PUNTEROS

Definición: Un *puntero* es una variable que almacena una dirección de memoria de otra variable.

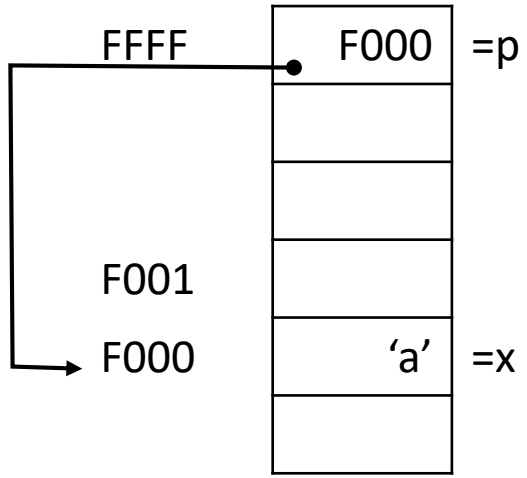
Declaración de una variable de tipo puntero (operador *)

```
<tipo> * <nombre_del_puntero> ;
```

Inicialización de una variable de tipo puntero

El operador **&** aplicado a una variable, devuelve la dirección donde se almacena dicha variable.

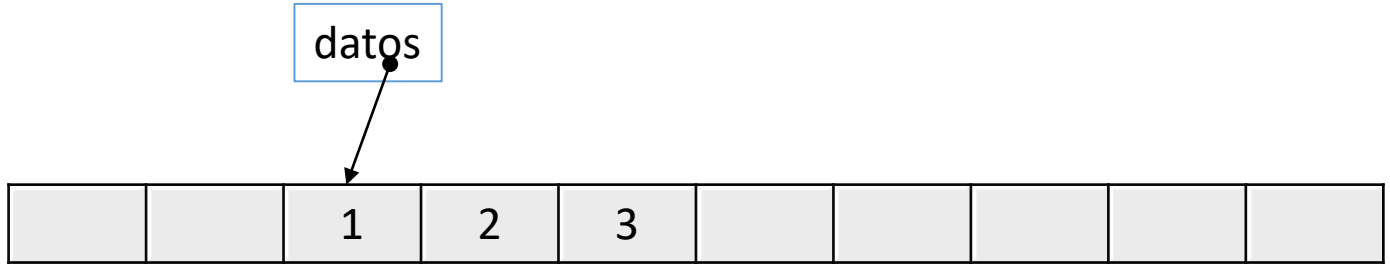
```
char *p;
char x='a';
p=&x;
cout <<p; //muestra la dirección memoria de x
cout<<*p; //muestra el contenido de la dirección de memoria
```



Relación entre arrays y punteros

Existe una fuerte relación entre un vector y un puntero. Un array es un puntero

int datos[3]




```
cout << datos[0];
cout << datos [1];
cout << datos[2];
```

```
cout << *datos;
cout << *(datos +1);
cout << *(datos+2);
```

```cpp
//punteros03.cpp
#include <iostream>
#include <stdlib.h>
using namespace std;

int main()
{
   int datos[4]={1,2,3,4};
   cout << datos[0];
   cout << datos [1];
   cout << datos[2]<<endl;
   cout << *datos;
   cout << *(datos +1);
   cout << *(datos+2);

   system("pause");

}
```

```cpp
//punteros04.cpp
//Recorrido de un vector con punteros
#include <iostream>
#include <stdlib.h>
using namespace std;

int main()
{
   int temperatura[ 10 ]={23,21,24,25,7,24,12,13,14,16};
   int *p;
   p = temperatura;
   for (int i = 0; i<10; i++)
   {
      cout<<*p<<endl;
      p++;
   }

   system("pause");

}
```

Punteros a estructuras

Podemos declarar un puntero a una estructuras tal y como se declara para cualquier otro tipo de dato.

```
struct empleado
   {
      char nombre[30];
      int edad;
      float sueldo;
   };
   // cada variable empleado ocupa 38 bytes
   empleado emp01 = { “Jose", 28, 1800};
   empleado *p =&emp01;
```

Para acceder a los miembros, tenemos dos posibilidades:

A) Utilizando el operador punto (.) sobre el valor apuntado por el puntero

(*p).nombre

B) Utilizando el operador ->

p->nombre

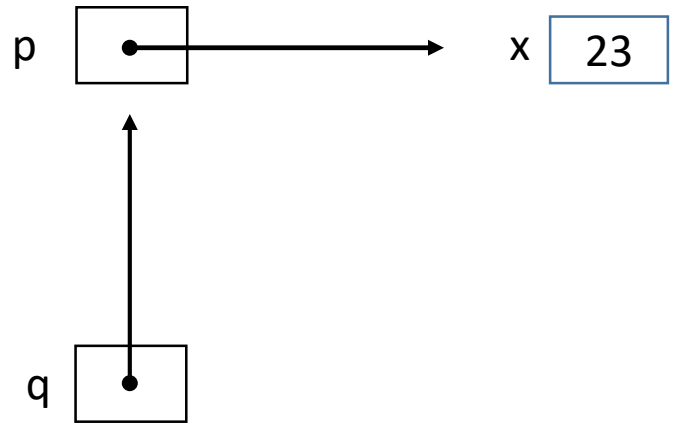
```
//punteros02.cpp
#include <iostream>
#include <stdlib.h>
using namespace std;

int main() {
   struct empleado
   {
      char nombre[30];
      int edad;
      float sueldo;
   };

   // cada variable empleado ocupa 38 bytes
   empleado emp01 = { "Jose", 28, 1800}, emp02 = {"Antonio", 30, 1200};
   empleado *p=&emp01;
   cout<<(*p).nombre<<endl; //Acceso a un miembro de struct
   cout<<p->nombre; //Otra forma de acceso
   *p=&emp02;
   cout<<(*p).nombre<<endl;
   cout<<p->nombre;
   system("pause");
   return 0
}
```

```
//punteros04.cpp
#include <iostream>
#include <stdlib.h>
using namespace std;

int main() {
   struct empleado
   {
      char nombre[30];
      int edad;
      float sueldo;
   };

   empleado plantilla[3]={ "Jose", 28, 1800,"Antonio", 30, 1200,"Maria", 48, 1400};
   empleado *p;
   p=plantilla;
   for (int i=0;i<3;i++)
   {
      cout<<p->nombre<<endl;
      p++;
   }
   system("pause");

}
```

Puntero a puntero

Un puntero puede apuntar a una variable de tipo puntero. Se declara con **

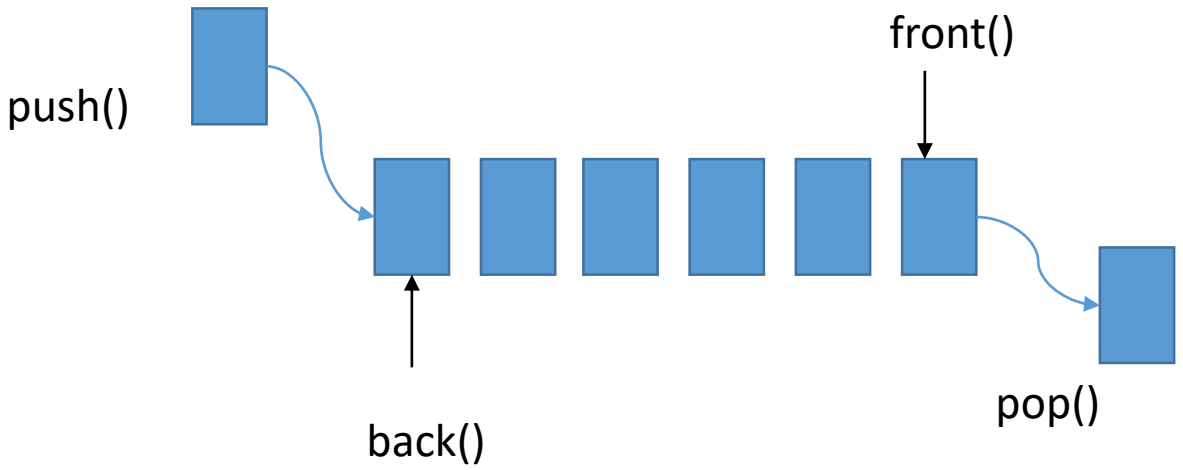
```
//punteros05.cpp
#include <iostream>
#include <stdlib.h>
using namespace std;

int main() {
   int x = 23;
   int *p ;
   int **q ;
   p = &x;
   q = &p;
   cout << *p << endl;
   cout << **q;
    system("pause");
}
```



PILAS, COLAS, LISTAS
Y HASH

ESTRUCTURA COLA FIFO (#include <queue>)

Los datos entran en última posición y se obtienen de la primera



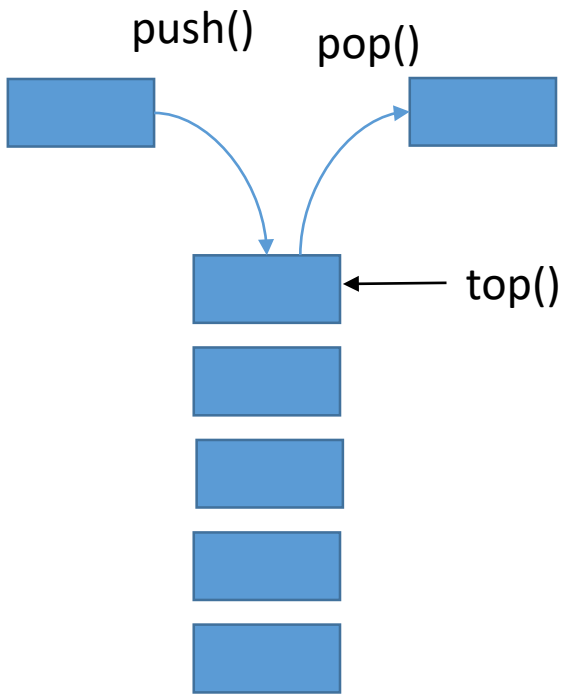
Métodos

Nombre	Descripción
empty	cierto (true) si la cola está vacia
pop	borra el elemento del frente de la cola
push	agrega un elemento al final de la cola
size	regresa el número de elementos en la cola
front	regresa una referencia al primer elemento en la cola
back	regresa una referencia al último elemento en la cola

```
//estructura cola
#include <iostream>
#include <queue>
using namespace std;

int main()
{
   queue <int> cola;
   for (int i=1;i<=10;i++)
            cola.push(i);
   cout<<"Elementos cola="<<cola.size()<<endl;
   while (!cola.empty())
   {
            cout<<cola.front()<<endl;
            cola.pop();
   }
   if (cola.empty())
            cout<<"Cola vacía";
   return 0;
}
```

ESTRUCTURA PILA LIFO (#include <stack>)

Los datos se recuperan de la última posición



Nombre	Descripción
empty	cierto (true) si la pila está vacia
pop	borra el elemento superior de la pila
push	agrega un elemento a la pila
size	número de elementos en la pila
top	referencia al primer elemento en la pila

```
//estructura pila
#include <iostream>
#include <stack>
using namespace std;

int main()
{
   stack <int> pila;
  for (int i=1;i<=10;i++)
            pila.push(i);
   cout<<"Elementos pila="<<pila.size()<<endl;
   while (!pila.empty())
   {
            cout<<pila.top()<<endl;
            pila.pop();
   }
  if (pila.empty())
            cout<<"Pila vacía";
  return 0;
}
```

LISTA DE ACCESO SECUENCIAL


```
#include <list>

List<tipodato> id_variable;
```

Métodos Listas	
back	devuelve una referencia a el último componente de la
begin	devuelve un **iterator** al principio de la lista
clear	elimina todos los componentes de la lista
empty	true si la lista está vacía
end	devuelve un **iterator** al final de la lista
pop_back	elimina el último componente de la lista
pop_front	elimina el primer componente de la lista
push_back	añade un componente al final de la lista
push_front	añade un componente al frente de la lista
size	devuelve el número de componentes en la lista
sort	ordena la lista (no para listas acceso directo)

```
//lista01.cpp
using namespace std;
void crea_lista(list<string> & lista)
{
   lista.push_back("Juan");
   lista.push_back("Oscar");
   lista.push_back("Samantha");
   lista.push_back("Angela");
}

void mostrar_lista(list<string>  lista)
{
   list<string>::iterator it = lista.begin(); //puntero al comienzo lista
   while( it != lista.end() )
        cout << "\t" << *it++ << endl;
}

int main()
{
   list<string>  lista_clase;

   crea_lista(lista_clase);
   mostrar_lista(lista_clase);
   lista_clase.sort();
   mostrar_lista(lista_clase);
   lista_clase.clear();
   mostrar_lista(lista_clase);
}
```

LISTA DE ACCESO DIRECTO

```
#include <vector>
vector<tipodato> id_variable;
```

//lista02.cpp

```
#include <string>
#include <iostream>
#include <vector>
using namespace std;
void crea_vector(vector<string> & v)
{
   v.push_back("Juan");
   v.push_back("Oscar");
   v.push_back("Samantha");
   v.push_back("Angela");
}
void mostrar_vector(vector<string> v)
{
   for (int i=0;i<v.size();i++)
                cout<<v[i]<<endl;
}
int main()
{
   vector<string>  v_clase;
   crea_vector(v_clase);
   mostrar_vector(v_clase);
   v_clase.clear();
   mostrar_vector(v_clase);
}
```

TABLAS HASH

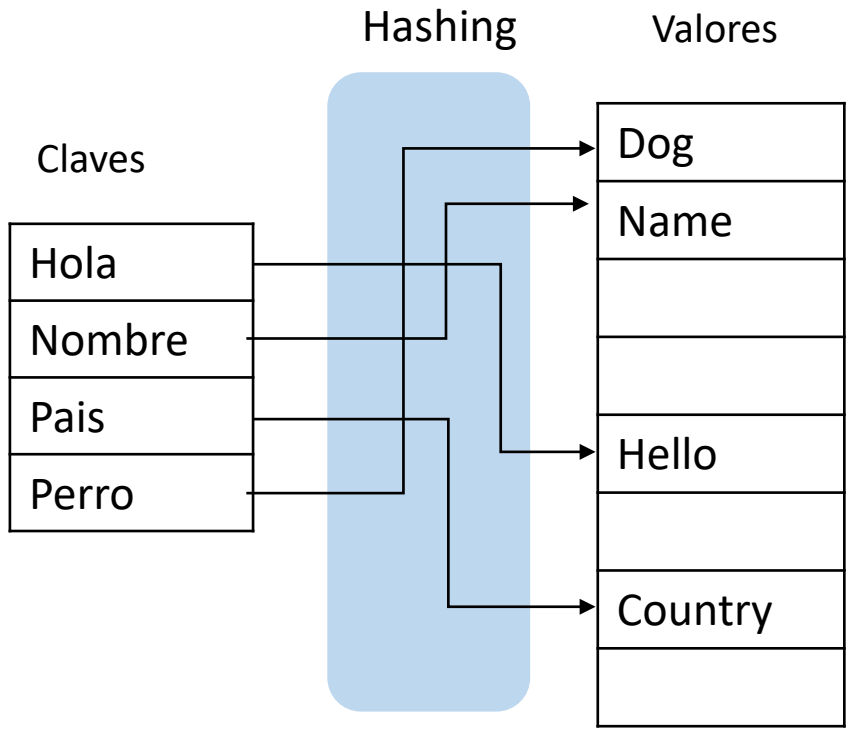
Un diccionario es una estructura donde podemos guardar datos ordenados según una clave, de tal modo que resulte muy eficiente buscar el dato conociendo su clave correspondiente.

map<K , T> donde K es el tipo de clave y T el tipo de datos

Ejemplo:

map<string,string> traductor;

```
traductor["Perro"]="Dog";
traductor["Nombre"]="Name";
traductor["Pais"]="Country";
traductor["Hola"]="Hello";
```



```
//hash01.cpp
#include <string>
#include <iostream>
#include <map>

using namespace std;

void crea_traductor(map<string,string> & traductor)
{
   traductor[“Perro"]=“Dog";
   traductor["Nombre"]="Name";
   traductor["Pais"]="Country";
   traductor[“Hola"]=“Hello";

}

void mostrar_traductor(map<string,string>  traductor)
{
   map<string,string>::iterator it = traductor.begin();

   while( it != traductor.end() )
  {
        cout<<it->first<<"  "<<it->second<<endl;
        it++;
   }
}
```

```
string traducir(map<string,string>  traductor , string s)
{
      map<string, string>::iterator p = traductor.find(s);
      if (p != traductor.end())
                return p->second;
      else
                return "No encontrada";
}

int main()
{
   map<string,string>  traductor;

   crea_traductor(traductor);
   mostrar_traductor(traductor);
   cout<<traducir(traductor,"Perro");
}
```