

Programación y Análisis de Algoritmos

Dr. Norberto Alejandro Hernández Leandro

CIMAT-Unidad Monterrey

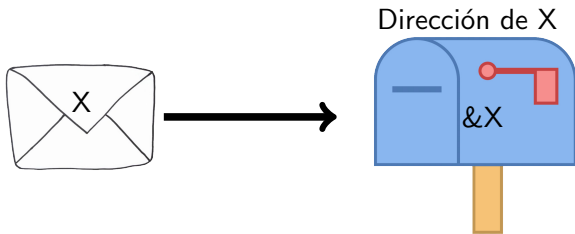
Septiembre 2020

Contenido

- Apuntadores
- Memoria dinámica
- Escritura y lectura de archivos

Direcciones

- Una variable declarada en un código está representada por una dirección y un valor.
- La dirección de una variable le indica al programa dónde está guardado el valor de dicha variable.
- Para acceder a la dirección de una variable se hace uso del operador de referencia `&`.



Apuntadores

- Los apuntadores representan variables que almacenan direcciones de memoria.
- Para acceder al valor que guarda una variable apuntador se hace uso del operador de desreferencia `*`.

```
1      int a = 40;
2      int *a_ptr = &a;
3      cout << a << endl;           //Imprime 40
4      cout << &a << endl;          //Imprime 0x7ffe39fcbdf4
5      cout << a_ptr << endl;        //Imprime 0x7ffe39fcbdf4
6      cout << *a_ptr << endl;       //Imprime 40
```

Apuntadores


- El operador de desreferencia tiene mayor prioridad de que los operadores aritméticos.

```
1      int a = 10;  
2      int *b = &a;  
3      int c = a**b**b;  
4      cout << c << endl;
```

Apuntadores y funciones

- Para objetos muy grandes, puede resultar muy costoso el pasarlos como argumentos de una función; más aún si la función es llamada repetitivamente dentro de un algoritmo. Sin embargo, se puede evitar todo este trabajo si se utilizan apuntadores.

```
1  typedef struct data{
2      tipo1 dato_1;
3      tipo2 dato_2;
4      ...
5      tipon dato_n;
6  };
7
8  void foo1(data dat){
9      tipo1 d = dat.dato_1;
10     ...
11 }
12
13 void foo2(data *dat){                //Ojo, si se modifica un elemento
14     tipo1 d = dat->dato_1;           //de dat, el objeto del que
15     ...                               //proviene también se modifica
16 }
```



Apuntadores y funciones

- Si se requiere más de una salida de una función, las salidas extras al `return` se pueden obtener mediante el uso de apuntadores.

```
1      int maximo(int Arr[], int tamaño, int *indice){
2          int maxi = INT_MIN;          // Librería limits.h o climits
3          for(int i = 0; i < tamaño ; i++){
4              if(Arr[i] > maxi){
5                  maxi = Arr[i];
6                  *indice = i;
7              }
8          }
9          return maxi;
10     }
11     int main(){
12         int maxi, indice;
13         ...          // Definición del arreglo
14         maxi = maximo(Arr, tamaño, &indice);
15         return 0;
16     }
```

Aritmética de apuntadores

Los apuntadores son números enteros en representación hexadecimal; por tanto, se puede operar con ellos. Sin embargo, no todas las operaciones tienen sentido.

- **Suma/Resta de un número entero:** el resultado es un apuntador del tipo correspondiente, y representa una traslación en la memoria.
- **Resta de dos apuntadores:** el resultado es un apuntador del tipo correspondiente, y representa una dirección de memoria que ese encuentra entre los dos apuntadores.

Así mismo, se pueden aplicar los operadores de comparación siempre y cuando se utilicen sobre apuntadores del mismo tipo.

Aritmética de apuntadores

Los apuntadores son números enteros en representación hexadecimal; por tanto, se puede operar con ellos. Sin embargo, no todas las operaciones tienen sentido.

- **Suma/Resta de un número entero:** el resultado es un apuntador del tipo correspondiente, y representa una traslación en la memoria.
- **Resta de dos apuntadores:** el resultado es un apuntador del tipo correspondiente, y representa una dirección de memoria que ese encuentra entre los dos apuntadores.

Así mismo, se pueden aplicar los operadores de comparación siempre y cuando se utilicen sobre apuntadores del mismo tipo.

Precaución: Cuidado al realizar aritmética de apuntadores, se puede acceder a direcciones de memoria del sistema que si se modifican pueden causar un error crítico en el S.O.

Aritmética de apuntadores

Las operaciones sobre apuntadores son muy útiles al manejar arreglos.

```
1  #define N 5
2
3  int tab [N] = {1 , 2 , 6 , 0 , 7 } ;
4
5  int main ( ){
6      int *p ;
7
8      for ( p = &tab[0] ; p <= & tab[N -1] ; p++)
9          printf ( " %d \n" , *p );
10
11     for ( p = &tab[N -1] ; p >= &tab[0]; p--)
12         printf ( " %d \n" , *p );
13
14     return 0 ;
15 }
```

Precaución: si se hace un mal manejo de los apuntadores tendremos un error conocido como "Segmentation fault".

Arreglos dinámicos

En ocasiones no se sabe el tamaño del arreglo antes definirlo. Para resolver esta situación se tienen dos opciones:

- Usar un tamaño fijo máximo y definir un arreglo estático de ese tamaño. El programa a este punto es capaz de manejar todo.
- Usar memoria dinámica, la cual puede ser usada mediante las palabras reservadas `new` (malloc) y `delete`.

Tener cuenta que la memoria estática es asignada y liberada de manera automática. Sin embargo, esto no pasa al utilizar memoria dinámica, este tipo de memoria es persistente; esto es, que esta asignación no es liberada incluso saliendo del alcance de las variables.

Arreglos dinámicos

- Los arreglos dinámicos se definen mediante apuntadores.
- En cualquier momento se puede cambiar el tamaño del arreglo.
- Si se cambia el tamaño de un arreglo ya definido, la nueva definición contendrá los elementos correspondientes del antiguo arreglo.
- La asignación y liberación de memoria se tienen que hacer manual.
- Si la memoria no puede ser asignada, el arreglo tendrá valor **NULL**.
- Si se intenta acceder a índices no válidos dentro del arreglo, se tendrá un *Segmentation fault*.

Declaración

La declaración de un arreglo dinámico se realiza mediante la palabra reservada `new`.

```
1      int *arr = new int [5];  
2      ....  
3      arr = new int [10];
```

La liberación de la memoria dinámica se realiza mediante la palabra reservada `delete`.

```
1      delete [] arr;
```

Ejemplo

Imagina que se tiene que programar una función en la que se tiene como argumento y salida un arreglo.

```
1  #define N 100
2  int *applyFunction(int *Arr){
3      int arrOut[N];
4      ...
5      return &arrOut[0];
6  }
7  int main(){
8      int arreglo[N];
9      int *result = applyFunction(arreglo);
10     return 0;
11 }
```

Ejemplo

Imagina que se tiene que programar una función en la que se tiene como argumento y salida un arreglo.

```
1  #define N 100
2  int *applyFunction(int *Arr){
3      int arrOut[N];
4      ...
5      return &arrOut[0];
6  }
7  int main(){
8      int arreglo[N];
9      int *result = applyFunction(arreglo);
10     return 0;
11 }
```

Este error se puede arreglar usando arreglos dinámicos o declarando el arreglo como estático.

Lectura y escritura de archivos

En C/C++ existen diferentes herramientas para leer o escribir archivos de texto plano.

- FILE, fopen, fread, fwrite, fclose en C son utilizadas para guardar la información del fichero, leer, escribir y cerrar, respectivamente.
- En C++ se sustituyen estas instrucciones y se definen los tipos ifstream, ofstream, y fstream para abrir un fichero para lectura, escritura, y lectura/escritura.

Lectura y escritura de archivos

- Para poder leer información un archivo hay que definir un stream para manejarlo.

```
1 ifstream ifile(char *name, ios::in);    //stream para
2 ifile.open(char name, ios::in);        //lectura
```

- Para cambiar el modo en el que se abre un archivo, sólo hay que cambiar la definición del stream por ofile (escribir) y fstream (leer/escribir).
- Finalmente, para cerrar el stream se utiliza el método close.

```
1 ifile.close();
```

Lectura y escritura de archivos

- Se utilizan los operadores `<<` y `>>` se utilizan para leer y escribir información en el archivo.

```
1      fstream ffile("hola.txt", ios::in | ios::out);
2      char word[100];
3      ffile >> word;
4      ffile << "Hola";
5      ffile.close();
```

- Se pueden utilizar los siguientes métodos para revisar el estado del stream.

```
1      fstream ffile;
2      ...
3      ffile.is_open(); //True si stream está ligado a archivo
4      ffile.bad();      //True si no se puede leer/escribir
5      ffile.good();     //True si se puede leer/escribir
6      ffile.fail();     //True si hay error de formato
7      ffile.eof();      //True si se alcanza el final
```

Ejemplo

```
1  int main(){
2      //Escritura
3      int *vect, tam;
4      ofstream ofile("vector.txt", ios::out);
5      cout << "Dame el tamaño: " << endl;
6      cin >> tam;
7      vect = new int[tam];
8      for(int i=0; i<tam; i++){
9          vect[i] = rand() % 100;
10         ofile << vect[i] << endl;
11     }
12     ofile.close();
13
14     //Lectura
15     ifstream ifile("vector.txt", ios::in);
16     for(int i=0; i<tam; i++){
17         int elem;
18         ifile >> elem;
19         cout << elem << endl;
20     }
21     ifile.close();
22     return 0;
23 }
```