PRÁCTICA 1
INTRODUCCIÓN A LA PROGRAMACIÓN EN C++

ESTRUCTURAS DE DATOS Y ALGORITMOS
FACULTAD DE INFORMÁTICA

CURSO 2003-2004

Objetivos

El objetivo de la siguiente práctica es conocer el entorno de programación y familiarizarse con algunos aspectos del lenguaje C++ de cara a poder realizar con él el resto de prácticas de la asignatura.

1. Compilando un programa C++

Las prácticas se realizan en el entorno Linux que dispone del compilador `g++`. Si tenemos el fichero `hola.cc` siguiente:

```
1: #include <iostream>
2: using namespace std;
3: int main() {
4:   cout << "Hola mundo!\n";
5: }
```

podemos ejecutar el comando:

```
g++ hola.cc
```

que produce como resultado el fichero ejecutable `a.out`. Resulta conveniente poder indicar el nombre del ejecutable producido, para ello se utiliza la opción `-o` de la manera siguiente:

```
g++ -o hola hola.cc
```

o también

```
g++ hola.cc -o hola
```

Otras opciones interesantes permiten modificar el nivel de verbosidad del compilador (se aconseja utilizar `-Wall`) y el nivel de optimización (para lograr ejecutables más eficientes se aconseja utilizar `-O3`, observa que la letra "o" tiene significados distintos en mayúscula y en minúscula). Probad ejecutar

```
man g++
```

o también[1]

```
info g++
```

[1]Lo puedes llamar desde Emacs con `M-x info`.

para conocer con detalle éstas y otras opciones. En realidad `g++` se trata del ya conocido compilador `gcc` que se invoca con una opción para que reconozca el lenguaje C++ en lugar de C.

Para que el compilador produzca un ejecutable es imprescindible que esté definida la función `main`. También es posible compilar ficheros sin la función `main` y producir lo que se denominan *ficheros objeto*[2] que suelen tener extensión `.o`

Así, si tenemos el fichero `rectangulo.cc` siguiente:

```
1: #include <iostream>
2: using namespace std;
3: void rectangulo(int alto, int ancho, char letra = '*') {
4:   for (int i = 0; i < alto; i++) {
5:     for (int j = 0; j < ancho; j++)
6:       cout << letra;
7:     cout << endl;
8:   }
9: }
```

podemos compilarlo así:

```
g++ -c rectangulo.cc -Wall
```

y nos generará el fichero `rectangulo.o` que podremos enlazar (*link*) posteriormente para formar un fichero ejecutable. Para poder utilizar las funciones de este fichero objeto en nuestros programas basta con declarar las cabeceras de las funciones, no es necesario el cuerpo. De esta manera, podemos tener un fichero `principal.cc` siguiente:

```
1: void rectangulo(int alto, int ancho, char letra = '*');
2: int main() {
3:   rectangulo(4,5);
4: }
```

que se podrá compilar con el comando

```
g++ -o principal principal.cc rectangulo.o
```

para obtener un ejecutable de nombre `principal`

Normalmente, en lugar de declarar las funciones en el fuente que las vaya a utilizar, se escribe un fichero de cabecera (*header*) que suele tener extensión `.h`, en tal caso escribiríamos un fichero denominado[3] `rectangulo.h` que contiene simplemente:

```
1: void rectangulo(int alto, int ancho, char letra = '*');
```

de manera que ahora la función principal queda:

```
1: #include "rectangulo.h"
2: int main() {
3:   rectangulo(4,5);
4: }
```

[2]Esta denominación no tiene nada que ver con la programación orientada a objetos

[3]No es imprescindible, aunque sí aconsejable, utilizar el mismo nombre para los `.h` y los `.cc` asociados.

2. Paso de parámetros en la línea de órdenes

Observa que la función `main` devuelve un valor `int`. En el entorno Unix y Linux los programas tienen la función `main` definida como en el ejemplo siguiente (fichero `muestraargumentos.cc`)

```
1: #include <iostream>
2: using namespace std;
3: int main(int argc, char *argv[]) {
4:   int i;
5:   for (i = 0; i < argc; i++) {
6:     cout << "Argumento " << i << "-esimo es: \"" << argv[i] << "\"\n";
7:   }
8: }
```

Para que el programa pueda acceder a los distintos parámetros que le pasamos al invocarlo. Si ejecutamos este programa así:

```
muestraargumento uno dos y tres
```

aparecerá por pantalla lo siguiente:

```
Argumento 0-esimo es: "muestraargumentos"
Argumento 1-esimo es: "uno"
Argumento 2-esimo es: "dos"
Argumento 3-esimo es: "y"
Argumento 4-esimo es: "tres"
```

3. Entrada y salida con `streams`

En los ejemplos anteriores hemos visto la manera de mostrar texto por la salida estándar utilizando los `streams`. Una alternativa consiste en seguir utilizando las funciones de la librería estándar ISO-C denominada `<stdio.h>` que en C++ se conoce también como `<cstdio>` (las funciones `printf`,`scanf`, etc. ya conocidas por todos).

Para trabajar con `streams` incluimos la librería `<iostream>` que nos ofrece, entre otras cosas, tres objetos de tipo `stream`: `cout`, `cerr` y `cin` que corresponden, respectivamente, a salida estándar, salida de error y entrada estándar. Podemos leer de entrada estándar de la manera siguiente:

```
1: #include <iostream>
2: using namespace std;
3: int main() {
4:   int i,j;
5:   cout << "Dame dos enteros separados por espacios:";
6:   cin >> i >> j;
7:   cout << "El estado de cin es " << cin.fail() << endl;
8:   cout << "Los enteros son: " << i << ", " << j << endl;
9: }
```

Observa que además de mostrar por pantalla los valores introducidos, también mostramos el valor `cin.fail()`. Si ejecutas este programa puede resultar interesante ver qué ocurre con este valor cuando no introduces los números correctamente. Las funciones `scanf`,`fscanf`,`sscanf` tienen la ventaja de retornar el número de argumentos correctamente leídos.

3.1. Leer un fichero

Ya sabemos más o menos leer de entrada estándar. Ahora vamos a ver cómo leer y escribir en ficheros. Para ello utilizamos la librería `<fstream>`. Veamos un ejemplo de programa (`leecadenas.cc` en la página siguiente arriba) que espera como único argumento el nombre de un fichero, lo abre en modo lectura y muestra su contenido cadena por cadena. Esto nos servirá para ver cómo actúa el operador `>>` de un `stream` de entrada cuando ponemos un vector de caracteres.

Este programa comprueba que el fichero ha sido correctamente abierto y utiliza el manipulador `setw(int)` de la librería `<iomanip>` para ajustar el ancho del campo al mostrar el entero `cuenta`. Prueba a ejecutar este programa de la manera siguiente:

```
g++ -o leecadenas leecadenas.cc
leecadenas
leecadenas uno dos y tres
leecadenas nosoynombrefichero
leecadenas leecadenas.cc | less
```

```
1:  #include <iostream>
2:  #include <iomanip>
3:  #include <fstream>
4:  using namespace std;
5:  int main(int argc, char *argv[]) {
6:    int cuenta;
7:    const int longcadena = 1000;
8:    char cadena[longcadena];
9:    if (argc != 2) {
10:     cerr << "Uso del programa: " << argv[0] << " nombre_de_fichero\n";
11:   } else {
12:     fstream fich;
13:     fich.open(argv[1],ios::in); // abrimos en modo lectura
14:     if (!fich) {
15:       cerr << "He tenido problemas para abrir el fichero \""
16:            << argv[1] << "\"\n";
17:     } else {
18:       // mostramos cadena por cadena lo que nos ofrece el fichero
19:       cuenta = 1;
20:       while (!fich.eof()) {
21:         fich >> cadena;
22:         cout << "La cadena " << setw(3) << cuenta
23:              << " es: \"" << cadena << "\"\n";
24:         cuenta++;
25:       }
26:     }
27:     fich.close();
28:   }
29: }
```

Puede resultar de interés leer línea a línea. Para ello utilizamos el método `getline` que recibe tres parámetros: un puntero `char*` al principio de la cadena donde guardar el resultado, el tamaño máximo de esta cadena y, por último, el caracter que delimita la línea. Este último argumento tiene un valor por defecto igual a `'\n'`, es decir, no hace falta ponerlo cuando lo que queremos es leer líneas. Veamos una versión del programa anterior para leer líneas:

```
1:  #include <iostream>
2:  #include <iomanip>
3:  #include <fstream>
4:  using namespace std;
5:  int main(int argc, char *argv[]) {
6:    int cuenta;
7:    const int longlinea = 1000;
8:    char linea[longlinea];
9:    if (argc != 2) {
10:     cerr << "Uso del programa: " << argv[0] << " nombre_de_fichero\n";
11:   } else {
12:     fstream fich;
13:     fich.open(argv[1],ios::in); // abrimos en modo lectura
14:     if (!fich) {
15:       cerr << "He tenido problemas para abrir el fichero \""
16:            << argv[1] << "\"\n";
17:     } else {
18:       // mostramos linea por linea lo que nos ofrece el fichero
19:       cuenta = 1;
20:       while (fich.getline(linea,longlinea)) {
21:         cout << "La línea " << setw(3) << cuenta
22:              << " es: \"" << linea << "\"\n";
23:         cuenta++;
24:       }
25:     }
26:     fich.close();
27:   }
28: }
```

Ejercicio 1 Realiza un programa que recibe un número indeterminado de parámetros que interpretará como nombres de fichero.

- Si el número de argumentos `argc` es 1 (el propio nombre del programa) debe finalizar sin más.
- Si recibe al menos un parámetro (`argc > 1`), el primero se refiere a un fichero destino. El resto de nombres se refieren a ficheros origen que abrirá en modo lectura. El comportamiento del programa consiste en concatenar los ficheros origen en el fichero destino línea a línea poniendo el nombre del fichero origen al principio de cada línea.

Otra librería que nos podría ser útil es `<sstream>` que nos permite utilizar una cadena de caracteres como un `stream`. Veamos un ejemplo. Vamos a combinar dos programas anteriores para que, dado un fichero, nos muestre cada línea y las palabras que contiene:

```
1:  #include <iostream>
2:  #include <iomanip>
3:  #include <fstream>
4:  #include <sstream>
5:  using namespace std;
6:  int main(int argc, char *argv[]) {
7:    int cuenta;
8:    const int longlinea = 1000;
9:    char linea[longlinea];
10:   const int longcadena = 100;
11:   char cadena[longcadena];
12:   if (argc != 2) {
13:     cerr << "Uso del programa: " << argv[0] << " nombre_de_fichero\n";
14:   } else {
15:     fstream fich;
16:     fich.open(argv[1],ios::in); // abrimos en modo lectura
17:     if (!fich) {
18:       cerr << "He tenido problemas para abrir el fichero \""
19:            << argv[1] << "\"\n";
20:     } else {
21:       // mostramos linea por linea lo que nos ofrece el fichero
22:       cuenta = 1;
23:       while (fich.getline(linea,longlinea)) {
24:         cout << "Línea " << setw(3) << cuenta
25:              << ": \"" << linea << "\"\n";
26:         cuenta++;
27:         // ahora mostramos las palabras de línea
28:         istringstream fichlinea(linea); // creamos un stream cadena
29:         int cuenta_cadena = 1;
30:         while (!fichlinea.eof()) {
31:           fichlinea >> cadena;
32:           cout << "Cadena " << cuenta_cadena << "-esima: \""
33:                << cadena << "\"\n";
34:           cuenta_cadena++;
35:         }
36:       }
37:     }
38:     fich.close();
39:   }
40: }
```

4. Trabajando con clases

Antes de aprender a crear nuestras propias clases vamos a utilizar una clase ya existente. El objetivo consisten en aprender a declarar instancias de una clase (vamos, objetos) adecuadamente llamando a un método constructor y posteriormente utilizar los objetos. Para utilizar un objeto se invoca un método sobre él. Para probar estos conceptos os ofrecemos un ejemplo de definición de una clase, denominada `moda`, que nos permite hallar la moda de un conjunto de enteros.

```
1:  class moda {
2:  public:
3:    moda(int tamanyo); // constructor
4:    ~moda(); // destructor
5:    void insertar(int numero);
6:    int obtener_moda();
7:  private:
8:    int tam,*vector;
9:  };
10: moda::moda(int tamanyo) { // valores a insertar entre 0 y tam-1
11:   int i;
12:   tam = tamanyo;
13:   vector = new int[tamanyo];
14:   for (i=0; i<tam;i++)
15:     vector[i] = 0;
16: }
17: moda::~moda() {
18:   delete[] vector;
19: }
20: void moda::insertar(int numero) {
21:   vector[numero]++;
22: }
23: int moda::obtener_moda() {
24:   int i,max,imax;
25:   max = vector[0]; imax = 0;
26:   for (i=1; i < tam; i++) {
27:     if (vector[i] > max) {
28:       max = vector[i];
29:       imax = i;
30:     }
31:   }
32:   return imax;
33: }
```

Ejercicio 2 Escribe un programa que, utilizando la clase `moda`, lea por la entrada estándar una secuencia de número enteros en el rango $[0, 9999]$ –hasta fin de fichero– y halle la moda. Para ello has de crear un objeto de tipo `moda` y utilizar los métodos disponibles. Recuerda que para invocar un método se utiliza la sintaxis:

```
vobjeto.nombre_método(parámetros);  // si es un objeto
pobjeto->nombre_método(parámetros); // si es un puntero a objeto
```

Vamos a ver ahora cómo realizar una estructura de datos tipo cola con una clase que denominamos `cola_float` y que implementaremos utilizando vectores. En el método constructor le daremos el valor máximo de elementos que puede contener la cola y que se utiliza como tamaño del vector. Tendremos los métodos que se muestran:

```
1:  class cola_float {
2:    // en este caso una cola de tipo float
3:    float *vector;
4:    int tamanyo_maximo;
5:    int entra; // índice que apunta a la posición ocupada por 1er elemento
6:    int cuantos; // número de elementos en la cola
7:  public:
8:    cola_float(int tam); // constructor
9:    ~cola_float(); // destructor
10:   bool insertar(float valor); // devuelve si ha cabido
11:   int hay(); // devuelve número de elementos
12:   float extraer(); // si está vacía da igual lo que devuelve
13:   bool extraer(float *deposito); // otra forma, devuelve si había elemento
14: };
15: cola_float::cola_float(int tam) {
16:   if (tam <= 0)
17:     tam = 10; // no permitimos tam <= 0
18:   tamanyo_maximo = tam;
19:   vector = new float[tam];
20:   entra = 0;
21:   cuantos = 0;
22: }
23: cola_float::~cola_float() {
24:   delete[] vector;
25: }
```

Observa que la memoria dinámica se pide y libera en los métodos constructor y destructor.

Ejercicio 3 El objetivo del ejercicio es terminar la implementación de los métodos y la utilización de la clase en un programa que recibe de entrada estándar[4] una serie de líneas del estilo:

```
inserta 3.5
inserta 2.4
extrae
inserta 5.2
extrae
extrae
```

Los valores extraidos se mostrarán por salida estándar. Se debe controlar el caso en que se intenta extraer de la cola vacía.

5. Sobrecarga de funciones y de operadores

Ahora que tenemos algo más de soltura, vamos a profundizar en algunos detalles del lenguaje C++ que nos serán útiles en sucesivas prácticas.

Una de esas características es la sobrecarga de funciones y de operadores, pues nos permitirá, en particular, utilizar los operadores de comparación `<`, `<=`, `>`, `>=`, `==`, etc. sea cual sea el tipo de elemento que disponga de un orden total.

5.1. Sobrecarga de funciones

Ya hemos empleado la sobrecarga de funciones con los métodos `extraer` definidos en la clase `cola_float`, vamos a ver ahora cómo sobrecargar los operadores.

[4]Siempre puedes tener los datos en un fichero y redirigir ese fichero a la entrada estándar.

5.2. Sobrecarga de operadores

La sobrecarga de operadores se puede realizar de dos modos distintos. Si el primer operando es un objeto, podemos sobrecargar el operador en la propia clase. En cualquier caso, podemos definir el operador como una función, aunque algunos operadores sólo pueden definirse del primer modo (ver transparencias de teoría).

Sintácticamente, sobrecargar un operador es como declarar una función poniendo como nombre de la misma la secuencia formada por la palabra reservada `operator` y el operador a sobrecargar. Por ejemplo, vamos a sobrecargar el operador `<<` para insertar elementos en la cola definida anteriormente:

```
1: cola_float& operator << (float valor);
```

y su respectiva implementación:

```
1: cola_float& cola_float::operator << (float valor) {
2:   insertar(valor);
3:   return *this;
4: }
```

Observa que el operador *devuelve* una referencia a un objeto del mismo tipo `cola_float`. La utilidad de devolver este valor es que podemos enlazar esta operación de la manera siguiente:

```
1: cola_float cf(100);
2: cf << 0.1 << 0.2 << 0.3;
```

Ejercicio 4 Se pide sobrecargar los operadores `<<` y `>>` para las operaciones de introducir y extraer elementos de la cola.

Los operadores tienen como desventaja una limitación en flexibilidad. En este caso no podemos emplear la forma de `extraer` que nos indica si el elemento ha sido correctamente extraido.

5.3. Sobrecarga de los operadores para utilizar `streams`

Si sobrecargamos el operador `<<` para la clase `ostream` y un objeto de tipo `cola_float` podremos hacer cosas como esta:

```
1: cout << "La cola es: " << cf << endl;
```

Supondremos que la forma de obtener la salida por pantalla es la siguiente: si tenemos una cola en la que hemos insertado los valores 0,1, 0,2 y 0,3 la salida sería:

```
La cola es: cola[0.3,0.2,0.1]
```

Ejercicio 5 Se pide sobrecargar el operador `<<` tal y como se ha explicado.

Una forma de realizar este ejercicio consiste en acceder desde esa función a los atributos de la clase, si intentas hacer esto el compilador no te dejará porque los atributos están definidos como `private`. Puedes hacerlos públicos o también puedes definir el operador como "función amiga" de la clase poniendo una declaración de esa función en la declaración de la clase y precediéndola de la palabra reservada `friend` de la manera siguiente:

```
1: friend ostream& operator << (ostream& s, cola_float&cola);
```