ALGORITMOS Y ESTRUCTURA DE DATOS

Operaciones sobre arrays

Antes de comenzar

Este documento resume las principales operaciones que son generalmente utilizadas para la manipulación de *arrays*. Además busca inducir al alumno para que descubra la necesidad de trabajar con tipos de datos genéricos, implementados con *templates*, y también la importancia de poder desacoplar las porciones de código que son propias de un problema, de modo tal que el algoritmo pueda ser genérico e independiente de cualquier situación particular; delegando dichas tareas en la invocación de funciones que se reciben cómo parámetros (punteros a funciones).

Autor: Ing. Pablo Augusto Sznajdleder.

Revisores: Ing. Analía Mora, Martín Montenegro.

Agregar un elemento al final de un array

La siguiente función agrega el valor `v` al final del *array* `arr`, incrementa su longitud `len` y retorna su posición.

```
int add(int arr[], int& len, int v)
{
   arr[len]=v;
   len++;
   return len-1; // retorna la posicion del elemento que agrego
}
```

Recorrer y mostrar el contenido de un array

La siguiente función recorre el *array* `arr` mostrando por consola el valor de cada uno de sus elementos.

```
void mostrar(int arr[], int len)
{
   for(int i=0; i<len; i++)
   {
      cout << arr[i] << endl;
   }
   return;
}
```

Determinar si un array contiene o no un determinado valor

La siguiente función permite determinar si el *array* `arr` contiene o no al elemento `v`; retorna la posición que `v` ocupa dentro de `arr` o un valor negativo si `arr` no contiene a `v`.

```
int find(int arr[], int len, int v)
{
   int i=0;
   while( i<len && arr[i]!=v )
   {
      i++;
   }
   return i<len?i:-1; // retorna la posicion de v o -1 si v no existe en arr
}
```

Eliminar el valor que se ubica en una determinada posición del array

La siguiente función elimina el valor que se encuentra en la posición `pos` del *array* `arr`, desplazando al *i*-ésimo elemento hacia la posición *i*-1, para todo valor de *i*>`pos` y *i*<`len`.

```
void remove(int arr[], int& len, int pos)
{
   for(int i=pos; i<len-1; i++ )
   {
      arr[i]=arr[i+1];
   }
   // decremento la longitud del array
   len--;

   return;
}
```

Insertar un valor en una determinada posición del array

La siguiente función inserta el valor `v` en la posición `pos` del *array* `arr`, desplazando al *i*-ésimo elemento hacia la posición *i*+1, para todo valor de *i* que verifique: *i*>=`pos` e *i*<`len`.

```
void insert(int arr[], int& len, int v, int pos)
{
   for(int i=len-1; i>=pos; i--)
   {
      arr[i+1]=arr[i];
   }
   // inserto el elemento e incremento la longitud del array
   arr[pos]=v;
   len++;
   return;
}
```

Insertar un valor respetando el orden del array

La siguiente función inserta el valor `v` en el *array* `arr`, en la posición que corresponda según el criterio de precedencia de los números enteros. El *array* debe estar ordenado o vacío.

```
int orderedInsert(int arr[], int& len, int v)
{
   int i=0;
```

```
   // mientras no me pase de largo y mientras no encuentre lo que busco...
   while( i<len && arr[i]<=v )
   {
      i++;
   }
   // inserto el elemento en la i-esima posicion del array
   insert(arr,len,v,i); // invoco a la funcion insert
   return i; // retorna la posicion en que se inserto al elemento
}
```

Más adelante veremos como independizar el criterio de precedencia para lograr que la misma función sea capáz de insertar un valor respetando un criterio de precedencia diferente entre una y otra invocación.

Insetar un valor respetando el orden del array, sólo si aún no lo contiene

La siguiente función busca el valor `v` en el *array* `arr`; si lo encuentra entonces asigna `true` a `enc` y retorna la posición que `v` ocupa dentro de `arr`. De lo contrario asigna `false` a `enc`, inserta a `v` en `arr` respetando el orden de los números enteros y retorna la posición en la que finalmente `v` quedó ubicado.

```
int searchAndInsert(int arr[], int& len, int v, bool& enc)
{
   // busco el valor
   int pos = find(arr,len,v);
   // determino si lo encontre o no
   enc = pos>=0;
   // si no lo encontre entonces lo inserto ordenado
   if( !enc )
   {
      pos = orderedInsert(arr,len,v);
   }
   return pos;
}
```

Templates

Los *templates* permiten parametrizar los tipos de datos con los que trabajan las funciones, generando de este modo verdaderas funciones genéricas.

Generalización de las funciones add y mostrar

```
template <typename T> int add(T arr[], int& len, T v)
{
   arr[len]=v;
   len++;
   return len-1;
}
template <typename T> void mostrar(T arr[], int len)
{
   for(int i=0; i<len; i++)
   {
      cout << arr[i];
      cout << endl;
   }

   return;
}
```

Veamos como invocar a estas funciones genéricas.

```
int main()
{
   // declaro un array de cadenas y su correspondiente longitud
   string aStr[10];
   int lens=0;

   // trabajo con el array de cadenas
   add<string>(aStr,lens,"uno");
   add<string>(aStr,lens,"dos");
   add<string>(aStr,lens,"tres");

   // muestro el contenido del array
   mostrar<string>(aStr,lens);

   // declaro un array de enteros y su correspondiente longitud
   int aInt[10];
   int leni =0;

   // trabajo con el array de enteros
   add<int>(aInt,leni,1);
   add<int>(aInt,leni,2);
   add<int>(aInt,leni,3);

   // muestro el contenido del array
   mostrar<int>(aInt,leni);

   return 0;
}
```

Ordenamiento

La siguiente función ordena el *array* `arr` de tipo `T` siempre y cuando dicho tipo especifique el criterio de precedencia de sus elementos mediante los operadores relacionales `>` y `<`. Algunos tipos (y/o clases) válidos son: `int`, `long`, `short`, `float`, `double`, `char` y `string`.

```
template <typename T> void sort(T arr[], int len)
{
   bool ordenado=false;
   while(!ordenado)
   {
      ordenado = true;
      for(int i=0; i<len-1; i++)
      {
         if( arr[i]>arr[i+1] )
         {
            T aux = arr[i];
            arr[i] = arr[i+1];
            arr[i+1] = aux;
            ordenado = false;
         }
      }
   }

   return;
}
```

Punteros a funciones

Las funciones pueden ser pasadas como parámetros a otras funciones para que éstas las invoquen.

Utilizaremos esta carácteristica de los lenguajes de programación para parametrizar el criterio de precedencia que queremos que la función `sort` aplique al momento de comparar cada par de elementos del *array* `arr`.

Observemos con atención el tercer parámetro que recibe la función `sort`. Corresponde a una función que retorna un valor de tipo `int` y recibe dos parámetros de tipo `T`, siendo `T` un tipo de datos genérico parametrizado por el *template*.

La función `criterio`, que debemos desarrollar por separado, debe comparar dos elementos *e1* y *e2*, ambos de tipo `T`, y retornar un valor: negativo, positivo o cero según se sea: *e1<e2*, *e1>e2* o *e1=e2* respectivamente.

```
template <typename T> void sort(T arr[], int len, int criterio(T,T))
{
   bool ordenado=false;
   while(!ordenado)
   {
      ordenado=true;
      for(int i=0; i<len-1; i++)
      {
         // invocamos a la funcion para ver si corresponde o no permutar
         if( criterio(arr[i],arr[i+1])>0 )
         {
            T aux = arr[i];
            arr[i] = arr[i+1];
            arr[i+1] = aux;
            ordenado = false;
         }
      }
   }
   return;
}
```

Ordenar arrays de diferentes tipos de datos con diferentes criterios de ordenamiento

A continuación analizaremos algunas funciones que comparan pares de valores (ambos del mismo tipo) y determinan cual de esos valores debe preceder al otro.

Comparar cadenas, criterio alfabético ascendente

```
int criterioAZ(string e1, string e2)
{
   return e1>e2?1:e1<e2?-1:0;
}
```

Comparar cadenas, criterio alfabético descendente

```
int criterioZA(string e1, string e2)
{
   return e2>e1?1:e2<e1?-1:0;
}
```

Comparar enteros, criterio numérico ascendente

```
int criterio09(int e1, int e2)
{
   return e1-e2;
}
```

Comparar enteros, criterio numérico descendente

```
int criterio90(int e1, int e2)
{
   return e2-e1;
}
```

Probamos lo anterior:

```
int main()
{
   int len = 6;

   // un array con 6 cadenas
   string x[] = {"Pablo", "Pedro", "Andres", "Juan", "Zamuel", "Oronio"};

   // ordeno ascendentemente pasando como parametro la funcion criterioAZ
   sort<string>(x,len,criterioAZ);
   mostrar<string>(x,len);
```

```
   // ordeno descendentemente pasando como parametro la funcion criterioZA
   sort<string>(x,len,criterioZA);
   mostrar<string>(x,len);

   // un array con 6 enteros
   int y[] = {4, 1, 7, 2, 8, 3};

   // ordeno ascendentemente pasando como parametro la funcion criterio09
   sort<int>(y,len,criterio09);
   mostrar<int>(y,len);

   // ordeno ascendentemente pasando como parametro la funcion criterio90
   sort<int>(y,len,criterio90);
   mostrar<int>(y,len);

   return 0;
}
```

Arrays de estructuras

Trabajaremos con la siguiente estructura:

```
struct Alumno
{
   int legajo;
   string nombre;
   int nota;
};

// esta funcion nos permitira "crear alumnos" facilmente
Alumno crearAlumno(int le, string nom, int nota)
{
   Alumno a;
   a.legajo = le;
   a.nombre = nom;
   a.nota = nota;
   return a;
}
```

Usando arrays de estructuras

```
int main()
{
   Alumno arr[6];
   int len=0;

   add<Alumno>(arr,len,crearAlumno(30,"Juan",5));
   add<Alumno>(arr,len,crearAlumno(10,"Pedro",8));
   add<Alumno>(arr,len,crearAlumno(20,"Carlos",7));
   add<Alumno>(arr,len,crearAlumno(60,"Pedro",10));
   add<Alumno>(arr,len,crearAlumno(40,"Alberto",2));
   add<Alumno>(arr,len,crearAlumno(50,"Carlos",4));

   for(int i=0; i<len; i++)
   {
      cout<<arr[i].legajo<<", "<<arr[i].nombre<<", "<<arr[i].nota<<endl;
   }

   return 0;
}
```

Pregunta: ¿Por qué no utilizamos la función `mostrar` para mostrar el contenido del *array* de alumnos?

Ordenar arrays de estructuras, por diferentes criterios

Recordemos la función `sort`:

```
template <typename T> void sort(T arr[], int len, int criterio(T,T))
{
   bool ordenado=false;
   while(!ordenado)
   {
      ordenado=true;
      for(int i=0; i<len-1; i++)
      {
         if( criterio(arr[i],arr[i+1])>0 )
         {
            T aux = arr[i];
            arr[i] = arr[i+1];
            arr[i+1] = aux;
            ordenado = false;
         }
      }
   }
   return;
}
```

Definimos diferentes criterios de precedencia de alumnos:

`a1` precede a `a2` si `a1.legajo<a2.legajo`:

```
int criterioAlumnoLegajo(Alumno a1, Alumno a2)
{
   return a1.legajo-a2.legajo;
}
```

`a1` precede a `a2` si `a1.nombre<a2.nombre`:

```
int criterioAlumnoNombre(Alumno a1, Alumno a2)
{
   return a1.nombre<a2.nombre?-1:a1.nombre>a2.nombre?1:0;
}
```

`a1` precede a `a2` si `a1.nombre<a2.nombre`. A igualdad de nombres entonces precederá el alumno que tenga menor número de legajo:

```
int criterioAlumnoNomYLeg(Alumno a1, Alumno a2)
{
   if( a1.nombre == a2.nombre )
   {
      return a1.legajo-a2.legajo;
   }
   else
   {
      return a1.nombre<a2.nombre?-1:a1.nombre>a2.nombre?1:0;
   }
}
```

Ahora sí, probemos los criterios anteriores con la función `sort`.

```
int main()
{
   Alumno arr[6];
   int len=0;

   add<Alumno>(arr,len,crearAlumno(30,"Juan",5));
   add<Alumno>(arr,len,crearAlumno(10,"Pedro",8));
   add<Alumno>(arr,len,crearAlumno(20,"Carlos",7));
   add<Alumno>(arr,len,crearAlumno(60,"Pedro",10));
   add<Alumno>(arr,len,crearAlumno(40,"Alberto",2));
   add<Alumno>(arr,len,crearAlumno(50,"Carlos",4));

   // ordeno por legajo
   sort<Alumno>(arr,len,criterioAlumnoLegajo);
   // recorrer y mostrar el contenido del array...

   // ordeno por nombre
   sort<Alumno>(arr,len,criterioAlumnoNombre);
   // recorrer y mostrar el contenido del array...

   // ordeno por nombre+legajo
   sort<Alumno>(arr,len,criterioAlumnoNomYLeg);
   // recorrer y mostrar el contenido del array...

   return 0;
}
```

Resumen de plantillas

Función `add`.

Descripción: Agrega el valor `v` al final del *array* `arr`, incrementa su longitud y retorna la posición

```
template <typename T>
int add(T arr[], int& len, T v)
{
   arr[len]=v;
   len++;
   return len-1;
}
```

Función `find`.

Descripción: Busca la primer ocurrencia de `v` en `arr`; retorna su posición o un valor negativo si `arr` no contiene a `v`.

```
template <typename T, typename K>
int find(T arr[], int len, K v, int criterio(T,K))
{
   int i=0;
   while( i<len && criterio(arr[i],v)!=0 )
   {
      i++;
   }

   return i<len?i:-1;
}
```

Función `remove`.

Descripción: Elimina el valor ubicado en la posición `pos` del *array* `arr`, decrementando su longitud.

```
template <typename T>
void remove(T arr[], int& len, int pos)
{
   int i=0;
   for(int i=pos; i<len-1; i++ )
   {
      arr[i]=arr[i+1];
   }

   len--;
   return;
}
```

Función `insert`.

Descripción: Inserta el valor `v` en la posición `pos` del *array* `arr`, incrementando su longitud.

```
template <typename T>
void insert(T arr[], int& len, T v, int pos)
{
   for(int i=len-1; i>=pos; i--)
   {
      arr[i+1]=arr[i];
   }

   arr[pos]=v;
   len++;
   return;
}
```

Función `orderedInsert`.

Descripción: Inserta el valor `v` en el *array* `arr` en la posición que corresponda según el criterio `criterio`.

```
template <typename T>
int orderedInsert(T arr[], int& len, T v, int criterio(T,T))
{
   int i=0;
   while( i<len && criterio(arr[i],v)<=0 )
   {
      i++;
   }

   insert<T>(arr,len,v,i);

   return i;
}
```

Función `searchAndInsert`.

Descripción: Busca el valor `v` en el *array* `arr`; si lo encuentra entonces retorna su posición y asigna `true` al parámetro `enc`. De lo contrario lo inserta donde corresponda según el criterio `criterio`, asigna `false` al parámetro `enc` y retorna la posición en donde finalmente quedó ubicado el nuevo valor.

```
template <typename T>
int searchAndInsert(T arr[], int& len, T v, bool& enc, int criterio(T,T))
{
   // busco el valor
   int pos = find<T,T>(arr,len,v,criterio);

   // determino si lo encontre o no
   enc = pos>=0;

   // si no lo encontre entonces lo inserto ordenado
   if( !enc )
   {
      pos = orderedInsert<T>(arr,len,v,criterio);
   }

   return pos;
}
```

Función `sort`.

Descripción: Ordena el *array* `arr` según el criterio de precedencia que indica la función `criterio`.

```
template <typename T>
void sort(T arr[], int len, int criterio(T,T))
{
   bool ordenado=false;
   while(!ordenado)
   {
      ordenado=true;
      for(int i=0; i<len-1; i++)
      {
         if( criterio(arr[i],arr[i+1])>0 )
         {
            T aux = arr[i];
            arr[i] = arr[i+1];
            arr[i+1] = aux;
            ordenado = false;
         }
      }
   }

   return;
}
```

Búsqueda binaria

Función `binarySearch`.

Descripción: Busca el elemento `v` en el *array* `arr` que debe estar ordenado según el criterio `criterio`. Retorna la posición en donde se encuentra el elemento o donde este debería ser insertado.

```
template<typename T, typename K>
int binarySearch(T a[], int len, K v, int criterio(T,K), bool& enc)
{
   int i=0;
   int j=len-1;
   int k=(i+j)/2;
```

```
   enc=false;
   while( !enc && i<=j )
   {
      if( criterio(a[k],v)>0 )
      {
         j=k-1;
      }
      else
      {
         if( criterio(a[k],v)<0 )
         {
            i=k+1;
         }
         else
         {
            enc=true;
         }
      }
      k=(i+j)/2;
   }
   return criterio(a[k],v)>=0?k:k+1;
}
```