Estructura de Datos y Algoritmos

Concepto y manipulación de punteros

1. Apuntadores

Un apuntador es una variable que contiene la dirección de otra variable.

Los apuntadores se utilizan mucho en C, en parte debido a que ellos son en ocasiones la única forma de expresar una operación y en parte debido a que por lo general llevan un codigo más compacto y eficiente de lo que se puede obtener en otras formas.

Los apuntadores y los arreglos están relacionados intimamente, veremos dichas relaciones y como explotarlas.

1.1. Apuntadores y direcciones

Empecemos viendo como se orgamiza la memoria. Una computadora típica tiene un arreglo de celdas de memoria numeradas o direccionadas consecutivamente, que pueden manipularse individualmente o en grupos contiguos. Una situación comun es que cualquier byte puede ser un char, un par de celdas de un byte puede tratarse como un entero short y cuatro bytes adjacentes forman un long.

Un apuntador es un grupo de celdas (generalmente dos o cuatro) que pueden mantener una dirección. Si c es un char y p un apuntador a char, podemos realizar las siguientes operaciones:

p = & c;

que coloca la dirección de c en p (el & es llamado *operador de dirección*) ,

*p = 'a';

que coloca 'a' en la variable apuntada por p, en este caso c (el * es conocido como *operador de indirección* o *operador de desreferenciación*).

Los apuntadores como cualquier otra variable debe ser declarado antes de ser usado. Los apuntadores pueden declararse para que apunten a objetos de cualquier tipo de datos. Declaramos apuntadores del siguiente modo:

```
int *iptr;
char *cptr;
```

declara iptr como puntero a int y cptr como puntero a char.

Los apuntadores se deben inicializar, ya sea al declararlos o mediante una asignación. Un apuntador puede inicializarse a 0, a NULL o a una dirección. Un apuntador con valor 0 o NULL no apunta a nada. NULL es una constante

definida en <iostream.h>. NULL y 0 son equivalentes. Apuntadores que valen 0 o NULL no pueden ser desreferenciados.

Podemos realizar las siguientes operaciones:

```
int x=1,y=2,z[10];
int *ip; /* ip es un apuntador a int */

ip = &x; /* ip apunta a x */
y = *ip; /* y es ahora 1 */
*ip = 0; /* x es ahora 0 */
ip = &z[0]; /* ip ahora apunta a z[0]*/
```

Si ip apunta al entero x entonces *ip puede presentarse en cualquier contexto donde x puede hacerlo, por ejemplo

```
*ip = *ip +10
```

incrementa *ip en 10.

Los operadores unarios * y & ligan más estrechamente que los operadores aritmeticos, entonces:

```
y = *ip + 1;
```

toma aquello a lo que apunta ip, le suma 1 y asigna el resultado a y, mientras que:

```
*ip + = 1;
```

incrementa en uno aquello a lo que ip apunta. Los operadores ++ y - - son unarios igual que *. Estos se asocian de derecha a izquierda. Luego:

```
++*ip; /* suma 1 a lo que apunta ip */

(*ip)++; /* idem. Los parentesis son necesarios */
```

Como los apuntadores son variables, se pueden emplear sin desreferenciamento. Por ejemplo si iq es un puntero a int, puedo asignar con:

```
iq=ip; /* copia el contenido de ip a iq*/
```

luego de la asignación iq apunta a lo mismo que ip.

La referencia a un valor a través de un apuntador se conoce como **indirección**.

1.2. Apuntadores y argumentos de funciones

Consideremos algoritmos de ordenación sobre arrays, es usual que tengan la operación de intercambio de elementos en el array. No es suficiente con escribir intercambio(a,b) donde intercambio está definida como:

```
void intercambio(int x, int y)
{
    int temp;

    temp=x;
    x=y;
    y=temp;
}
```

el problema es que las variables se pasan por valor, luego la función no modifica a y b que están en el programa que invoco la función. La función intercambio, solo intercambia copias de a y de b (que se asocian a x e y).

Para que intercambio funcione debemos pasar los argumentos **por referencia**, esto es x e y deben ser punteros. La llamada se realiza en la forma:

```
intercambio(&a,&b)
```

y la funcion intercambio se define en la forma:

```
void intercambio(int *x, int *y)
{
    int temp;

    temp=*x;
    *x=*y;
    *y=temp;
}
```

Los argumentos tipo apuntador permiten a una función tener acceso y cambiar objetos que están en la función invocadora.

Podemos tambien devolver apuntadores, por ejemplo declarando una función como:

```
int * funcion1 (int *)
```

indica que el argumento de funcion1 es un apuntador a int y que retorna un apuntador a int.

1.2.1. Formas de llamados de funciones por referencia

Hay dos formas de pasar argumentos por referencia: *llamado por referencia con argumentos de referencia* y *llamado por referencia con argumentos de apuntador*.

Los apuntadores pueden servir para pasar apuntadores a objetos de datos grandes, evitando la sobrecarga de pasar los objetos mediante llamadas por valor.

En el caso de arrays, el nombre de estos es la localidad inicial del mismo en memoria (el nombre del arreglo es equivalente a &nombrearreglo[0]).

Veremos los parámetros de referencia que son otra forma de pasar argumentos por referencia.

Para indicar que el parámetro de una función se pasa por referencia colocamos un & despues del tipo del parámetro en el prototipo de la función. Utilizamos la misma convención en el tipo del encabezado de la función. Veamos el ejemplo de intercambio, la declaración de la función sería:

```
void intercambio(int &x, int &y)
{
    int temp;

    temp=x;
    x=y;
    y=temp;
}
```

en la llamada de la función, mencionamos la variable por nombre y esta será pasada por referencia, por ejemplo:

```
intercambio(a,b)
```

En el cuerpo de la función llamada, la variable es mencionada por su nombre de parametro. Esto hará referencia a la variable original del invocador, la cual puede ser modificada en la función llamada.

El prototipo de la función debe coincidir con su encabezado.

Las referencias también pueden servir como alias de otras variables, por ejemplo:

```
int count=1; /* declara variable entera count */
int &cref = count; /* crea cref como alias de count */
++cref; /* incrementa count (utilizando su alias) */
```

las variables de referencia deben inicializarse en sus declaraciones. Una vez que una referencia se declara como alias de otra variable todas las operaciones realizadas sobre el alias se realizan en la variable original. El alias simplemente es otro nombre de la variable.

1.2.2. Empleo de const con apuntadores

El calificador **const** le permite al programador informarle al compilador que no se debe modificar el valor de una variable en particular.

Si un valor no debe cambiar en el cuerpo de una función a la cual se pasa, el parámetro debe ser declarado como const con el fin de asegurar que no sea modificado por accidente. Si se intenta modificar un valor const, el compilador lo detecta y emite un aviso o un error, dependiendo del compilador.

Por ejemplo, un apuntador no constante hacia datos constantes es un apuntador que se puede modificar para que apunte hacia cualquier elemento de información del tipo apropiado, pero los datos a los que apunta no pueden modificarse mediante ese apuntador.

Veamos un ejemplo:

```
void imprimo(const int *a)
{
     int i=0;

     a=&i;
     cout << *a;
}
```

No está permitido hacer: *a=100;

Al invocar una función con un arreglo como argumento, éste se pasa de manera automática a la función simulando una llamada por referencia, sin embargo las estructuras siempre se pasan mediante llamada por valor. Esto requiere la sobrecarga en tiempo de ejecución que es causada por copiar todos los datos. Cuando hay que pasar una estructura a una función puede utilizarse un apuntador hacia datos constantes logrando en desempeño de una llamada por referencia y la protección de una llamada por valor.

Un apuntador constante hacia datos no constantes es un apuntador que siempre apunta a la misma localidad de memoria, los datos de dicha localidad se pueden modificar mediante dicho apuntador. Esto es lo predeterminado para un nombre de arreglo, el cual es un apuntador constante hacia el inicio del arreglo. En general:

```
int main()
{
    int x, y;

    int * const ptr = &x;

    *ptr = 7; /* permitido */
    ptr = &y; /* no permitido */
    return 0;
}
```

El menor privilegio de acceso se logra mediante un apuntador constante a datos constantes. Ésta es la manera en que debe pasarse un arreglo a una función que solo consulta dicho array y no lo modifica. La declaración es de la forma

const int *const ptr;

1.3. Expresiones de apuntadores y aritmética de apuntadores

Los apuntadores son operandos válidos en expresiones aritméticas, las expresiones de asignación y las expresiones de comparación. Sin embargo, no todos los operadores que se utilizan normalmente en estas expresiones son válidos con variables de apuntador.

Veremos que operadores pueden tener punteros como operandos y la forma en que se utilizan.

Un apuntador puede incrementarse (++) o decrementarse (- -) además es posible sumar un entero a un apuntador (+ o +=), restar un entero de un apuntador (- o -=) y restar un apuntador de otro.

Suponga se ha declarado un arreglo int v[5] y que su primer elemento está en la localidad 3000 de la memoria. Además suponga se ha inicializado el apuntador vPtr para que apunte a v[0], es decir el valor de vPtr es 3000.vPtr puede inicializarse mediante cualquiera de las instrucciones:

vPtr=v;
o vPtr = &v[0];

en la aritmetica de apuntadores, cuando a un apuntador se suma o resta un entero, dicho apuntador se incrementa o decrementa en el entero multiplicado por el tamaño del objeto hacia el que apunta el apuntador. Por ejemplo

vPtr += 2;

producirá $3000 + 2 * 4 = 3008$ suponiendo que los enteros se almacenan en 4 bytes de memoria. En el arreglo, vPtr apuntará a v[2].

Al efectuar aritmética de apuntadores sobre un arreglo de caracteres, los resultados serán consistentes con la aritmética común pues cada char tiene un byte de longitud.

Si vPtr se hubiera incrementado a 3016, que apunta a v[4] la instrucción

vPtr -= 4;

establecería vPtr nuevamente a 3000. Las instrucciones:

++vPtr;

vPtr++;

incrementan el apuntador para que apunte a la siguiente localidad del arreglo y

- - vPtr;

vPtr - -;

decrementan el apuntador para que apunte al elemento previo del arreglo.

La utilización de aritmética de apuntadores sobre un apuntador que no hace referencia a un arreglo normalmente es un error de lógica. También lo es salirse de cualquiera de los extremos de un arreglo.

1.4. Relación entre apuntadores y arreglos

Podemos utilizar intercambiablemente nombres de arreglo y apuntadores.

Suponga hemos declarado un array de enteros b[5] y un puntero a entero bPtr. Podemos hacer que bPtr apunte al primer elemento del array mediante

bPtr=b;
o bPtr=&b[0]

el elemento b[3] del arreglo puede referenciarse mediante

*(bPtr+3)

la dirección del tercer elemento se puede escribir como

&b[3]
o
bPtr+3

El arreglo mismo se puede utilizar como apuntador, por ejemplo

*(b+3)

también se refiere a b[3]. De igual modo se pueden utilizar punteros como si fueran en nombre del array, por ejemplo:

bPtr[1]

hace referencia a b[1].

Como los nombres de array son apuntadores constantes, la expresión

b += 3;

es inválida (a diferencia de los apuntadores no contantes).

2. Apuntadores y cadenas

Los arreglos pueden contener apuntadores. Un uso común es para formar **arreglos de cadenas**. Cada entrada del arreglo es una cadena, y en C/C++ una cadena es un apuntador a su primer caracter.

Considere la declaración del arreglo de cadena palo que podria ser util para representar barajas:

char *palo[4] = {“Corazones”,“Diamantes”, “Treboles”, “Espadas”}

La declaración indica un arreglo de 4 elementos. La parte char * indica que cada elemento del arreglo es de tipo puntero a char.

Cada uno de los cuatro valores se almacena como una cadena de caracteres que termina con el caracter nulo (o sea ’\0’) luego la longitud de las cadenas es uno más que el número de caracteres que hay entre comillas.

Cada apuntador apunta al primer caracter de su cadena correspondiente.

Una constante de cadena escrita como:

”Soy una cadena”

es un arreglo de caracteres. Podemos encontrar el fin de una cadena encontrando el caracter nulo.

Ilustremos más aspectos de los apuntadores y las cadenas viendo dos funciones útiles. La primera es strcpy(s,t) que copia la cadena t a la cadena s. Sería agradable decir simplemente $s = t$ pero esto copia el apuntador no los caracteres. Para copiar los caracteres se requiere un ciclo. La primera versión es:

```
void strcpy(char *s, char *t)
{
      int i;

      i=0;
      while ((s[i]=t[i])!=’\ 0’) i++;
}
```

la versión con apuntadores es:

```
void strcpy(char *s, char *t)
{

      while ((*s=*t)!=’\ 0’) {
            s++;
            t++;
      }
}
```

otra forma de escribir strcpy sería:

```
void strcpy(char *s, char *t)
{
      while ((*s++=*t++)!='\ 0'); }
```

esto traslada el incremento de s y t hacia dentro del ciclo.

La segunda rutina que examinaremos es strcmp que compara dos cadenas de caracteres. Regresa un valor negativo, cero o positivo según s sea lexicográficamente menor, igual o mayor que t. El valor se obtiene al restar los caracteres de la primera posición en que s y t no coinciden.

```
void strcmp(char *s, char *t)
{
      int i;

      for(i=0;s[i]==t[i];i++)
         if (s[i] == '\ 0') return 0;
      return (s[i]-t[i]);
}
```

la versión con apuntadores es:

```
void strcmp(char *s, char *t)
{
      int i;

      for(;*s==*t;s++,t++)
         if (*s == '\ 0') return 0;
      return *s-*t;
}
```

3. Apuntadores a funciones

Un apuntador a una función contiene la dirección que tiene la función en la memoria. Un nombre de función es la dirección inicial en memoria del código que lleva a cabo la tarea de la función.

Los apuntadores a funciones pueden ser pasados a las funciones, devueltos por ellas, almacenados en arreglos y asignados a otros apuntadores a funciones.

Consideremos como ejemplo el programa de la burbuja. La función burbuja recibe como argumento un apuntador a una función. Las funciones cuyo apuntador pasamos como argumento son : ascendente y descendente.

El programa le pide al usuario que seleccione si el arreglo debe ordenarse en forma ascendente o descendente.

Veamos el programa (suponemos intercambio está definida):

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

```
int descendente (int a, int b)
{ return b>a; }

void burbuja(int A[],const int tamanio,int (*compare)(int,int))
{
      for(int i=1;i<tamanio;i++)
         for(j=0;j<tamanio-1;j++)
            if ((*compare)(A[j],A[j+1]))
               intercambio(&A[j],&A[j+1]);
}
```

observaciones:
&A[i] y A+i son identicas.
Si pa es un apuntador con la dirección de a, entonces pa[i] es identico a *(pa+i).

Las llamadas a burbuja serán en la forma:

```
burbuja(A,n,ascending);
```
y
```
burbuja(A,n,descending)
```

El siguiente parámetro aparece en el encabezado de la función burbuja:

```
int (*compare)(int,int)
```

esto le dice a burbuja que espere un parámetro que es un apuntador a una función que recibe dos parámetros enteros y devuelve un resultado entero. Se necesitan parentesis alrededor de compare porque * tiene menor precedencia que el paréntesis que encierra los parámetros de la función. Si no se hubieran incluido los paréntesis la declaración habría sido

```
int *compare(int,int)
```

que declara una función que recibe dos enteros y devuelve un apuntador a entero.

El parámetro correspondiente al prototipo de función de burbuja es

```
int (*)(int,int)
```

Uno de los usos de los apuntadores a funciones es en los sistemas operados por menús. Al usuario se le solicita desde un menú que de una opción. Cada opción es atendida por una función distinta. En un arreglo de apuntadores a funciones se almacenan los apuntadores a las funciones. Se toma la selección del usuario como subindice del arreglo. Ejemplo:

supongamos tenemos funciones : funcion1, funcion2, funcion3 que tienen un entero como argumento. Los prototipos son:

```
void function1(int);
void function2(int);
void function3(int);
```

inicializamos el arreglo con:

void (*f[3])(int) = {funcion1,funcion2,funcion3};

invocamos la función con indice i y argumento j como sigue:

(*f[i])(j)