Análisis y Diseño de Algoritmos I

Práctica 1: Introducción a la programación genérica con C++

Versión 2.0

Índice

Índice de cuadros

Índice de figuras

1. Historia

El lenguaje C fue inventado por Dennis Ritchie, mientras trabajaba para los laboratorios Bell de ATT, a principios de los años 70. Fue creado a partir del lenguaje B, el cual provenía a su vez del BCPL, con el propósito de escribir el sistema operativo UNIX[1] en un lenguaje de más alto nivel que el ensamblador. En los años sucesivos, UNIX se distribuía gratuitamente en las universidades junto con un compilador de C y el código fuente del sistema completo, con lo que ambos gozaron de una gran difusión.

A lo largo de los años surgieron varios dialectos o variantes de C, así como también lenguajes más modernos inspirados en él. Surgió la necesidad de estandarizar o normalizar el lenguaje, para lo que se constituyó en 1985 un comité estadounidense de ANSI. El comité terminó su trabajo en 1989 y el resultado final fue un estándar del lenguaje, al que se llamó C89, con el que resolver el problema de la proliferación de dialectos. La organización internacional de estándares, ISO, formó un comité internacional que hizo suya las propuestas de ANSI con ligeras ampliaciones. En 1999 se realizó una revisión del estándar, dando lugar a lo que se conoce como C99.

Si se desea crear un programa que pueda ser entendido por cualquier persona que conozca C, que no contenga ambigüedades y que pueda transportarse directamente a plataformas distintas de la de desarrollo, se deberá utilizar C ANSI/ISO y evitar sistemáticamente cualquier construcción para la que el estándar especifique que su resultado no está definido o que depende de la implementación.

Hoy día, C es un lenguaje que más de 30 años después de su creación conserva una gran aceptación y prevalencia en áreas muy diversas. Esto es especialmente cierto en la programación de sistemas, que es realmente para lo que fue diseñado.

Sin embargo, para resolver ciertos problemas, sobre todo de programación a gran escala, el lenguaje C se queda corto. No obstante, al ser tan popular y querido por los programadores, han surgido nuevos lenguajes basados en él, sobre los que destaca principalmente el lenguaje C++.

C++ fue inventado por Bjarne Stroustrup alrededor de 1985, quien curiosamente también trabajaba en los laboratorios Bell. En su creación, trató de mantener la máxima compatibilidad con C, añadiendo características que lo hicieran especialmente apropiado para su empleo en simulación, para lo que se inspiró en el lenguaje SIMULA 67.

El nombre del lenguaje deriva del operador de incremento de C, con lo que C++ puede interpretarse como un «C incrementado», un paso más en la evolución del C.[2]

2. Características

El lenguaje C++ es un compromiso entre el bajo y el alto nivel. Compatible con C en un gran porcentaje, mantiene las características de éste que le acercan a los lenguajes ensambladores:

[1] UNIX® es una marca registrada de XOPEN.

[2] Se bromeaba con la posibilidad de que el sucesor de C se llamara D, por ser la siguiente letra del alfabeto en la secuencia B, C; o P, por ser la siguiente letra de BCPL.

operadores y campos de bits, uniones, etc. A su vez, incorpora las construcciones presentes en cualquier lenguaje de alto nivel moderno.

Esto permite al programador escribir código con un nivel de detalle adecuado al problema que desea resolver. En realidad, C++ es un lenguaje multiparadigma, lo que significa que permite varios estilos de programación. Esto es una consecuencia de su evolución.

Por un lado, C++ está construido sobre C, que es un lenguaje esencialmente estructurado. Por otro, Stroustrup desarrolló primero lo que denominó «C con clases», una extensión de C que incorporaba la idea de objeto que aparecía en SIMULA 67: las características generales de los objetos del mismo tipo se definen mediante clases que pueden organizarse jerárquicamente. Finalmente, Stroustrup incluyó el concepto de definición paramétrica que permite un tipo muy potente de genericidad.

En consecuencia, C++ es un lenguaje eficiente y flexible que permite programar directamente siguiendo tres paradigmas distintos:

1. El paradigma de la programación estructurada.
2. El paradigma de la programación orientada a objetos.
3. El paradigma de la programación genérica.

Estos paradigmas no son incompatibles y no es extraño que se utilicen conjuntamente dentro de un mismo programa. De hecho, para muchos programadores, la programación orientada a objetos es la evolución natural de la programación estructurada.

3. Compatibilidad con C

Salvo por algunos detalles, C es un subconjunto de C++. Esto implica que prácticamente cualquier programa válido en C lo es también en C++. Sin embargo, existen diferencias, algunas de ellas sutiles, que pueden hacer que un programa en C no sea aceptado por un compilador de C++ o que presente un comportamiento diferente del esperado.

Se comentan a continuación las diferencias más importantes al objeto de facilitar una hipotética traducción de un programa de C a C++ sin emplear ninguna de las características exclusivas de este último.

C++ mejora diversos aspectos de C regulándolos de manera más estricta. En primer lugar, se prohíben explícitamente algunas malas costumbres:

- La declaración implícita de funciones. Recuérdese que en C se presupone que una función no declarada devuelve un entero y recibe un número indeterminado de parámetros de tipos desconocidos.
- El empleo de **int** como tipo por omisión.[3] Recuérdese que en C se permite omitir el tipo en ciertos contextos, por ejemplo al especificar el tipo de devolución de una función, presuponiéndose éste **int**.

[3] Salvo la posibilidad de obviar el tipo de devolución de *main*().

- La redefinición de un objeto de datos. Recuérdese que en C se permiten múltiples declaraciones de un objeto de datos global siempre que a lo sumo se inicialice una de ellas.
- La inicialización de un vector con más elementos de los declarados. Recuérdese que en C los elementos sobrantes se desprecian.
- La conversión implícita de entero a enumerado.
- La conversión implícita de puntero genérico a otro tipo puntero.

En segundo lugar, proporciona una serie de modificaciones de algunas de sus características que puede ser necesario tener en cuenta:

- Incorpora **bool** como tipo nativo, con los literales booleanos **false** y **true** como palabras reservadas, haciendo que sea innecesario incluir *<stdbool.h>* o utilizar **int** para representar valores booleanos.
- Redefine el tamaño de los literales de carácter a **sizeof**(**char**) y hace que el de las enumeraciones dependa del rango de valores declarado. Recuérdese que en C ambos tamaños equivalen a **sizeof**(**int**).
- Especifica que el nombre de una estructura oculte a cualquier otro nombre idéntico externo al ámbito de su declaración.
- Exige que se emplee la sintaxis de prototipos en la declaración y definición de funciones.
- Especifica que si en el prototipo de una función no aparece ningún parámetro, ésta no puede recibir parámetros. Recuérdese que en C esto significa que nada se conoce acerca del número y tipo de los parámetros, igual que cuando únicamente aparece el especificador de parámetros variables (...).

Por último, hay que tener en cuenta que aparecen nuevas palabras reservadas. Si en un programa en C se emplea un identificador que coincide con alguna de ellas, será necesario cambiar su nombre apropiadamente al traducirlo a C++.

4. Mejoras en el lenguaje

En la parte que C++ comparte con C existen diversas mejoras. Enumeramos a continuación las más importantes. Omitimos la gestión de memoria a través de los operadores **new** y **delete**, los *espacios de nombres* (que comentaremos brevemente al hablar de las mejoras introducidas en la biblioteca estándar), y las *excepciones*.

4.1. Constantes

C++ introduce algunas mejoras menores sobre las constantes que pueden resultar de utilidad:

- Modifica el enlace de las constantes de externo a interno, haciendo factible que aparezcan en los ficheros de cabecera.
- Permite definir constantes cuyo inicializador no sea una expresión constante conocida en tiempo de compilación.

Por ejemplo, supongamos que se desea definir globalmente la constante π calculando $4 \operatorname{arc\,tg} 1$ mediante la función *atan*() de la biblioteca estándar:

const double *pi* = 4.0 ∗ *atan*(1.0);

La definición anterior es válida en C++ pero no en C, ya que su inicializador contiene una llamada a función y no es una expresión constante en el sentido de C.

El empleo de macros para especificar valores constantes debe evitarse en la medida de lo posible, ya que las constantes permiten especificar directamente el tipo deseado amén de otras ventajas.

4.2. Funciones en línea

Puede interesarnos, fundamentalmente por razones de eficiencia, que el código de una cierta función se sustituya en el lugar de cada una de sus llamadas. Para sugerir al compilador que emplee dicha mejora basta anteponer la palabra reservada **inline** a la definición en cuestión.

inline double *cuadrado*(**double** *x*) { **return** *x* ∗ *x*; }

```
inline double distancia(double x0, double y0, double x1, double y1)
{
  return sqrt(cuadrado(x1 − x0) + cuadrado(y1 − y0));
}
```

Por supuesto, las funciones en línea no pueden ser recursivas y deben ser razonablemente cortas, sobre todo si se emplean un gran número de veces a lo largo de un programa. Las funciones en línea tienen enlace interno por lo que son apropiadas para su definición en ficheros de cabecera.

El empleo de macros para especificar código en línea debe evitarse en la medida de lo posible, ya que presenta las desventajas de la sustitución textual, que no aparecen cuando se utilizan funciones en línea.

4.3. Declaraciones

C++ es mucho más flexible que C a la hora de realizar una declaración local. Mientras que C únicamente permite que aparezcan al comienzo de un bloque, C++ las permite casi en cualquier lugar.

Esto puede emplearse, por ejemplo, para reducir el ámbito, la visibilidad o el tiempo de vida de una variable. Especialmente interesante es la posibilidad de emplear declaraciones para los inicializadores del **for**, como se ilustra en el siguiente fragmento:

```
for (int k = 0; k < n; ++k)
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j) {
      double d = p[i][k] + p[k][j];
      if (d < p[i][j])
        p[i][j] = d;
    }
```

Aquí, las variables i, j y k tienen ámbito de **for**, es decir, no existen fuera del bucle que las declara.

Otra mejora introducida por C++ en las declaraciones es que los nombres de las enumeraciones, estructuras y uniones son realmente tipos. Por ejemplo, podemos definir una estructura para almacenar fechas:

```
struct fecha { unsigned short d, m, a; };
```

y declarar un objeto empleando *fecha* en vez de **struct** *fecha* como nombre de tipo. Recordemos que en C es muy común emplear **typedef** para conseguir este efecto:

```
typedef struct { unsigned short d, m, a; } fecha;
```

algo que no es necesario en C++.

4.4. Referencias

Mientras que en C el paso de parámetros es realmente por copia y la única manera de simular el paso por referencia es emplear punteros, C++ incluye primitivamente el concepto de *referencia*: la declaración de referencias es análoga a la de los punteros, sustituyendo $*$ por &.

Por ejemplo, la siguiente función actualiza el espacio e y la velocidad v de un móvil sujeto a una aceleración constante a durante un tiempo t:

```
void actualiza(double& e, double& v, double a, double t)
{
  e += v * t + 0.5 * a * t * t;
  v += a * t;
}
```

y se emplea de la siguiente forma:

```
double e = 20.0, v = 0.0;
actualiza(e, v, 9.8, 1.0);
```

También es común emplear referencias constantes para evitar la copia de un objeto de gran tamaño, análogamente a como se emplean en C los punteros constantes.

4.5. Sobrecarga

C++ permite emplear un mismo nombre de función para realizar acciones distintas, siempre que se pueda distinguir a qué función nos referimos por el número o el tipo de los parámetros empleados. A esto se le conoce como *sobrecarga*.

Siguiendo con el ejemplo anterior, sería interesante disponer de otra función que actualizara la posición de un móvil sometido a una velocidad constante. Podemos mantener el mismo nombre sin que exista ambigüedad, ya que ambas tienen distinto número de parámetros.

```
void actualiza(double& e, double v, double t) { e += v * t; }
```

También es posible sobrecargar operadores análogamente a como se hace con las funciones: basta utilizar la palabra reservada **operator** seguida del nombre del operador como nombre de la función a sobrecargar. Por ejemplo, podemos sobrecargar los operadores básicos de comparación, == y <, para comprobar si dos fechas son iguales o si una es menor que otra, respectivamente:

```
bool operator ==(const fecha& x, const fecha& y)
{
  return x.d == y.d && x.m == y.m && x.a == y.a;
}

bool operator <(const fecha& x, const fecha& y)
{
  return x.a < y.a ||
         (x.a == y.a && x.m < y.m) ||
         (x.a == y.a && x.m == y.m && x.d < y.d);
}
```

4.6. Plantillas

Podemos utilizar la sobrecarga para crear versiones especializadas de una misma función que trabajen con distintos tipos de datos, por ejemplo:

```
inline int cuadrado(int x) { return x * x; }
inline double cuadrado(double x) { return x * x; }
```

Sin embargo, es más interesante definir la función cuadrado, de una vez por todas, de forma que permita calcular el cuadrado de cualquier objeto para el que se defina el operador *. C++ permite este tipo de definiciones paramétricas declarando una *plantilla* mediante la palabra reservada **template** y una lista de parámetros de tipo. Por ejemplo:

```
template <typename T>
inline T cuadrado(T x) { return x * x; }
```

Al no limitar de antemano el tipo de los objetos a emplear, conviene utilizar referencias constantes siempre que sea posible en lugar de los objetos en sí, a fin de evitar una, posible-

mente costosa, copia. Piense que x bien podría ser una matriz enorme o un polinomio de gran longitud, o cualquier otro objeto pesado.

```
template <typename T>
inline T cuadrado(const T& x) { return x * x; }
```

4.7. Clases

Las clases son el mecanismo que proporciona C++ para permitir la programación orientada a objetos. Ya que este paradigma será estudiado en detalle en una asignatura posterior, nos limitaremos a dar una brevísima introducción.

Las clases se introducen en C++ como una generalización de la noción de estructura. En cierto modo, una clase es una estructura de C que puede contener no sólo *miembros de datos*, sino también *funciones miembro*, y que puede especificar qué miembros son visibles desde el exterior y cuáles no.[4] Las funciones visibles desde el exterior forman la *interfaz* de la clase. A una variable de un tipo tal, se la llama *objeto* y al conjunto de valores de sus datos se le denomina *estado interno*.

Esta forma de trabajar es especialmente útil en simulación. Por ejemplo, podemos definir una clase para simular el comportamiento de un móvil al que queremos someter a la acción de fuerzas constantes de intensidad y duración arbitrarias. Comenzamos por su declaración:

```
class movil {
public:
  movil(double m);
  double espacio() const;
  double tiempo() const;
  double velocidad() const;
  double aceleracion() const;
  void aplicaFuerza(double f, double dt);
private:
  double m, e, t, v, a;
};
```

Como se observa, existen dos secciones dentro de esta clase: una pública (**public**) y otra privada (**private**). En la parte privada se declaran los miembros de datos, que quedan ocultos al exterior: la única forma de modificar el estado interno de un objeto que la clase proporciona es a través de su interfaz. El estado interno de un móvil se compone de su masa, espacio recorrido, tiempo transcurrido, velocidad y aceleración.

La declaración de la clase se coloca normalmente en un fichero de cabecera, mientras que la definición se coloca en un fichero separado desde el que se incluye su cabecera. En realidad, se pueden definir las funciones a la vez que se declaran, en cuyo caso se consideran funciones en línea.[5] Si la definición se hace por separado, los nombres de las funciones miembro deberán ir

[4] En el contexto de la orientación a objetos se habla de *atributos* y *métodos* para referirse, respectivamente, a los miembros de datos y a las funciones miembro. También se habla genéricamente de *datos* y *operaciones*.

[5] Por lo tanto, no debe hacerse si la función no es absolutamente simple.

precedidos del de la clase por medio de ::, que se denomina *operador de resolución de ámbito*, de lo contrario, ¿cómo podríamos saber que se trata de miembros de la clase y no de funciones externas a ella?

La primera declaración de la parte pública corresponde a un *constructor*, que es una función especial que se encarga de crear objetos. Su cometido será crear un móvil de la masa indicada, en el origen del espacio-tiempo y sin velocidad ni aceleración. Definámoslo a continuación. Nótese que emplea una sintaxis especial para la inicialización de los miembros de datos, que no se indica el tipo de devolución y que, en este caso, su cuerpo está vacío:

```
movil::movil(double m):
  m(m), e(0.0), t(0.0), v(0.0), a(0.0)
{}
```

Seguidamente, aparecen cuatro funciones miembro que se limitan a devolver una porción relevante del estado interno de un móvil. Es por ello que se denominan *observadoras*. El empleo de **const** en la declaración y en la definición indica precisamente eso: que las funciones no modificarán el estado interno del objeto.

```
double movil::espacio() const { return e; }
double movil::tiempo() const { return t; }
double movil::velocidad() const { return v; }
double movil::aceleracion() const { return a; }
```

A continuación aparece la única función miembro de la interfaz capaz de alterar el estado interno de un móvil. Por esta razón, las funciones de este tipo se denominan *modificadoras*.

```
void movil::aplicaFuerza(double f, double dt)
{
  a = f / m;
  e += v * dt + 0.5 * a * dt * dt;
  v += a * dt;
  t += dt;
}
```

Un ejemplo de utilización de esta clase es el siguiente, donde creamos un móvil, le aplicamos una fuerza constante durante un periodo de tiempo y observamos el espacio recorrido y la velocidad final.

```
movil proyectil(1000.0);
proyectil.aplicaFuerza(1000.0, 150.0);
cout << "Espacio recorrido: " << proyectil.espacio() << " m\n"
     << "Velocidad de impacto: " << proyectil.velocidad() << " m/s" << endl;
```

Un tipo particular de clase muy útil lo constituyen las *clases de envoltorio*, que permiten ocultar los detalles de utilización de una biblioteca bajo una interfaz adecuada. Por ejemplo, he aquí un cronómetro realizado a partir de la función *clock*() de la biblioteca estándar:

```
const double pps = CLOCKS_PER_SEC;
```

```
class cronometro {
public:
  cronometro(): activo(false) {}
  void activar() { activo = true; t0 = clock(); }
  void parar() { if (activo) { t1 = clock(); activo = false; } }
  double tiempo() const { return ((activo ? clock() : t1) - t0) / pps; }
private:
  bool activo;      // Estado de actividad del cronómetro.
  clock_t t0, t1;   // Tiempos inicial y final.
};
```

La clase nos permite crear un cronómetro que podemos activar y parar a voluntad. En cualquier momento podemos obtener el tiempo en segundos transcurrido desde su última activación o, si lo hemos parado, el tiempo medido desde su última activación hasta el momento en que se paró. Por ejemplo, el siguiente fragmento muestra el tiempo que se emplea en el control de un bucle **for** que itera 1 000 000 000 de veces:

```
cronometro c;
c.activar();
for (long i = 0; i < 1000000000L; ++i)
  ;
c.parar();
cout << c.tiempo() << " s" << endl;
```

5. Mejoras en la biblioteca estándar

Existen diversas mejoras y numerosas ampliaciones respecto de la biblioteca de C. En general, ésta forma parte de la de C++: por cada fichero de cabecera de C existe otro en C++ cuyo nombre se obtiene de eliminar el sufijo «*.h*» y añadir como prefijo una «*c*». Así, para disponer del prototipo de la función *sqrt*(), se debe incluir la cabecera <*cmath*> en lugar de <*math.h*>.

La biblioteca estándar se encuentra definida bajo un espacio de nombres especial: *std*, el espacio de nombres estándar, de manera que se evite la colisión accidental de sus nombres con los empleados comúnmente al programar. Por lo tanto, para poder emplearla en un espacio de nombres distinto, por ejemplo, en el espacio de nombres por omisión, es normal hacer:

```
using namespace std;
```

cargar selectivamente los nombres que se van a emplear, por ejemplo:

```
using std::sqrt;
```

o indicarlo explícitamente al utilizarlo, como en:

```
double hip(double x, double y) { return std::sqrt(x * x + y * y); }
```

5.1. Entrada y salida de datos

La entrada y salida de datos en C++ se basa, al igual que en C, en un modelo de flujos. Un flujo no es más que una abstracción para representar una corriente de datos, por ejemplo, una secuencia de caracteres que recibimos o enviamos a un dispositivo.

Recordemos que en C esto se hace a través de las funciones declaradas en la cabecera estándar <*stdio.h*> que define una estructura, llamada *FILE*, que almacena el estado de un flujo. Cuando se abre un fichero con la función *fopen*(), ésta devuelve un puntero a una estructura *FILE* que contiene la información relevante sobre el fichero. Posteriormente, toda la entrada y salida se gestiona a través de funciones que reciben dicho puntero y manipulan el fichero a través de él. Existen en C, así mismo, tres flujos estándar predefinidos: la entrada estándar (*stdin*), la salida estándar (*stdout*) y la salida estándar de errores (*stderr*).

Nunca se insiste lo suficiente en el hecho de que los flujos son abstractos. No existen «el teclado» ni «la pantalla» para un programa en C: *stdin* puede estar recibiendo sus datos de un teclado, pero también de una redirección o de un tubo del sistema operativo; igualmente, *stdout* o *stderr* pueden estar enviando sus datos a una pantalla o a una ventana, pero también a una redirección o a un tubo, quizás a una tarjeta de sonido o a un módem. Es el sistema en el que se ejecuta el programa el que determina de dónde vienen y a dónde van, realmente, los datos que el programa manipula a través de sus flujos.

Es posible emplear el mismo modelo en C++ y exactamente de la misma forma: basta incluir la cabecera estándar <*cstdio*>. Sin embargo, existe una forma mucho mejor e incluso más sencilla: trabajar con el modelo de flujos orientado a objetos que proporciona la biblioteca IOStreams, que forma parte de la biblioteca estándar de C++.

En IOStreams existen tres objetos que representan los flujos estándar predefinidos en C: la entrada estándar (*cin*), la salida estándar (*cout*) y la salida estándar de errores (*cerr*). La entrada de datos se realiza aplicando el operador >> a un flujo de entrada y a un objeto del tipo apropiado (el espacio en blanco se desprecia). La salida de datos se realiza aplicando el operador << a un flujo de salida y al dato en cuestión. También existen funciones para leer o escribir caracteres, líneas, etc.; lo básico está disponible en la cabecera <*iostream*>.

Los operadores >> y << están sobrecargados para cada uno de los tipos elementales y devuelven una referencia al flujo empleado de manera que se puedan aplicar reiteradamente sobre él. Los flujos también pueden ser empleados en un contexto donde se requiera un booleano, por ejemplo, en un **if** o un **while**, en tal caso, un valor **false** indica que se ha producido algún error durante la operación. El siguiente programa ilustra todos estos aspectos.

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

int main()
{
  int n = 0;
  double s = 0.0;
  cout << "Datos numéricos:\n" << endl;
  while (cin) {
    cout << 'x' << n << " = ";
```

```
    double x;
    if (cin >> x) {
      s += x;
      ++n;
    }
  }
  cout << "\nLa media aritmética es " << (n ? s / n : 0) << '.' << endl;
  return 0;
}
```

El manejo de ficheros se realiza a través de las clases *ifstream* y *ofstream*, accesibles desde <*fstream*>. Consúltese [1] para más información.

5.2. Cadenas

C++ dispone de una clase, *string*, accesible desde el fichero de cabecera <*string*>, que permite trabajar cómodamente con cadenas. Las cadenas se pueden definir, inicializar, asignar, comparar, leer, escribir, etc., como si fueran objetos de un tipo primitivo; también se pueden concatenar:

```
string completo;
string nombre = "Segundo", apellido = "Expósito";
apellido += " de Leguineche";
completo = apellido + ", " + nombre;
```

Es posible acceder a sus caracteres individualmente, como con un vector. El tipo apropiado para el índice es *string*::*size_type*, un tipo entero sin signo que depende de la implementación, y la longitud puede obtenerse con las funciones miembro *length*() o *size*(), indistintamente. Por ejemplo, el siguiente programa muestra los parámetros de su línea de órdenes en minúsculas:

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

string minusculas(const string& nombre)
{
  string c = nombre;
  for (string::size_type i = 0; i < c.size(); ++i)
    c[i] = tolower(c[i]);      // Pasa a minúsculas.
  return c;
}

int main(int argc, char *argv[])
{
  for (int i = 1; i < argc; ++i)
    cout << minusculas(argv[i]) << endl;   // Conversión implícita a string.
}
```

Los operadores >> y << están sobrecargados también para las cadenas. Hay que tener en cuenta que >> trabaja como con los tipos elementales, es decir, salta el espacio blanco inicial, lee una secuencia de caracteres hasta el próximo espacio blanco y deja el posible resto de la entrada pendiente. Es por ello que, a veces, la función *getline*() es extremadamente útil: permite leer líneas completas de longitud arbitraria en una cadena.

```
cout << "Nombre completo: ";
string nombre;
getline(cin, nombre);
cout << "Hola, " << nombre << '.' << endl;
```

Existen también diversas funciones de búsqueda y de operación con subcadenas.

5.3. STL

La STL o biblioteca estándar de plantillas[6] se compone principalmente de una serie de *clases contenedoras* y de *algoritmos genéricos* para su manipulación. Un contenedor almacena una colección de elementos y permite realizar ciertas operaciones con ellos. Los algoritmos genéricos trabajan sobre los *iteradores* que proporcionan los contenedores, no sobre éstos en sí. Los iteradores son una abstracción del concepto de puntero: básicamente, permiten recorrer una colección de elementos y acceder a ellos individualmente.

5.3.1. Iteradores

Puede pensarse en un iterador como en una abstracción del concepto de puntero que presenta los elementos de un contenedor como si fueran una secuencia que podemos recorrer, independientemente de que en la organización interna del contenedor esto sea o no cierto. Por ejemplo, un árbol no es una estructura lineal, pero si un contenedor se implementa internamente mediante un árbol y proporciona iteradores apropiados producirá en los algoritmos que lo utilicen la ilusión de estar trabajando con una estructura lineal en la que los elementos pueden recorrerse uno tras otro.

De este modo, los iteradores abstraen también las políticas de recorrido de los contenedores: podrían proporcionarse iteradores que recorrieran el árbol en anchura, pero también otros para recorrerlo en profundidad. Distintos algoritmos de recorrido emplearían el mismo código: sólo cambiaría el tipo de iterador utilizado.

Los iteradores nos permiten trabajar con un contenedor completo o sólo con una parte. Por convenio, para delimitar una secuencia de elementos dentro de un contenedor se emplea una pareja, i y j, de iteradores: i indica la posición del primer elemento de la secuencia y j la posición *posterior* al último (exista o no un elemento en ella). Se dice que esta pareja es un *rango* y se representa formalmente por $[i, j)$ para expresar que el elemento indicado por j queda excluido. Así, $[i, i)$ es un *rango vacío*, ya que no contiene ningún elemento.

Existen otras formas de emplear iteradores para hacer referencia a secuencias de elementos. En ocasiones, se utiliza el iterador inicial y el número de elementos de la secuencia para

[6] Denominada así por las siglas de *standard template library*.

Figura 1: Rango delimitado por dos iteradores, i y j.

delimitarla, dando lugar al rango $[i, i+n)$. Cuando se quieren especificar dos rangos de igual longitud suelen emplearse tres iteradores i, j y k: los dos primeros representan el rango $[i, j)$ de longitud $n = j - i$ y el segundo el rango $[k, k+n)$. También se pueden emplear tres iteradores i, j, y k con $i \leq k \leq j$ para delimitar un rango $[i, j)$ y dos subrangos: $[i, k)$ y $[k, j)$.[7]

Todo iterador sobrecarga los operadores $*$ y $++$ para, respectivamente, acceder al elemento indicado e incrementarse para apuntar al siguiente elemento. Los iteradores se pueden clasificar atendiendo a las operaciones que se permiten sobre ellos, de acuerdo al cuadro 1. La semántica de dichas operaciones es idéntica a la de sus contrapartidas sobre los punteros.

Nombre	Operaciones
Entrada	$*$, $->$ (ambos sólo para lectura), $++$, $==$ y $!=$
Salida	$*$ (sólo para escritura) y $++$
Monodireccionales	Las de los iteradores de entrada y de salida
Bidireccionales	Las de los iteradores monodireccionales más $--$
Acceso directo	Las de los iteradores bidireccionales más $[]$, $<$, $<=$, $>$, $>=$
	$+$ (con iter. y entero) y $-$ (con iter. y entero, o con dos iter.)

Cuadro 1: Categorías de iteradores de la STL.

Observe que los iteradores de acceso directo son los más generales: les son aplicables las mismas operaciones que a los punteros. De hecho, pueden emplearse punteros en cualquier función que requiera iteradores, lo que permite integrar, por ejemplo, cadenas y vectores de bajo nivel en el esquema general de la STL al permitir la aplicación de los algoritmos genéricos sobre punteros en lugar de iteradores.

Cada contenedor proporciona iteradores de la categoría apropiada a las operaciones que se pueden realizar eficientemente sobre ellos.

5.3.2. Contenedores estándar

En el cuadro 2 se presentan los contenedores estándar. Éstos se dividen en *secuencias* (vectores, colas dobles y listas) y *contenedores asociativos ordenados* (conjuntos, multiconjuntos, aplicaciones y relaciones generales).

[7] Aquí $i+n$ representa la posición del n-ésimo elemento tras i, $j-i$ el número de elementos entre i y j, sin incluir j, e $i \leq j$ significa que el elemento indicado por i precede o es igual al indicado por j.

Nombre	Cabecera	Descripción
vector	<*vector*>	Vectores
deque	<*deque*>	Colas dobles
list	<*list*>	Listas
set, *multiset*	<*set*>	Conjuntos y multiconjuntos
map, *multimap*	<*map*>	Aplicaciones y relaciones generales

Cuadro 2: Contenedores de la STL.

Todos estos contenedores comparten una serie de características: se pueden definir, inicializar, asignar y comparar de forma natural, se pueden pasar a una función y ser devueltos normalmente, poseen ciertas definiciones públicas de tipos y también funciones miembro para construir iteradores, insertar elementos, eliminar elementos, etc. Todos los contenedores tienen la capacidad de expandirse y contraerse dinámicamente cuando se asignan unos a otros o se les insertan o eliminan elementos. Esto hace que su manejo sea muy cómodo.

Hay distintas formas de definir e inicializar contenedores que son comunes a todos los contenedores estándar. Definir un contenedor vacío es sencillo, ya que es la forma en que se construyen *por omisión*:

vector<**int**> *t*; *// Vector de enteros vacío.*

Nada impide emplear como parámetro de tipo otro contenedor. Por ejemplo, así se define un vector de vectores de booleanos:

vector<*vector*<**bool**> > a; *// Matriz de booleanos inicialmente vacía.*

Una vez que un contenedor se ha rellenado con algunos datos, puede resultar interesante copiarlo. Para ello, los contenedores pueden construirse *por copia*, inicializándolos con otros del mismo tipo previamente definidos:

vector<**int**> *u*(*t*); *// Copia de t.*
vector<**int**> *v* = *t*; *// Ídem, sintaxis tradicional.*

Otra forma es construirlos *por rango*, suministrando un par de iteradores durante la definición. Esto permite, por ejemplo, copiar los datos de un contenedor a otro de distinta categoría:

deque<**int**> *w*(*v.begin*(), *v.end*()); *// Cola doble con los elementos del vector v.*

Las funciones miembro *begin*() y *end*() permiten construir un rango que engloba a todos los elementos del contenedor y son comunes a todos los contenedores estándar.

Es posible utilizar la construcción por rango para trasvasar datos entre contenedores cuyos elementos sean de tipos distintos pero compatibles. También se pueden emplear punteros como si fueran iteradores y trabajar así con datos alojados en vectores de bajo nivel:

// Vector de bajo nivel. Contiene punteros a carácter que apuntan a los nombres
// de los doce meses del año, representados por cadenas de bajo nivel.

```
const char* const mes[] = {
  "enero", "febrero", "marzo", "abril", "mayo", "junio",
  "julio", "agosto", "septiembre", "octubre", "noviembre", "diciembre"
};

// Listas de cadenas con los nombres de los meses de cada semestre.

list<string> primer_semestre(mes, mes + 6);        // De «enero» a «junio».
list<string> segundo_semestre(mes + 6, mes + 12); // De «julio» a «diciembre».
```

En general, si c es un objeto contenedor de tipo C, sus elementos son de tipo C::*value_type* y quedan comprendidos en el rango $[c.begin(),\ c.end())$, donde *begin*() y *end*() devuelven iteradores de tipo C::*iterator*, o C::*const_iterator* si c es **const**. El recorrido de los elementos de c se expresa de la siguiente forma (sustituya C por cualquier clase contenedora estándar):

```
for (C::iterator i = c.begin(); i != c.end(); ++i)
  // Código para procesar *i, puede que modificándolo.
```

Si c es constante, hay que emplear *const_iterator*. Esto es muy común con las referencias constantes que suelen aparecer como parámetros formales en funciones que reciben contenedores que no desean modificar ni tampoco copiar. También puede hacerse si c no es constante pero se desea dejar claro que no se va a modificar:

```
for (C::const_iterator i = c.begin(); i != c.end(); ++i)
  // Código para procesar *i sin modificarlo.
```

Salvo los vectores y las colas dobles, que proporcionan iteradores de acceso aleatorio, el resto de los contenedores estándar proporcionan iteradores bidireccionales.

Puede averiguar si c está vacío comprobando $c.empty()$ o calculando cuántos elementos contiene con $c.size()$, que devuelve un valor de tipo C::*size_type*. No emplee *size*() para comprobar si un contenedor está vacío: con una lista puede ser $O(n)$, mientras que *empty*() es $O(1)$. Por la misma razón, evite calcular *size*() a cada iteración de un bucle, en concreto, en su condición.

```
C::size_type n = c.size();
for (C::size_type i = 0; i < n; ++i)    // Evitamos recalcular el tamaño.
  // ...
```

Se pueden eliminar elementos de manera similar en todos los contenedores estándar:

```
c.clear();          // Elimina todos los elementos.
c.erase(k);         // Elimina el elemento indicado por el iterador k.
c.erase(i, j);      // Elimina los elementos del rango [i, j).
```

Secuencias

Aparte de las formas ya presentadas, las secuencias se pueden definir indicando un número inicial de elementos y opcionalmente un valor para todos ellos que, si se omite, será el valor por omisión del tipo parámetro.

```
vector<int> u(10);                          // Vector de 10 enteros inicializados a 0.
vector<int> v(10, 1);                       // Vector de 10 enteros inicializados a 1.
list <string> l(100);                       // Lista de 100 cadenas vacías.
list <string> l(100, "No presentado");      // Lista de 100 cadenas con el valor indicado.
```

Recuerde que nada impide emplear como parámetro de tipo otro contenedor; aquí *a* se define como un vector de diez vectores de diez booleanos inicializados a **true**:

```
vector<vector<bool> > a(10, vector<bool>(10, true));
```

Las secuencias poseen *front*() y *back*() para acceder, respectivamente, al primer y al último elemento de cualquier secuencia que no esté vacía. Del mismo modo, siempre es posible insertar al final con *push_back*(). En las colas dobles y las listas se puede además insertar al principio con *push_front*(). Todas estas operaciones tardan un tiempo constante.[8]

Como contrapartida, los vectores (y también las colas dobles), poseen acceso directo a sus elementos mediante la sobrecarga del operador [] , con lo que se puede trabajar con ellos como si fueran vectores de bajo nivel. A diferencia de las operaciones anteriores, [] no expande dinámicamente el contenedor por lo que no debe accederse nunca a un elemento inexistente.

Conviene emplear **typedef** para simplificar expresiones complejas de tipo que se repitan profusamente:

```
typedef vector<vector<bool> > Adyacencia;
typedef Adyacencia::size_type Indice;

bool simetrica(const Adyacencia& m)
{
  const Indice n = m.size();
  for (Indice i = 0; i < n - 1; ++i)
    for (Indice j = i + 1; j < n; ++j)
      if (m[i][j] != m[j][i])
        return false;
  return true;
}
```

Contenedores asociativos ordenados

Los contenedores asociativos ordenados permiten una búsqueda de información basada en claves. La información se mantiene ordenada por la clave, respecto de un determinado orden estricto, lo que permite proporcionar operaciones muy eficientes.

El contenedor *set* se emplea para representar conjuntos, es decir, colecciones de elementos de un mismo tipo, sin repeticiones. Podemos representar multiconjuntos con *multiset*, que permite que los elementos se repitan. En ambos contenedores los elementos almacenados son, a su vez, las claves.

[8] En los vectores, *push_back*() es de *tiempo amortizado* constante. Esto significa que el tiempo de un *push_back*() puede ser sensiblemente superior a $O(1)$, puede que $O(n)$, pero el tiempo total de una secuencia de n de ellos no supera $O(n)$, lo que, si se amortiza a lo largo de las n operaciones, nos da un tiempo constante.

El contenedor *map* representa una relación funcional o *aplicación* entre claves y valores. Esto significa que a cada clave sólo corresponde un valor, es decir, que cada clave es única. Sin embargo, el contenedor *multimap* representa una relación general entre claves y valores que permite que a una misma clave puedan corresponder varios valores. La información en ambos casos se almacena internamente mediante pares clave-valor. En un *multimap*, por cada valor que corresponda a una clave se almacena un par clave-valor. En general, el tipo de las claves y el de los valores no tiene por qué coincidir.

Los pares son objetos de una clase muy simple, llamada *pair*, que posee dos miembros de datos accesibles desde el exterior, *first* y *second*, con la clave y el valor, respectivamente. Aparte de en las cabeceras de los contenedores asociativos, se encuentran en <*utility*>. Los pares se pueden inicializar durante su definición y también construirse con la función *make_pair*():

```
pair<string, double> e("e", exp(1.0)), pi;     // El segundo recibe valores por omisión.
pi = make_pair("pi", 4.0 * atan(1.0));
```

Existen formas comunes de insertar nuevos elementos. En el caso de *set* y *multiset*, cada elemento es una clave. Para *map* y *multimap*, cada elemento es un par clave-valor. Se puede insertar un elemento individual o todo un rango:

```
c.insert(x);        // Inserta el elemento x.
c.insert(i, j);     // Inserta los elementos del rango [i, j).
```

Las operaciones de búsqueda reciben siempre una clave y buscan elementos cuyas claves cumplan una cierta propiedad respecto a ella:

```
c.count(x);         // Devuelve el número de elementos con clave equivalente.
c.find(x);          // Devuelve un iterador al primer elemento con clave equivalente.
c.lower_bound(x);   // Devuelve un iterador al primer elemento con clave no menor.
c.upper_bound(x);   // Devuelve un iterador al primer elemento con clave mayor.
c.equal_range(x);   // Devuelve un par con c.lower_bound(x) y c.upper_bound(x).
```

Tanto *find*() como *lower_bound*() y *upper_bound*() devuelven *c.end*() si no pueden encontrar un elemento con la propiedad requerida. Las tres últimas operaciones tienen su principal aplicación en *multiset* y *multimap*, ya que *equal_range*() calcula un rango de iteradores cuyos elementos comparten la misma clave, como se ilustra en la figura 2. Si el elemento no aparece, se obtiene un rango vacío.

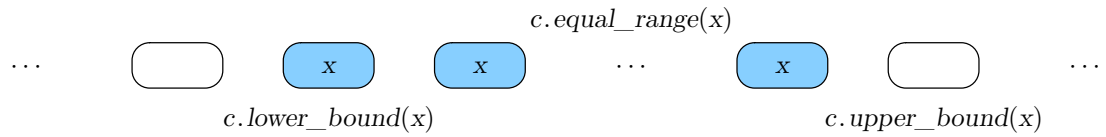

Figura 2: Búsqueda con *equal_range*().

El contenedor *map* sobrecarga el operador [] para acceder al valor correspondiente a una clave. Si durante el acceso la clave no existe, se inserta automáticamente un par con dicha clave y el valor por omisión correspondiente, de manera que la operación no falle. Esto permite emplear una notación muy cómoda para manipular un *map* al estilo de una tabla asociativa:

map<*string*, **double**> *tabla*;
tabla[`"e"`] = *exp*(1.0);
tabla[`"phi"`] = 0.5 ∗ (1.0 + *sqrt*(5.0));
tabla[`"pi"`] = 4.0 ∗ *atan*(1.0);
tabla[`"1 / pi"`] = 1.0 / *tabla*[`"pi"`];

Si se recorre un *map* o un *multimap* con un iterador se obtienen los pares clave-valor al desreferenciar éste. Por ejemplo, para mostrar la tabla del ejemplo anterior, se puede hacer:

for (*map*<*string*, **double**>::*const_iterator i* = *tabla.begin*(); *i* != *tabla.end*(); ++*i*)
  cout << *i*->*first* << `"\t= "` << *i*->*second* << *endl*;

Hay que insistir en que, si bien las nociones matemáticas de conjunto, multiconjunto o relación no implican la existencia de un orden entre los elementos, estos contenedores exigen que se les proporcione un orden adecuado; de ahí el apelativo de «ordenados». Se emplea un orden estricto entre los elementos a almacenar para obtener una organización interna eficiente. Por omisión, este orden vendrá dado por el operador < del tipo en cuestión. Siguiendo con el ejemplo anterior, se imprime:

```
1 / pi  = 0.31831
e       = 2.71828
phi     = 1.61803
pi      = 3.14159
```

de manera que `"1 / pi"` aparece al principio y no al final, ya que, como cadena, es la menor de las claves del contenedor.

Para comprender mejor la diferencia entre *map* y *multimap* supongamos que vamos a diseñar un sistema de gestión para correos y queremos mantener la información que nos permita acceder eficientemente al código postal de una calle y a todas las calles que tienen el mismo código postal. El código postal de una calle es único: una calle sólo puede tener un código postal y, naturalmente, un código postal hace referencia a varias calles. Podemos emplear un *map* y un *multimap* para representar la información pertinente:

map<*string*, **int**> *calle_cp*; // *Relaciona cada calle con su CP.*
multimap<**int**, *string*> *cp_calle*; // *Relaciona cada CP con sus calles.*

En este caso, se dice que existe una relación de «uno a varios» entre los códigos postales y las calles. Otra opción hubiera sido sustituir el *multimap* por un *map* que a cada código postal asociara una lista de calles:

map<**int**, *list*<*string*> > *cp_calle*; // *Relaciona cada CP con la lista de sus calles .*

Si recorremos los contenedores *calle_cp* y *cp_calle*, el primero nos presentará sus pares ordenados por calle (en el orden lexicográfico de las cadenas de caracteres), mientras que el segundo lo hará por código postal (en el orden numérico ordinario).

5.3.3. Adaptadores de secuencia

Un *adaptador* es un componente que modifica la interfaz de otro. La STL define tres adaptadores de secuencia, que funcionan como «envoltorios» que restringen las operaciones a realizar sobre los contenedores subyacentes y les proporcionan nombres apropiados.

Nombre	Cabecera	Descripción
stack	<*stack*>	Pilas
queue, *priority_queue*	<*queue*>	Colas simples y de prioridades

Cuadro 3: Adaptadores de secuencia de la STL.

Estos adaptadores poseen operaciones *empty*() y *size*() idénticas a las de los contenedores estándar. Además, todos tienen una operación *push*(), para introducir un elemento, y otra, *pop*(), para eliminar el elemento correspondiente según la disciplina del adaptador.[9] Aparte de esto, las pilas poseen *top*() para acceder a la cima, las colas simples *front*() y *back*() como las secuencias, y las colas de prioridades *top*() para obtener el elemento de mayor prioridad.

Las colas de prioridades requieren que los elementos se puedan comparar entre sí apropiadamente, de forma que se pueda establecer entre ellos un orden.

Los adaptadores no son contenedores propiamente dichos y por lo tanto carecen de algunas de sus características. Principalmente, hay que tener en cuenta que no proporcionan iteradores de ningún tipo para su recorrido.

5.3.4. Algoritmos

Existen algunos algoritmos simples que pueden aplicarse a elementos de cualquier tipo para los que su definición tenga sentido. Los más comunes son:

```
min(x, y);      // Mínimo de x e y.
max(x, y);      // Máximo de x e y.
swap(x, y);     // Intercambia x e y.
```

Sin embargo, la potencia de la STL radica en gran medida en sus muchos algoritmos genéricos. Éstos son independientes del contenedor, en el sentido de que son capaces de operar con cualquiera que les proporcione los iteradores apropiados, ya que trabajan precisamente con iteradores, no con los contenedores en sí. Como muestra, valgan los siguientes:

```
min_element(i, j);      // Iterador al (primer) mínimo elemento del rango [i, j).
max_element(i, j);      // Iterador al (primer) máximo elemento del rango [i, j).
reverse(i, j);          // Invierte los elementos del rango [i, j).
random_shuffle(i, j);   // Baraja aleatoriamente los elementos del rango [i, j).
sort(i, j);             // Ordena los elementos del rango [i, j).
```

[9] «Último en entrar» para pilas, «primero en entrar» para colas simples y «mayor prioridad» para colas de prioridades.

Los dos primeros requieren iteradores monodireccionales, el tercero, bidireccionales, y los dos últimos, de acceso directo.[10] Todos estos algoritmos aparecen declarados en <*algorithm*>.

6. Programación genérica

La plantillas permiten obtener directamente un alto grado de genericidad. Por ejemplo, consideremos la siguiente función, que decide si un vector (de la STL) está ordenado:

```
template <typename T>
bool ordenado(const vector<T>& v)
{
  typedef typename vector<T>::size_type I;
  I i = 0;
  const I n = v.size();
  for (I k = 1; k < n; ++k) {
    if (v[k] < v[i])
      return false;
    i = k;
  }
  return true;
}
```

Su definición es paramétrica y adecuada para cualquier vector de un tipo arbitrario T. Lo único que se exige al vector es que sus elementos sean comparables mediante el operador $<$ para que la expresión $v[k] < v[i]$ tenga sentido. Por supuesto, para que el algoritmo sea correcto, este operador deberá representar una relación de orden sobre T.

Sin embargo, este algoritmo no funcionaría para una lista, ya que, aunque se sustituya *vector*<*T*> por *list*<*T*>, las listas carecen de operador de acceso. La razón es simple: los objetivos de diseño de vectores y listas son bien distintos. Mientras que los vectores proporcionan acceso directo a un número predeterminado de elementos, las listas proporcionan acceso secuencial a un número indeterminado de ellos. Los creadores de la STL podrían haber sobrecargado **operator** [] para las listas, pero únicamente habría servido para crear una ilusión: los accesos no serían realmente directos y su empleo sería, en general, ineficiente.

Esto es lamentable porque, en el fondo, la idea para comprobar si un vector, una lista o *cualquier secuencia* está ordenada es la misma: basta recorrerla comprobando que no exista ningún elemento que sea menor que su predecesor. Para que la palabra «predecesor» tenga aquí sentido, el recorrido ha de comenzar por el segundo elemento; si hay menos de dos elementos, ya se sabe que está ordenada.

Cualquier secuencia puede recorrerse empleando iteradores, luego podemos definir una función que, en vez de trabajar con un vector genérico, lo haga con un par de iteradores de un tipo arbitrario I. Este par representaría un rango $[i, j)$.

[10] Es cierto que *min_element*() y *max_element*() pueden, y suelen, diseñarse para requerir únicamente iteradores de entrada, pero la cuestión es que el estándar sólo exige que proporcionen iteradores monodireccionales. Probablemente, se sacrifica generalidad en aras de no excluir otros diseños eficientes que requieren que los iteradores no sólo sean de entrada, sino también de salida.

```
template <typename I>
bool ordenado(I i, I j)
{
  if (i != j) {
    I k = i; ++k;              // Y no k = i + 1, o restringiría el iterador.
    for (; k != j; ++k) {
      if (*k < *i)
        return false;
      i = k;
    }
  }
  return true;
}
```

La función resultante no sólo es válida para vectores y listas, sino para cualquier secuencia, sea o no de la STL: lo único que se requiere es que proporcione iteradores apropiados. En este caso, lo que se exige a los iteradores (aparte de que se les pueda inicializar y asignar) es que sea posible comprobar su igualdad, incrementarlos y acceder a través de ellos al elemento indicado.

Como se observa los requisitos son mínimos. Es lo menos que se puede pedir a un iterador útil y, de hecho, todos los contenedores de la STL proporcionan iteradores que poseen las operaciones necesarias. Por ejemplo, si tenemos

```
vector<double> v;
list <string> l;
```

es posible preguntar por *ordenado*(*v.begin*(), *v.end*()) y *ordenado*(*l.begin*(), *l.end*()).

Aquí radica la esencia de la programación genérica, en desarrollar soluciones lo más generales posibles a los problemas, imponiendo las menores restricciones posibles a los datos de entrada y abstrayendo los detalles irrelevantes de su organización y manipulación.

Se han presentado sólo los conceptos fundamentales relacionados con la programación genérica en C++ con la STL. Véanse las referencias, y en especial [2, 3], para más información. En [4] se encuentra disponible un buen manual de referencia sobre la STL para su consulta.

Referencias

[1] Stroustrup, Bjarne. *The C++ programming language. Special edition.* Addison-Wesley. 2000.

[2] Austern, Matthew H. *Generic programming and the STL. Using and extending the C++ standard template library.* Addison-Wesley. 1998.

[3] Musser, David R. y Saini, Atul. *STL tutorial and reference guide. C++ programming with the standard template library.* Addison-Wesley. 2.ª ed. 2001.

[4] Silicon Graphics, Inc. Standard template library programmer's guide. `http://www.sgi.com/tech/stl`