben implementar interfaces razonables (no debe haberinterfaces demasiado pesados).
- Deben ser simples y eficientes.
- Proporcionar implementaciones ad-hoc cuando la eficiencia lo requiera (vectores de bits,...).

CONTENEDORES ESTÁNDAR

Opreciones sobre iteradores

- En general, un contenedor puede verse como una secuencia de elementos recorridos por el orden dado por su iterador (o el inverso).
- Tienen las siguientes operaciones:
  - **`begin()`**: apunta al primer elemento.
  - **`end()`**: apunta al último elemento,
  - **`rbegin()`**: apunta al primer elemento en orden inverso
  - **`rend()`**: apunta al último elemento en orden inverso.
- También proporcionan acceso directo a algunos elementos:
  - **`front()`**: primer elemento
  - **`back()`**: último elemento
  - **`[ ]`**: acceso indexado **sin comprobación** (no para listas)
  - **`at()`**: acceso indexado **con comprobación** (no para listas)

Operaciones de pilas y colas

Algunas operaciones de manipulación de extremos en secuencias:

- **`push_back()`**: añade al final,
- **`pop_back()`**: borra el último elemento,
- **`push_front()`**: añade al comienzo (para listas y colas doblemente enlazadas)
- **`pop_front()`**: borra del comienzo (para listas y colas doblemente enlazadas)

Ejemplo

```
#include <cstdlib>
#include <iostream>
#include <iomanip>
#include <list>

using namespace std;

int main(int argc, char *argv[])
{
list<double> lista;
double num, suma=0;

do
{
cout << "Número? (0 salir):";
cin >> num;
if (num != 0)        lista.push_back(num);
}
while (num != 0);

cout << "----------" << endl;

while( !lista.empty() )
{
num = lista.front();
cout << num << endl;
suma += num;
lista.pop_front();
}
cout << "----------" << endl;

cout << suma << endl;

system("PAUSE");
return EXIT_SUCCESS;
}
```

Operaciones de listas

Las siguientes operaciones de listas se encuentran implementadas:

- **`insert(p,x)`**: añade x antes de p,
- **`insert(p,n,x)`**: añade n copias de x antes de p,
- **`insert(p, first, last)`**: añade desde first hast last (sin incluir este último) antes de p,
- **`erase(p)`**: borra p,
- **`erase(first, last)`**: borra desde first hast last (sin incluir este último),
- **`clear()`**: borra todos los elementos.

Ejemplo

```
#include <cstdlib>
#include <iostream>
#include <list>

using namespace std;

// Ejemplo: buscar e insertar
void insertar()
{
list <string> nombres;

nombres.push_back("Juan");
nombres.push_back("Jose");
nombres.push_back("Pedro");
nombres.push_back("Pablo");

// Se obtiene un iterador list<string>::iterator
it = nombres.begin();
// Buscamos el elemento "Pedro"
while(*it!="Pedro"&&it!=nombres.end())        it++;

// Insertamos un elemento "Maria"  nombres.insert(it, 1,
"Maria");

it = nombres.begin();
while( it != nombres.end() )
{
cout << *it++ << endl;
}
}

int main(int argc, char *argv[])
{
insertar();
system("PAUSE");
return EXIT_SUCCESS;
}
```

Otras operaciones

- **`size()`**: número de elementos,
- **`empty()`**: ¿está vacío?,
- **`max_size()`**: tamaño del mayor contenedor posible,
- **`capacity()`**: espacio reservado por vector (sólo vectores),
- **`reserve()`**: reserva espacio para expansiones posteriores (sólo vectores),
- **`resize()`**: cambia el tamaño del contenedor (sólo para vectores, listas y colas doblemente enlazadas)
- **`swap()`**: intercambis los elementos de dos contenedores,
- **`==`** : ¿es el contenido de los dos contenedores el mismo?
- **`!=`**: ¿es el contenido de los dos contenedores el mismo?
- **`<`** : ¿es un contenedor lexicográficamente menor que otro?

(Cons/des)tructores

- **`container()`**: contenedor vacío
- **`container(n)`**: contenedor con n elementos por defecto (no para asociativos)
- **`container(n,x)`**: contenedor con n copias de x.
- container(first, last): contenedor con elementos de first a last (no incluido),
- **`container(x)`**: constructor de copia, los elementos iniciales se toman de x,
- **`~container()`**: destruye el contenedor y sus elementos.

Asignaciones

- **`operator=(x)`**: asignación de copia (del contenedor x),
- **`assign(n,x)`**: asigna n copias de x (no para asociativos),
- **`assign(first, last)`**: asigna desde first hasta last (no incluido).

Operaciones asociativas

- **`operator[](k)`**: accede al elemento con la llave k (para asociativos con llave única)
- **`find(k)`**: encuentra un elemento con la llave k,
- **`lower_bound(k)`**: encuentra el primer elemento cuya llave es k,
- **`upper_bound(k)`**: encuentra el primer elemento con la llave mayor que k,
- **`equal_range(k)`**: encuentra lower_bound y upper_bound para la llave k,

Ejemplo

```
#include <cstdlib>
#include <iostream>
#include <vector>

using namespace std;

int main(int argc, char *argv[]){
    char buffer[80];
    double suma;
    vector<double> v;

    v.push_back(999.25);
    v.push_back(888.50);
    v.push_back(777.25);

    suma = 0;

for(int i = 0; i < v.size(); i++){
    suma += v[i];
    cout << v[i] << endl;
}

cout << "----------" << endl;
cout << suma << endl;

system("PAUSE");

return EXIT_SUCCESS;}
```

Tiempos de ejecución

- Operaciones de lista:
  - Complejidad de O(n) normalmente.
  - El manejo de los extremos O(1).
  - En contenedores asociativos suelen caer a O(log n).
- Operaciones de colas y pilas:
  - Tiempo O(1) (O(log n) para colas de prioridad.
- Operaciones asociativas ([ ]):
  - Tiempo constante (en mapas es de O(log n)).

ALGORITMOS

C++ STL Algoritmos

- Algoritmos genéricos
  - Se aplican a una gran rango de tipos (*p.e. se pueden ordenar arrays de enteros o de cadenas)*
  - No necesitan relaciones de herencia
    - Los parámetros no tienen p.q. heredar de una misma clase (ej. de ordenación)
    - Necesitan ser modelos de un algoritmo
- Implementación en C++
  - Se usan plantillas de funciones, interfaces e iteradores para acceder a los contenedores

Ejemplo

Búsqueda lineal: encontrar **`char c`** en un **`string s`**

☛No es general

☛El final de la cadena debe ser siempre '\0'

☛No podemos tener el carácter \0 en la cadena.

```
char *strchr (char* s,char c)
{
while (*s != 0 && *s != c)
    ++s;
return *s==c?s:(char *)0;
}
```

Ejemplo (y2)

- ☛Proporciona el rango de búsqueda
- ☛Asume que el primero está despúes del último

- ☛Se comprueba en el **`while`** si el primer elemento es el búscado.

```
char *find1(char* first, char* last, char c)
{
while(first!=last && *first!=c)
    ++first;
return first;
}
```

Ejemplo (y3)

Observación: una implementación de búsqueda lineal debe:

- indicar la secuencia sobre la que se busca
- representar una posición en la secuencia
- detectar el final de la secuancia
- devolver valores como indicación de éxito o fracaso

Objetivo: garantizar estos requerimientos con flexibilidad y eficacia

Ejemplo (y4)

☛Usar plantillas para parametrizar los argumentos

☛Problema: usa punteros

```
template <typename T>
T *find2(T *first, T *last, const T &value)
{
    while(first!=last && *first!=value)
    ++first;
    return first;
}
```

Ejemplo (y5)

- ☛Es un algoritmo genérico

```
template <class Iterator, class T>
Iterator find(Iterator first,Iterator last,const T& value)
{
    while(first!=last && *first!=value)
    ++first;
    return first;
}
```

FICHEROS

¿Qué problemas existen con la I/O de C?

- **printf** y variantes forman un sistema monolítico: cuando uno emplea **printf** obtiene todo necesario para imprimir cualquier cosa aunque no se vaya a usar.
- Se interpretan en tiempo de ejecución lo que se manda a la salida/toma de la entrada.
- Al postponerse la evaluación no hay comprobación de errores en tiempo de compilación.
- Además, printf() sólo sirve para imprimir tipos de datos de C básicos (**char**, **int**, **float**, **double**, **wchar_t**, **char***, **wchar_t***, y **void***).

Streams

☛Una stream es un objeto que transporta y formatea caracteres de longitud fija.

☛Se pueden tener flujos de entrada (con istream) o de salida (con ostream)

☛Hay diferentes typos de subclases:

- pata ficheros **ifstream**,**ofstream**, y **fstream**,
- **istringstream**, **ostringstream**, y **stringstream** para cadenas

☛Sólo es necesario manejar un único interfaz que se debe aplicar en las nuevas clases.

Ejemplo

```
// Copia un fichero en otro (una línea cada vez)
#include <string>
#include <fstream>
using namespace std;

int main() {
    ifstream in("uno.cpp"); // Abrimos para lectura
    ofstream out("dos.cpp"); // Abrimos para escritura
    string s;
    while(getline(in, s))// Cogemos una línea
    out << s << "\n";    // y la copiamos, getline
//descarta el final de línea.
} ///:~
```

Sobrecarga del operador >>

```
istream& operator>>(istream& is, Date& d)
    is >> d.month;
    char dash;
    is >> dash;
    if(dash != '-')
        is.setstate(ios::failbit);
    is >> d.day;
    is >> dash;
    if(dash != '-')
        is.setstate(ios::failbit);
    is >> d.year;
    return is;
}
```

Sobrecarga del operador <<

```
ostream& operator<<(ostream& os, const Date& d) {
    char fillc = os.fill('0');
     os << setw(2) << d.getMonth() << '-'
    << setw(2) << d.getDay() << '-'
    << setw(4) << setfill(fillc) << d.getYear();
    return os;
}
```

Entrada línea a línea

Hay tres opciones:

- La función miembro get( )
- La función miembro getline( )
- La función global getline( ) definida en <string>

Las dos primeras toman 3 argumentos:

1. Un puntero al buffer dónde se guardará
2. El tamaño del buffer
3. El carácter de terminación ('\n' por defecto)

Entrada/Salida directa

Para lectura/escritura de bloques de bytes se emplean:

- **read:** toma como argumentos una dirección de memoria y el número de bytes a leer.
- **write:** idem.