Algoritmos en C++

Rodrigo Alexander Castro Campos
Última actualización: 2 de mayo de 2015

Este documento presenta una introducción a la biblioteca de algoritmos de C++ pensada para estudiantes que están cursando actualmente la UEA "Algoritmos y Estructuras de Datos" de la UAM Azcapotzalco. Si bien la elección de C como lenguaje para dicha UEA no presenta problemas (C es capaz de implementar eficientemente las estructuras de datos y algoritmos que se ven en dicho curso y la idea es que los alumnos aprendan a implementarlas), los alumnos que decidan seguir programando en C están destinados a reimplementar las utilidades anteriores cada vez que las necesiten en UEA posteriores o durante la resolución de problemas específicos. El objetivo de este documento es presentar al alumno la biblioteca de algoritmos de C++ y motivarlo a aprender a utilizarla con el fin de hacerlo más productivo al momento de escribir código.

Este documento comienza retomando brevemente el tema de apuntadores, que casi siempre es visto en la UEA "Programación Estructurada" únicamente en el contexto de paso de variables por referencia. Posteriormente se explicará de manera detallada la relación entre arreglos y apuntadores y estudiaremos la implementación con apuntadores de algoritmos elementales del nivel de "Programación Estructurada". Finalmente analizaremos la construcción genérica y eficiente de algunos algoritmos de ordenamiento en términos de algoritmos fundamentales mucho más sencillos. Prácticamente la totalidad de los algoritmos más fundamentales están ya implementados en la biblioteca de C++ y esto se hará notar conforme corresponda. A su vez, se irán introduciendo las características de C++ necesarias para entender el uso y la implementación de la biblioteca de C++.

Apuntadores

Un apuntador es una variable que es capaz de "apuntar" o "denotar" a otra. El tipo de un apuntador se escribe usando el tipo de la variable a denotar y el operador `*`. En una expresión, el operador *&* prefijo obtiene un apuntador a la variable en cuestión:

```
int a;
int* p = &a;        // p es un apuntador que denota a a
```

Usando apuntadores es posible modificar indirectamente el valor de una variable. El operador prefijo `*` sobre un apuntador obtiene la variable siendo denotada por éste. A esta operación se le denomina desreferencia del apuntador:

```
int a = 0;
int* p = &a;        // p es un apuntador que denota a a
*p = 5;             // modifica la variable siendo denotada por p
printf("%d", a);    // imprime 5
```

En caso de ser necesario, es posible hacer que un apuntador denote a una variable diferente:

```
int a = 0, b = 0;
int* p = &a;        // p es un apuntador que denota a a
p = &b;             // p ahora denota a b
*p = 5;             // modifica la variable siendo denotada por p
printf("%d %d", a, b);       // imprime 0 5 (a no fue modificada)
```

Paso por referencia y algoritmo de intercambio

El algoritmo de intercambio de dos variables es un algoritmo comúnmente usado para ilustrar el paso por referencia de variables a una función. Además, un alumno de computación nota rápidamente que el algoritmo de intercambio aparece como parte de muchos otros algoritmos, como en los algoritmos de ordenamiento basados en comparaciones.

En el lenguaje C el paso por referencia no existe sino que se simula mediante indirección usando apuntadores. La implementación como función en C del algoritmo de intercambio para dos variables de tipo `int` es la siguiente:

```
void intercambia(int* a, int* b)
{
   int temp = *a;
   *a = *b;
   *b = temp;
}

int a = 1, b = 2;
intercambia(&a, &b);
printf("%d %d", a, b);        // imprime 2 1
```

Por otro lado, el lenguaje C++ (además de incluir apuntadores) incluye también el concepto de referencia. En C++ una referencia es el equivalente a un alias de una variable:

```
int a = 0;
int& r = a;          // r es un alias de la variable a
r = 5;               // modifica a mediante su alias r
printf("%d", a);     // imprime 5
```

Una referencia se declara escribiendo el tipo de la variable a referir y el operador `&`. El uso de este operador no debe confundirse con el uso del operador prefijo & en una expresión (que sirve para obtener un apuntador a una variable):

```
int a = 0;
int& r = a;          // r es un alias a la variable a
r = 1;               // modifica a mediante su alias r
int* p = &a;         // p es un apuntador que denota a a
*p = 2;              // modifica a mediante desreferencia de p
```

Las referencias permiten presentar una implementación más sencilla (sintácticamente y conceptualmente) del algoritmo de intercambio:

```
void intercambia(int& a, int& b)
{
   int temp = a;
   a = b;
   b = temp;
}

int a = 1, b = 2;
intercambia(a, b);
printf("%d %d", a, b);        // imprime 2 1
```

Cuando una variable se desea pasar a una función y dicha función no necesita modificar la variable, lo normal es usar paso por valor. Sin embargo cuando se usan tipos `struct` muy grandes, la copia puede ser costosa:

```
struct alumno {
   char nombre[100];
   int edad;
   int calificaciones[100];
};

int devuelve_edad(alumno a)
   // hace una copia de todos los miembros del struct
{
   return a.edad;
}
```

Pasar por referencia evita la copia pero permite modificar la variable. Una alternativa es declarar referencias de sólo lectura mediante la palabra `const`: en caso de intentar modificar la variable usando la referencia en cuestión, se producirá un error de compilación:

```
int devuelve_edad(const alumno& a)
   // paso por referencia a const, no se hace una copia
{
   a.edad = 20;          // error, a es constante
   return a.edad;
}
```

A no ser de que se tenga la seguridad de que el parámetro es de un tipo cuya copia es barata, deberá preferirse el paso por referencia a `const` cuando no se requiera modificar la variable.

Implementación genérica del algoritmo de intercambio

Es fácil que ver que el algoritmo de intercambio es independiente del tipo de las variables que se estén intercambiando:

```
void intercambia(int& a, int& b)
{
   int temp = a;
   a = b;
   b = temp;
}

void intercambia(float& a, float& b)
{
   float temp = a;
   a = b;
   b = temp;
}
```

El código anterior es válido en C++ pero inválido en C. El poder declarar varias funciones con el mismo nombre pero con diferentes parámetros se denomina sobrecarga de funciones y es una

característica que C no posee. En C++ la elección de la función a invocar se determina mediante los argumentos usados en la invocación:

```
int a = 0, b = 1;
intercambia(a, b);       // llama a intercambia(int&, int&)
float e = 0, f = 1.5;
intercambia(e, f);       // llama a intercambia(float&, float&)
```

Aunque la sobrecarga de funciones es muy útil, el poder crear tipos de dato definidos por el usuario (tipos `struct`) potencialmente significa tener que definir una sobrecarga adicional para cada nuevo tipo de dato, lo cual es tedioso. C++ permite generalizar la función para cualquier tipo mediante un nombre simbólico para el tipo en cuestión, usando la siguiente sintaxis:

```
template<typename T>
void intercambia(T& a, T& b)
{
   T temp = a;
   a = b;
   b = temp;
}

int a = 0, b = 1;
intercambia(a, b);     // llama a intercambia(T&, T&) con T = int
float e = 0, f = 1.5;
intercambia(e, f);     // llama a intercambia(T&, T&) con T = float
```

Si los tipos de los argumentos usados en la invocación no son iguales entonces se genera un error de compilación debido a la ambigüedad en la deducción del tipo `T`:

```
int a = 0;
float f = 1.5;
intercambia(a, f);     // error (¿T = int o T = float?)
```

La característica anterior (denominada plantillas o `templates`) es la principal manera en la que C++ facilita el desarrollo bajo el paradigma de la programación genérica. Prácticamente la totalidad de la biblioteca estándar de C++ (posiblemente toda, excluyendo la biblioteca de C la cual también está incluída en C++) está diseñada tomando en cuenta este paradigma. Es fácil darse cuenta de que es tremendamente útil en la práctica.

Primer acercamiento a la biblioteca de C++

La biblioteca de C++ incluye una implementación genérica del algoritmo de intercambio. Lo anterior quiere decir que es innecesario reimplementar el algoritmo de intercambio en C++. La biblioteca incluye tanto la versión de intercambio mediante apuntadores (llamada `iter_swap`) como la versión de intercambio mediante referencias (llamada `swap`).

Para poder usar tanto `swap` como `iter_swap` es necesario incluir `algorithm` (nótese que el nombre no tiene extensión `.h` como ocurre con la biblioteca de C). La biblioteca estándar de C++ además está contenida en el espacio de nombres `std`, por lo que de manera predeterminada es necesario utilizar el prefijo `std::` para poder utilizar dichas funciones:

```
#include <algorithm>
#include <stdio.h>

int main( )
{
   int a = 1, b = 2;
   std::swap(a, b);
   printf("%d %d\n", a, b);   // imprime 2 1
   std::iter_swap(&a, &b);
   printf("%d %d\n", a, b);   // imprime 1 2
}
```

En caso de que se desee evitar el uso del prefijo `std` una alternativa es utilizar una directiva `using`, la cual inyecta todos los nombres del espacio de nombres indicado en el ámbito de la directiva:

```
#include <algorithm>
#include <stdio.h>
using namespace std;

int main( )
{
   int a = 1, b = 2;
   swap(a, b);                // no necesita std::
   printf("%d %d\n", a, b);
   iter_swap(&a, &b);         // no necesita std::
   printf("%d %d\n", a, b);
}
```

Por diversas razones que están más allá del alcance de este documento, nosotros no utilizaremos esta característica.

Arreglos y apuntadores

Un arreglo es una secuencia de elementos del mismo tipo y contiguos en memoria. Por razones históricas (y salvo contadas excepciones de las cuales no hablaremos), el nombrar un arreglo es equivalente a obtener un apuntador al primer elemento del mismo:

```
int arr[3];
int* p1 = &arr[0];  // p1 apunta al primer elemento de arr
int* p2 = arr;      // p2 apunta al primer elemento de arr
```

Cuando un apuntador denota un elemento de un arreglo, es posible avanzar el apuntador al siguiente elemento del arreglo incrementando el apuntador:

```
int arr[3];
int* p = arr;       // p apunta al primer elemento de arr
*p = 0;             // modifica el primer elemento de arr
++p;                // p avanza al siguiente elemento de arr
*p = 1;             // modifica el segundo elemento de arr
++p;                // p avanza al siguiente elemento de arr
*p = 2;             // modifica el tercer elemento de arr
printf("%d %d %d", arr[0], arr[1], arr[2]);   // imprime 0 1 2
```

También es posible hacer retroceder el apuntador decrementando el mismo. Sumar un entero a un apuntador es equivalente a realizar ese mismo número de incrementos (o decrementos si el entero es negativo):

```
int arr[3] = { 0, 0, 0 };
int* p = arr;       // p apunta al primer elemento de arr
p += 2;             // p avanza al tercer elemento de arr
*p = 2;             // modifica el tercer elemento de arr
--p;                // p retrocede al segundo elemento de arr
*p = 1;             // modifica el segundo elemento de arr
printf("%d %d %d" arr[0], arr[1], arr[2]);   // imprime 0 1 2
```

En general, sumar un entero `n` a un apuntador es mucho más rápido que incrementar o decrementar `n` veces el mismo dentro de un ciclo (la suma se realiza en tiempo constante). De hecho, el acceso a los elementos de un arreglo es reescrito internamente por el compilador como un acceso mediante apuntador:

```
int arr[3];
arr[2] = 5;         // modifica el tercer elemento de arr
*(arr + 2) = 5;     // modifica el tercer elemento de arr
```

La suma de apuntadores no está definida. La resta de dos apuntadores `p - q` devuelve la cantidad de veces que es necesario incrementar `q` (o decrementarlo, lo cual se indica con un resultado negativo) para alcanzar a `p`:

```
int arr[3];
int* p = arr + 2;
int* q = arr;
printf("%d\n", p - q);        // imprime  2 (p == q + 2)
printf("%d\n", q - p);        // imprime -2 (q == p - 2)
```

La manera típica de iterar sobre los elementos de un arreglo es mediante un índice y un ciclo `for`. Usando apuntadores, una manera alternativa es la siguiente:

```
int arr[3] = { 10, 5, 7 };
int* p = arr;             // p apunta al primer elemento de arr

while (p <= &arr[2]) {    // mientras p esté sobre elementos de arr
   printf("%d\n", *p);    // imprime el elemento siendo visitado
   ++p;                   // avanza al siguiente elemento
}
```

Tanto C como C++ garantizan que puede construirse la dirección del elemento que seguiría al último elemento de un arreglo (para el ejemplo anterior ésta sería `&arr[3]` o de manera equivalente, `arr + 3`). Esto no quiere decir que el elemento en dicha posición pueda ser leído o sobreescrito pues estrictamente no existe en el arreglo. Lo anterior permite reescribir el ciclo:

```
int arr[3] = { 10, 5, 7 };
int* p = arr;             // p apunta al primer elemento de arr

while (p < arr + 3) {     // mientras p esté sobre elementos de arr
   printf("%d\n", *p);    // imprime el elemento siendo visitado
```

```
   ++p;                    // avanza al siguiente elemento
}
```

Diremos que un apuntador denota el fin del arreglo si denota la posición después del último elemento del mismo (en el ejemplo anterior, `arr + 2` es un apuntador al último elemento del arreglo mientras que `arr + 3` es un apuntador al fin del arreglo). Es fácil deducir que lo siguiente es cierto:

```
int arr[10];
int* ult = arr + 9;        // &arr[9], el último elemento de arr
int* fin = arr + 10;       // uno después del último, el fin de arr
printf("%d\n", ult - arr);  // imprime 9
printf("%d\n", fin - arr);  // imprime 10, que es el tamaño de arr
```

Es decir, la resta del apuntador al fin menos el apuntador al inicio es el número de elementos del arreglo (lo cual no ocurre si se utiliza el apuntador al último, que está desfasado en -1).

Cálculo del mínimo de una secuencia de elementos

Aunque muchos compiladores de C y C++ lo permiten, estrictamente es ilegal declarar un arreglo con cero elementos. Con esto en mente, podemos implementar una función que toma un arreglo y devuelva el elemento más pequeño del mismo:

```
int minimo_elemento(int* arr, int n)
{
   int res = arr[0];  // iniciamos creyendo que el menor es arr[0]

   for (int i = 1; i < n; ++i) {       // continuamos desde arr[1]
      if (arr[i] < res) {
         res = arr[i];
      }
   }

   return res;
}

int arr[3] = { 5, 8, 1 };
printf("%d", minimo_elemento(arr, 3));      // imprime 1
```

La observación de no existen arreglos al estilo C con cero elementos nos evita preocuparnos sobre qué valor devolver para este caso extremo.

El código anterior puede generalizarse para arreglos de elementos de tipo `T`, además de que el `if` puede eliminarse usando la función `std::min` de C++ que está en `algorithm`:

```
#include <algorithm>

template<typename T>
T minimo_elemento(T* arr, int n)
{
   T res = arr[0];    // iniciamos creyendo que el menor es arr[0]
```

```
   for (int i = 1; i < n; ++i) {       // continuamos desde arr[1]
      res = std::min(res, arr[i]);
   }

   return res;
}
```

La implementación anterior devuelve correctamente el valor del elemento más pequeño, pero si además necesitemos saber cuál es su posición dentro del arreglo (por ejemplo, por si necesitamos modificarlo) entonces la función no es de utilidad. Lo anterior puede corregirse devolviendo el índice del menor elemento en lugar de sólo su valor:

```
template<typename T>
int minimo_elemento(T* arr, int n)
{
   int res = 0;       // iniciamos creyendo que el menor es arr[0]

   for (int i = 1; i < n; ++i) {       // continuamos desde arr[1]
      if (arr[i] < arr[res]) {
         res = i;
      }
   }

   return res;
}

int arr[3] = { 7, 8, 5 };
printf("%d", minimo_elemento(arr, 3));  // imprime 2 (la posición)
printf("%d", arr[minimo_elemento(arr, 3)]); // imprime 5 (el valor)
```

¿Qué pasa si queremos calcular el elemento más pequeño de la segunda mitad del arreglo, en lugar de considerar el arreglo completo? Es posible usar la implementación anterior mandando como primer argumento el apuntador al inicio de la segunda mitad, en lugar del apuntador al inicio de todo el arreglo:

```
int arr[4] = { 7, 5, 6, 8 };
int pos = minimo_elemento(arr + 2, 2);
          // considerar dos elementos a partir de arr + 2: { 6, 8 }
```

Sin embargo, hay un detalle importante: la implementación anterior de `minimo_elemento` devuelve la posición del menor elemento relativa al apuntador `arr` recibido, por lo que es muy fácil utilizarla incorrectamente:

```
int arr[4] = { 7, 5, 6, 8 };
int pos = minimo_elemento(arr + 2, 2);      // considerar { 6, 8 }
printf("%d", pos);       // imprime 0 (la posición relativa)
printf("%d", arr[pos]);  // imprime 7, ¡incorrecto!
```

Si no queremos modificar la implementación de `minimo_elemento`, debemos recordar que la posición es relativa y entonces hacer el ajuste correspondiente al momento de utilizarla:

```
printf("%d", arr[2 + pos]);    // pos es 0 pero relativo a arr + 2
```

Para evitar tener que recordar este incómodo detalle, podemos modificar la implementación de `minimo_elemento` para que reciba el arreglo completo y adicionalmente el índice del primer elemento a considerar:

```
template<typename T>
int minimo_elemento(T* arr, int i, int n)
{
   int res = i;       // iniciamos creyendo que el menor es arr[i]

   for (int j = 1; j < n; ++j) {    // continuamos desde arr[i + 1]
      if (arr[i + j] < arr[res]) {       // j relativo a arr[i]
         res = i + j;
      }
   }

   return res;
}

int arr[4] = { 7, 5, 6, 8 };
int pos = minimo_elemento(arr, 2, 2); // considerar { ..., ..., 6, 8 }
printf("%d", pos);       // imprime 2 (la posición en arr)
printf("%d", arr[pos]);  // imprime 6, el menor de la segunda mitad
```

Aunque la complejidad de la implementación anterior es aún bastante manejable, podemos hacerlo mejor. Prácticamente todos problemas encontrados durante la implementación de este algoritmo surgieron por la utilización de índices relativos a una base: si la base cambia, entonces debemos recordar que los índices son relativos; en otro caso tenemos que mandar la base original como argumento de función además de los índices.

En este caso, los apuntadores tienen una ventaja: si un apuntador es válido, entonces sólo necesitamos desreferenciarlo sin necesidad (en general) de hacer ajustes. Podemos reimplementar `minimo_elemento` con apuntadores como se muestra a continuación, el cual es el mismo estilo que utiliza la biblioteca estándar de C++:

```
template<typename T>
int minimo_elemento(T* ini, T* fin)
           // el elemento apuntado por fin ya no debe considerarse
{
   T* res = ini;        // iniciamos creyendo que el menor es *ini

   while (ini != fin) {
      if (*ini < *res) {
         res = ini;
      }

      ++ini;
   }

   return res;
}

int arr[4] = { 7, 5, 6, 8 };
```

```
int* pos1 = minimo_elemento(arr, arr + 4);
printf("%d", *pos1);        // imprime el menor de todo el arreglo
int* pos2 = minimo_elemento(arr + 2, arr + 4);
printf("%d", *pos2);        // imprime el menor de la segunda mitad
```

Se asume que el apuntador al fin denota un elemento que en realidad no pertenece al arreglo y no debe procesarse, por lo que se dice que la pareja de apuntadores `ini`, `fin` denota una secuencia o intervalo de elementos semiabierto [`ini`, `fin`). Una consecuencia de esta decisión de diseño es que una secuencia vacía sí tiene una representación, la cual se da cuando `ini == fin`. En la implementación de `minimo_elemento`, se puede observar que cuando `ini == fin` entonces no se desreferencia ningún elemento y se devuelve el fin de manera simbólica.

El algoritmo `minimo_elemento` ya está implementado en C++ al incluir `algorithm`:

```
#include <algorithm>

int arr[4] = { 7, 5, 6, 8 };
int* pos1 = std::min_element(arr, arr + 4);
printf("%d", *pos1);        // imprime el menor de todo el arreglo
int* pos2 = std::min_element(arr + 2, arr + 4);
printf("%d", *pos2);        // imprime el menor de la segunda mitad
```

También se encuentra implementado el algoritmo `max_element`. De manera predeterminada ambos utilizan el operador < para calcular el mínimo o el máximo de la secuencia dada.

Implementación de ordenamiento por selección.

Ordenamiento por selección (*selection sort*) es un algoritmo de ordenamiento basado en comparaciones. Este algoritmo busca el elemento más pequeño de una secuencia y lo coloca en su lugar; esto se repite para el resto de los elementos que aún no están en su lugar. La típica implementación en C de este algoritmo para un orden ascendente es la siguiente:

```
void selection_sort(int* arr, int n)
{
   for (int i = 0; i < n; ++i) {
      int res = i;

      for (int j = i + 1; j < n; ++j) {
         if (arr[j] < arr[res]) {
            res = j;
         }
      }

      int temp = arr[res];
      arr[i] = arr[res];
      arr[res] = i;
   }
}

int arr[4] = { 4, 2, 6, 1 };
selection_sort(arr, 4);           // arr ahora es { 1, 2, 4, 6 }
```

Por supuesto, si queremos ordenar sólo una parte del arreglo utilizando una implementación basada en índices, nos enfrentaremos con algunos de los problemas que surgieron durante la implementación de `minimo_elemento`.

Ordenamiento por selección no está implementado en C++ pero puede implementarse una versión genérica de éste trivialmente con ayuda de la biblioteca de C++ (compare la implementación en términos de claridad y longitud con la implementación típica de C presentada arriba):

```
#include <algorithm>

template<typename T>
void selection_sort(T* ini, T* fin)
{
   while (ini != fin) {
      std::swap(*ini, *std::min_element(ini, fin));
      ++ini;
   }
}

int arr[4] = { 4, 2, 6, 1 };
selection_sort(arr, arr + 4);      // arr ahora es { 1, 2, 4, 6 }
```

C++ tiene implementada una rutina de ordenamiento llamada `sort` que ordena ascendentemente y que garantiza un tiempo de ejecución proporcional a ($n \log_2 n$) donde n es el tamaño de la secuencia a ordenar. Ordenamiento por selección toma un tiempo proporcional a n^2.

```
#include <algorithm>

int arr[4] = { 4, 2, 6, 1 };
std::sort(arr, arr + 4);           // arr ahora es { 1, 2, 4, 6 }
```

Criterios de orden personalizados

Al querer ordenar una secuencia, es común que un orden ascendente sea lo que se busca o bien, sea suficiente (`sort` de C++ produce un orden ascendente). Ahora supóngase que se desea ordenar descendentemente una secuencia de enteros. Una manera de obtener este orden es primero ordenar ascendentemente y luego invertir la secuencia:

```
#include <algorithm>

template<typename T>
void invierte(T* ini, T* fin)
{
   while (ini < fin) {
      --fin;
      std::swap(*ini, *fin);
      ++ini;

      // o bien, std::swap(*ini++, *--fin);
   }
}
```

```
int arr[4] = { 4, 2, 6, 1 };
std::sort(arr, arr + 4);           // arr ahora es { 1, 2, 4, 6 }
invierte(arr, arr + 4);            // arr ahora es { 6, 4, 2, 1 }
```

El algoritmo `invierte` tiene su contraparte en la biblioteca de C++ bajo el nombre `reverse`. Sin embargo, debemos notar que ordenar ascendentemente y luego invertir una secuencia es más lento que ordenarla de manera descendente desde el principio. ¿Cómo podemos lograr esto? Claramente implementar nuevas versiones de las rutinas de ordenamiento es una opción pero es una mala idea: si alguien quisiera ordenar una secuencia bajo una prioridad poco común, ni la versión ascendente ni la descendente con los operadores <, > servirán.

Lo que debemos entender es que el problema radica en que las rutinas de ordenamiento implementadas anteriormente son inflexibles: el criterio de orden está escrito directamente en el código (en el caso de un ordenamiento ascendente, el operador <). Lo que se necesita es permitir que el usuario especifique su propio criterio de ordenamiento. Ya que deseamos minimizar la cantidad de cambios que debamos hacer en la implementación de los algoritmos vistos, necesitamos que el criterio de ordenamiento sea:

- Un predicado binario (un predicado es una función que devuelve un valor booleano; en este caso es binario porque necesitamos que el predicado tome dos argumentos: `a`, `b`).
- Que el predicado se comporte como se comporta el operador **<**.

Este último punto necesita una explicación mucho más detallada. Lo que realmente necesitamos es que nuestro predicado `P` defina un orden débil estricto similar al operador <, es decir:

- Que `P(a, b)` devuelva verdadero si `a` debe ir antes que `b` (cuando `a` **<** `b` es verdadero en un orden ascendente entonces `a` va antes que `b`).
- Que `P(a, a)` sea falso.
- Si `P(a, b)` es verdadero entonces `P(b, a)` debe ser falso (no debe pasar que `a` **<** `b` y simultáneamente `b` **<** `a`).
- Si `P(a, b)` y `P(b, c)` son verdaderos entonces `P(a, c)` también debe serlo (haga la analogía con enteros para `a` < `b` y `b` < `c`).
- Si no existe un orden entre `a`, `b` y tampoco existe un orden entre `a`, `c` entonces no existe un orden entre `a`, `c`. Se dice que no existe un orden entre `a`, `b` si `P(a, b)` es falso y `P(b, a)` también lo es.

Cualquier orden que cumpla las propiedades mencionadas anteriormente puede usarse en los algoritmos de la biblioteca de C++ que aceptan un predicado de orden. En la práctica, la propiedad más importante a recordar es la primera: el predicado debe devolver verdadero si `a` va antes que `b`.

A manera de ejemplo, ordenaremos descendentemente una secuencia de enteros con ayuda de un predicado sobre dos elementos `a`, `b` que devuelva verdadero cuando `a` sea mayor que `b`. Usaremos la versión de `sort` de C++ que además de tomar el inicio y el fin de la secuencia, toma el predicado a utilizar durante el ordenamiento.

```
bool primero_mayor(int a, int b)
   // debe devolver verdadero si a va antes que b
{
   return a > b;
}
```

```
int arr[4] = { 4, 2, 6, 1 };
std::sort(arr, arr + 4, primero_mayor);  // arr queda { 6, 4, 2, 1 }
```

Difícilmente un alumno que apenas ha cursado la UEA "Programación Estructurada" sabrá cómo es posible enviar funciones como argumentos a otra función. La respuesta es que el lenguaje C (y obviamente también C++) incluyen el concepto de apuntador a función. El problema de usar apuntadores a función es lo confuso que puede resultar la sintaxis de su declaración:

```
bool menor_que(int a, int b)
{
   return a < b;
}

bool mayor_que(int a, int b)
{
   return a > b;
}

bool(*p)(int, int);  // p es un apuntador a función bool(int, int)
p = menor_que;               // p apunta a menor_que
printf("%d\n", p(1, 2));     // imprime 1
p = mayor_que;               // p apunta a mayor_que
printf("%d\n", p(1, 2));     // imprime 0
```

El tipo de un apuntador a función debe coincidir perfectamente con la firma de la función a apuntar:

```
void f(int&);            // función que toma int& y devuelve void
void(*p)(int) = f;           // error, f toma int por referencia
```

Una manera sencilla de evitar tener que escribir el tipo (exacto) de un apuntador a función usado como parámetro es usar un `template`:

```
template<typename T>
void f(T apuntador);

bool menor_que(int a, int b)
{
   return a < b;
}

f(menor_que);       // llama a ejemplo(T) con T = bool(*)(int, int)
```

Además de `sort`, también la función `min_element` de C++ tiene una versión que toma un predicado binario para determinar el elemento más pequeño de la secuencia bajo el orden inducido por el predicado. El análogo de nuestra función `minimo_elemento` es la siguiente:

```
template<typename T, typename F>
int minimo_elemento(T* ini, T* fin, F pred)
// pred es un predicado binario que induce un orden débil estricto
{
   T* res = ini;

   while (ini != fin) {
```

```
      if (pred(*ini, *res)) {
         res = ini;
      }

      ++ini;
   }

   return res;
}
```

Nuestra función de ordenamiento por selección se vería como sigue:

```
template<typename T, typenamed F>
void selection_sort(T* ini, T* fin, F pred)
{
   while (ini != fin) {
      std::swap(*ini, *std::min_element(ini, fin, pred));
      ++ini;
   }
}
```

Nótese que el único cambio en la implementación de `minimo_elemento` es el reemplazo del operador < por una llamada al predicado `pred`. Más aún, nótese que a `selection_sort` le basta pasar el predicado recibido a `min_element`; el resto del algoritmo no sufrió cambios.

Búsqueda lineal

Es frecuente necesitar saber si cierto valor está o no en una secuencia de elementos. El algoritmo de búsqueda lineal simplemente consiste en examinar todos los elementos de la secuencia. En general, búsqueda lineal es lo más rápido que puede hacerse sin conocer alguna propiedad adicional de la secuencia. Una implementación de este algoritmo es la siguiente:

```
template<typename T>
T* encuentra(T* ini, T* fin, const T& valor)
{
   while (ini != fin && !(*ini == valor)) {
      ++ini;
   }

   return ini;
}
```

Cuando el elemento no se encuentra en la secuencia, el algoritmo devuelve el fin de manera simbólica; si el valor buscado aparece más de una vez en la secuencia entonces se devuelve el primero encontrado. El algoritmo llamado `encuentra_si` es muy similar, excepto que encuentra un valor que cumpla un predicado unario en lugar de comparar directamente contra un valor dado:

```
template<typename T, typename F>
T* encuentra_si(T* ini, T* fin, F pred)
{
   while (ini != fin && !pred(*ini)) {
```

```
      ++ini;
   }

   return ini;
}

bool es_par(int n)
{
   return n % 2 == 0;
}

int arr[4] = { 5, 7, 2, 3 };
int* pos = encuentra_si(arr, arr + 4, es_par);

if (pos == arr + 4) {         // si no hay pares se devuelve el fin
   printf("no hay pares");
}
else {
   printf("%d", *pos);        // imprime 2
}
```

Ambos algoritmos se encuentran implementados en `algorithm` de C++ bajo los nombres `find` y `find_if`.

Búsqueda binaria

Cuando es necesario saber si varios valores están o no en una secuencia de elementos, entonces usar búsqueda lineal para cada uno de ellos es mala idea (necesitaríamos recorrer varias veces la secuencia o bien hacer algo equivalente). Cuando anticipamos esta necesidad entonces es mejor ordenar previamente la secuencia y después utilizar búsqueda binaria.

El algoritmo de búsqueda binaria es esencialmente lo que hacemos sin darnos cuenta al buscar una palabra en un diccionario: abrimos el diccionario aproximadamente a la mitad y si no tuvimos la fortuna de encontrar la palabra justo donde abrimos, entonces podemos descartar una de las dos mitades del diccionario dependiendo si la palabra debiera estar en la primera o en la segunda mitad. Una implementación de búsqueda binaria es la siguiente:

```
template<typename T>
bool busqueda_binaria(T* ini, T* fin, const T& valor)
   // utiliza el orden inducido por el operador <
{
   if (ini == fin) {             // secuencia vacía
      return false;
   }

   T* mitad = ini + (fin - ini) / 2;

   if (valor < *mitad) {         // valor debe ir antes de *mitad
      return busqueda_binaria(ini, mitad, valor);
   }
   else if (*mitad < valor) {    // valor debe ir después de *mitad
      return busqueda_binaria(mitad + 1, fin, valor);
```

```
   }

   return true;
}

int arr[4] = { 2, 3, 5, 7 };     // ordenado ascendentemente bajo <
printf("%d\n", busqueda_binaria(arr, arr + 4, 5));     // imprime 1
printf("%d\n", busqueda_binaria(arr, arr + 4, 9));     // imprime 0
```

El algoritmo anterior está implementado en `algorithm` bajo el nombre `binary_search` y también existe la versión que recibe un predicado que induce un orden compatible con el de la secuencia. Búsqueda binaria tiene un tiempo ejecución proporcional a $\log_2 n$ donde `n` es la longitud de la secuencia dada.

Desafortunadamente la función devuelve un `bool`. Si fuera necesario conocer la posición del elemento entonces dicha función no será de utilidad. Más aún, este algoritmo calcula (y después ignora) cuál es la posición que debería tener un elemento que actualmente no está en la secuencia.

Para ubicar la posición que debería tener un elemento (independientemente si está o no está) podemos usar un algoritmo llamado `cota_inferior`. Este algoritmo encuentra la posición del primer elemento no menor al valor dado:

```
template<typename T>
T* cota_inferior(T* ini, T* fin, const T& valor)
   // utiliza el orden inducido por el operador <
{
   if (fin - ini <= 2) {      // caso base
      while (ini != fin && *ini < valor) {
         ++ini;
      }

      return ini;
   }

   T* mitad = ini + (fin - ini) / 2;

   if (*mitad < valor) {   // de mitad para atrás todos son menores
      return cota_inferior(mitad + 1, fin, valor);
   }
   else {
      return cota_inferior(ini, mitad, valor);
   }
}
```

Este algoritmo está implementado en `algorithm` como `lower_bound`. También está implementado el algoritmo `upper_bound` el cual regresa el primer elemento mayor al valor dado. Un ejemplo que ilustra ambos algoritmos es el siguiente:

```
#include <algorithm>

int arr[4] = { 2, 3, 5, 6, 6, 8, 9 };
int* inferior = std::lower_bound(arr, arr + 4, 6);
```

```
     // apunta al primer elemento no menor que 6, que es 6
int* superior = std::upper_bound(arr, arr + 4, 6);
     // apunta al primer elemento mayor que 6, que es 8
printf("en la secuencia hay %d números 6", superior - inferior);
     // { 2, 3, 5, 6, 6, 8, 9 }
     //            ^     ^     superior - inferior == 2
```

El algoritmo `equal_range` de `algorithm` calcula la cota inferior y superior simultáneamente, por lo que es más rápido que calcular las dos por separado.

Objetos función y funciones de orden superior

El caso base de la función `cota_superior` mostrada anteriormente fue implementado como sigue:

```
template<typename T>
T* cota_inferior(T* ini, T* fin, const T& valor)
{
   if (fin - ini <= 2) {
      while (ini != fin && *ini < valor) {
         ++ini;
      }

      return ini;
   }

   //...
}
```

Es fácil ver el ciclo `while` corresponde con la búsqueda de un elemento que cumple un predicado (ser no menor que `valor`); es decir, conceptualmente realiza la misma tarea que la función `encuentra_si` o `find_if`. Sin embargo, el predicado que el elemento debe cumplir depende del parámetro `valor`. ¿Cómo podemos construir un predicado que dependa del valor de una variable local?

C++ permite definir qué significa el operador de invocación a función `(...)` para un objeto de un tipo definido por el usuario (es decir, una variable que sea un `struct`). De manera predeterminada, usar una variable que es un `struct` como si fuera una función es un error:

```
struct mi_tipo {
   int n;
};

mi_tipo t;
t.n = 123;
t( );          // error, t no es una función
```

Podemos definir el operador de invocación a función usando la siguiente sintaxis:

```
struct mi_tipo {
   int n;
```

```
   // código a ejecutar si la variable es usada como función
   // sin argumentos
   void operator()( )
   {
      printf("%d", n);
   }
};

mi_tipo t;
t.n = 123;
t( );                          // imprime 123
```

Es posible enviar y recibir valores como si se tratara de una función ordinaria, pero adicionalmente teniendo acceso a los valores que están almacenados dentro de la variable:

```
struct mi_tipo {
   int n;

   // código a ejecutar si la variable es usada como función
   // con dos ints como argumentos; devuelve un int
   int operator()(int a, int b)
   {
      return a + b + n;
   }
};

mi_tipo t;
t.n = 123;
printf("%d", t(1, 1));         // imprime 125
```

Un objeto que puede comportarse como una función se denomina un objeto función. Con la idea anterior, podemos implementar un predicado con estado de la siguiente manera:

```
struct menor_que_k {
   int k;

   bool operator()(int v)
   {
      return v < k;
   }
};

menor_que_k pred;
pred.k = 10;
printf("%d", pred(5));         // imprime 1
printf("%d", pred(20));        // imprime 0
```

Podemos generalizar el `struct` anterior para poder almacenar (y comparar) cosas que no son necesariamente `int`s. Esto se hace mediante `template`s:

```
template<typename T>
struct menor_que_k {
   T k;                              // el estado es de tipo T
```

```
   bool operator()(const T& v)     // recibe una variable de tipo T
   {
      return v < k;
   }
};

menor_que_k<double> pred;     // T = double
pred.k = 10.5;
printf("%d", pred(5.2));      // imprime 1
printf("%d", pred(20.8));     // imprime 0
```

Finalmente implementaremos un objeto función que almacena otro predicado y que al ser invocado, llama al predicado guardado y devuelve la negación de su resultado:

```
template<typename T, typename F>
struct negacion_unaria {
   F p;                       // el estado es un predicado de tipo F

   bool operator()(const T& v)     // recibe una variable de tipo T
   {
      return !p(v);           // invoca a p y niega el resultado
   }
};

menor_que_k<double> pred;     // T = double
pred.k = 10.5;
negacion_unaria<double, menor_que_k<double>> pred_negado;
pred_negado.p = pred;

printf("%d", pred_negado(5.2));    // imprime 0
printf("%d", pred_negado(20.8));   // imprime 1
```

Una función que toma una función y construye otra se denomina función de orden superior. Nuestro `struct` llamado `negacion_unaria` es una manera de lograr lo anterior en C++. Con esto podemos eliminar el ciclo del caso base de `cota_inferior`:

```
#include <algorithm>

template<typename T>
T* cota_inferior(T* ini, T* fin, const T& valor)
{
   if (fin - ini <= 2) {
      menor_que_k<T> pred;
      pred.k = valor;
      negacion_unaria<T, menor_que_k<T>> pred_negado;
      pred_negado.p = pred;

      return std::find_if(ini, fin, pred_negado);
   }

   //...
}
```

Lo anterior puede parecer peor que el ciclo `while` y lo es. Sin embargo (y aunque no entraremos en detalles) esto puede ser simplificado con la ayuda de la biblioteca de C++ de la siguiente manera:

```
#include <algorithm>
#include <functional>

template<typename T>
T* cota_inferior(T* ini, T* fin, const T& valor)
{
   if (fin - ini <= 2) {
      return std::find_if(ini, fin,
         std::not1(std::bind2nd(std::less<T>( ), valor))
      );
   }

   //...
}
```

Implementación de otros algoritmos de ordenamiento (ejercicio)

El algoritmo de ordenamiento por inserción (*insertion sort*) es un algoritmo que mantiene una secuencia ordenada a cada paso y va insertando un nuevo elemento a la vez. Con los algoritmos de la biblioteca estándar es posible implementarlo en pocas líneas de código:

```
#include <algorithm>

template<typename T>
void insertion_sort(T* ini, T* fin)
{
   for (T* actual = ini; actual != fin; ++actual) {
      T valor = *actual;
      *std::copy_backward(
         std::lower_bound(ini, actual, valor),
         actual,
         actual + 1
      ) = valor;
   }
}
```

Investigue el tiempo de ejecución de ordenamiento por inserción, cómo funciona el algoritmo `copy_backward` y por qué es necesario usar esta función en lugar de la función `copy`.

El algoritmo de ordenamiento rápido (*quick sort*) es un algoritmo de ordenamiento que parte una secuencia bajo un predicado unario y posteriormente ordena recursivamente las dos subsecuencias resultantes de la partición. Con los algoritmos de la biblioteca estándar es posible implementarlo en pocas líneas de código. Implemente el algoritmo de ordenamiento rápido e investigue su tiempo de ejecución.