POINTERS

Pointers

These are the variables that are used to hold the address of another variable

Pointer variable

- A pointer is a variable that contains the memory location of another variable.
- Syntax:-

type * variable name

- start by specifying the type of data stored in the location identified by the pointer.
- The asterisk tells the compiler that you are creating a pointer variable.
- Finally you give the name of the variable.

Features of Pointers

- Execution time with pointer is faster because data is manipulated with the address i.e. direct access to memory location.
- Dynamically memory is allocated.
- Pointers are used with data structures. They are useful for representing 2D and Multi-D array.

Declaration and Initialization

- Syntax for declaring pointer is

data_type * ptr variable name;

Example : int *p;

Syntax for Initialization of a pointer is

ptr var name= & var name;

Example : p=&a;

Declaring a Pointer Variable

- To declare ptr as an integer pointer:

```
int *ptr;
```

- To declare ptr as a character pointer:

```
char *ptr;
```

Address operator:

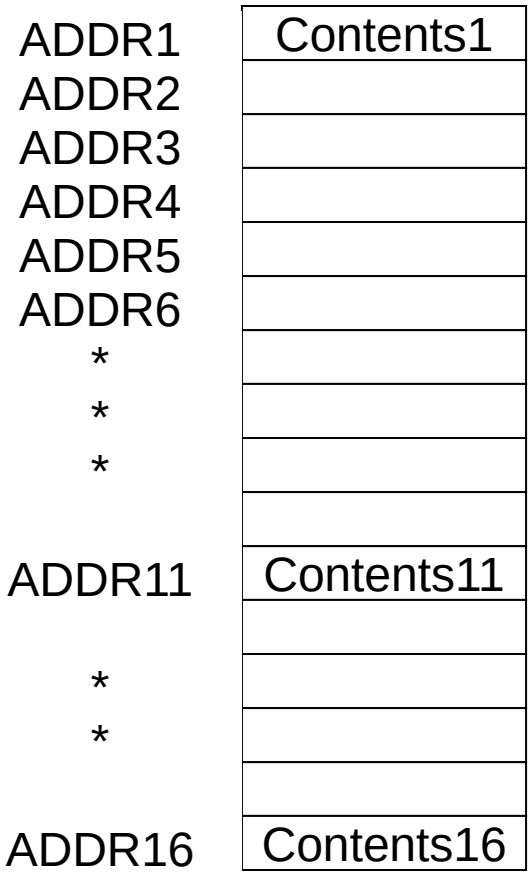
- Once we declare a pointer variable we must point it to something we can do this by assigning to the pointer the address of the variable you want to point as in the following example:

  ptr=#

- This places the address where num is stores into the variable ptr. If num is stored in memory 21260 address then the variable ptr has the value 21260.

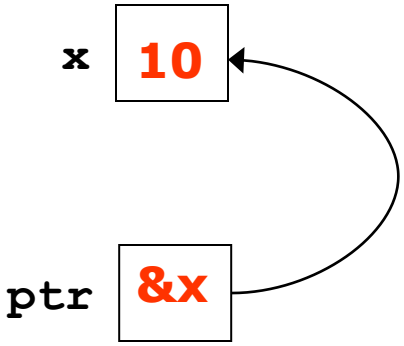
Address and Pointers

- Memory can be conceptualized as a linear set of data locations.
- Variables reference the contents of a locations
- Pointers have a value of the address of a given location



Pointer Variable

Assume ptr is a pointer variable and x is an integer variable




```
x = 10

ptr = &x
```

Now ptr can access the value of x.

HOW!!!!

Write: `*variable`. For example:

Cout<< *ptr;

Program to display variable value and address value

```
int main ()
{
int *p,a;
cout<<“enter the value of a”;
cin>>a;
p=&a;
cout<<“address of variable a using pointer is”<<p;
cout<< value of variable a using pointer is”<<*p;
}

// variable address= p
// variable value=*p
```

Another Program

```
int  a=20;
int *p;
p=&a;
cout<<a<<&a;
cout<<p<<&p<<*p;
```

P has an address 500

Variables, Addresses and Pointers

```
main()
{
int a = 5;
int b = 6;
int *c;
// c points to a
c = &a;
}
```

Memory	Value
a (1001)	5
b (1003)	6
c (1005)	1001

Pointer to pointer

Pointer variable that holds the address of another pointer

Declaration :

```
int **p1,*p;
```

Initialization

```
p1=&p;
```

Pointers to pointers

- A pointer variable containing address of another pointer variable is called pointer to pointer

```
void main()
{
int a=2, *p, **q;
p=&a;
q=&p;
cout<<a<<“is stored at ”<<p<<“and pointer is stored at ”<<q;}
```

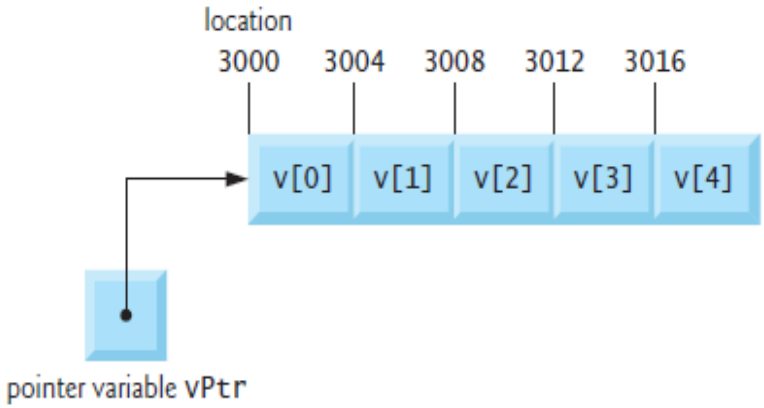
Pointer arithmetic

Valid operations on pointers include:

- the *sum of a pointer and an integer*
- the *difference of a pointer and an integer*
- pointer *comparison*
- the *difference of two pointers.*
- *Increment/decrement in pointers*
- *assignment operator used in pointers*

- Assume that array int v[5] has been defined and its first element is at location 3000 in memory.
- Assume pointer vPtr has been initialized to point to v[0]—i.e., the value of vPtr is 3000. Figure illustrates this situation for a machine with 4-byte integers.
- Variable vPtr can be initialized to point to array v with either of the statements

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



Array v and a pointer variable vPtr that points to v.

- In conventional arithmetic, 3000 + 2 yields the value 3002.
- This is not the case with pointer arithmetic. When an integer is added to or subtracted from a pointer, the pointer is not incremented or decremented simply by that integer, but by that integer times the size of the object to which the pointer refers.
- For example, the statement

```
vPtr += 2;
```

would produce 3008 **(**3000 + 2 * 4**),** assuming an integer is stored in 4 bytes of memory.

- If vPtr had been incremented to 3016, which points to v[4], the statement

```
vPtr -= 4;
```

would set vPtr back to 3000

- The statements

```
++vPtr;
vPtr++;
```

increments the pointer to point to the next location in the array

- The statements

```
--vPtr;
vPtr--;
```

decrements the pointer to point to the previous element of the array.

- Pointer variables may be subtracted from one another.
- For example, if vPtr contains the location 3000, and v2Ptr contains the address 3008, the statement

```
x = v2Ptr - vPtr;
```

would assign to x the number of array elements from vPtr to v2Ptr, in this case 2 (not 8).

- Pointers can be compared using equality and relational operators, but such comparisons are meaningless unless the pointers point to elements of the same array.

Comparison in pointers

- Two pointers of the same type, **p and q, may be compared as long** as *both of them point to objects within a single memory block*
- Pointers may be compared using the <, >, <=, >=, == , !=
- When you are comparing two pointers, you are comparing the values of those pointers *rather than the contents of memory* locations pointed to by these pointers

Pointer Arithmetic

```
 int main()
{
int *p,a,**p1;
cout<<“Enter the value of a”;
cin>>a;
p=&a;
p1=&p;
cout<<“address of variable a using ptr”<<p;
p=p+4;
cout<<“modified address is”<<p;
cout<<“value of variable a using ptr is”<<*p;
*p=*p+20;
cout<<“modified value of variable a is ”<<*p;
cout<<“address of pointer p is”<<p1;
}
```

Pointer Arithmetic

```
 void main()
{
 int a=25,b=78,sum;
 int *x,*y;
 x=&a;
 y=&b;
  sum= *x + *y;
 cout<<“Sum is : ”<<sum;
}
```

Which of the following is not true about pointer arithmetic?

(a) pointers can be added.

(b) pointers can be subtracted.

(c) pointer can be incremented.

(d) pointer can be decremented.

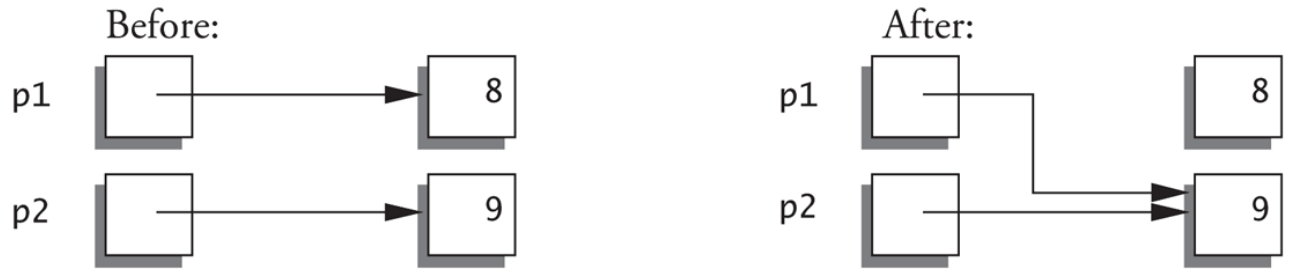
Assignment in pointers

- Pointer variables can be "assigned"
  int *p1, *p2;
  p2 = p1;
  - Assigns one pointer to another
  - "Make p2 point to where p1 points"
- Do not confuse with:
  *p1 = *p2;
  - Assigns "value pointed to" by p1, to "value pointed to" by p2

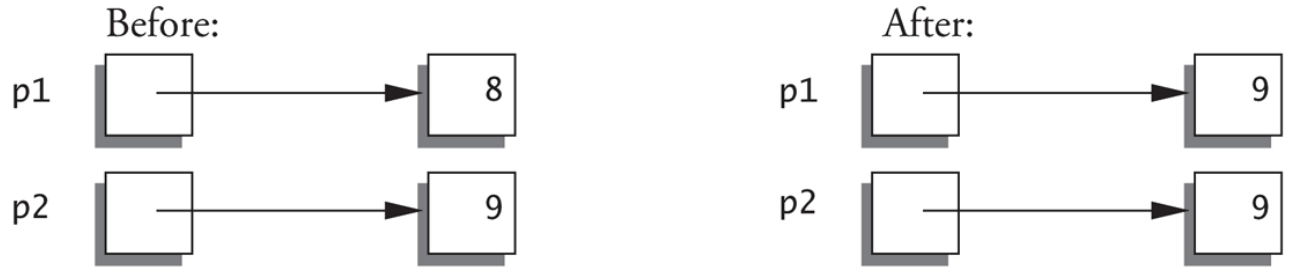
Diagrammatic representation

Display 10.1 Uses of the Assignment Operator with Pointer Variables

```
p1 = p2;
```




```
*p1 = *p2;
```



Void Pointer

- It is called the Generic pointer
- It is a special type of pointer that can be pointed at objects of any data type
- A void pointer is declared like a normal pointer, using the void keyword as the pointer's type
- Example: void *pVoid; // pVoid is a void pointer

- This is very useful when you want a pointer to point to data of different types at different times.

- Syntax:

  void * variable name;

  Print value stored in variable

  ***(data_type*)name of variable;**

Example

```
#include<iostream>
using namespace std;
int main()
{
        int x=10;
        char ch='A';
        void *gp;
        gp=&x;
        cout<<*(int*)gp<<endl;
        gp=&ch;
        cout<<*(char*)gp<<endl;
        return 0;
}
```

Other Properties of Void Pointer

- It Can’t be Dereferenced because void pointer does not know what type of object it is pointing to.
- It is not possible to do pointer arithmetic on a void pointer.

Constant Pointers

- Constant pointer means the pointer is constant.
- The address of constant pointer cannot be modified.
- This statement declares a pointer variable p which points to a constant integer value.
- int * const ptr2(Constant pointer) indicates that ptr2 is a pointer which is constant. This means that ptr2 cannot be made to point to another integer.

However the integer pointed by ptr2 can be changed.

- const int* const p;(Constant pointer to constant)

This statement declares a named constant pointer p that points to a constant integer value

- const int *p;(Pointer to constant)

Null Pointer

- A null pointer is generally used to signify that a pointer does not point to any object
- NULL pointer is a type of pointer of any data type and generally takes a value as zero. This is, however, not mandatory. This denotes that NULL pointer does not point to any valid memory address.

Example of Null Pointer

```
#include<iostream>
using namespace std;
int main()
{
        int *ptr=NULL;
        int a=10;
        cout<<ptr<<endl;
        ptr=&a;
        cout<<*ptr;
        return 0;
}
```

Difference between Null Pointer and Void Pointer

A Void pointer is a special type of pointer of void and denotes that it can point to any data type.

NULL pointers can take any pointer type, but do not point to any valid reference or memory address

Null Pointer

- NULL value can be assigned to any pointer, no matter what its type.

```
void *p = NULL;
int i = 2;
int *ip = &i;
p = ip;
cout<<*p;
cout<<*((int*)p ) ;
```

Output=2

```
#include<iostream>
using namespace std;
int main()
{
   int a=10;
   void *ptr=&a;
   cout<<*ptr;
}
```

a) Prints 10 on the screen.
b) Prints an address on the screen.
c) Prints a garbage value on screen.
d) Results in a compiler error.

Reference Variables

Reference variable = ***alias for another variable***

- Contains the address of a variable (like a pointer)
- No need to perform any dereferencing (unlike a pointer)
- Must be initialized when it is declared

```
int x = 5;
int &z = x;            // z is another name for x
int &y ;               //Error: reference must be initialized
cout << x << endl;     -> prints 5
cout << z << endl;     -> prints 5

z = 9;                 // same as x = 9;

cout << x << endl;     -> prints 9
cout << z << endl;     -> prints 9
```

Reference Variables

```
int a=5;
int *p;
p=&a;
cout<<p<<*p<<&p;
int &b=a;
a=a+2;
cout<<b;
```

BASIS FOR COMPARISON	POINTER	REFERENCE
Basic	The pointer is the memory address of a variable.	The reference is an alias for a variable.
Returns	The pointer variable returns the value located at the address stored in pointer variable which is preceded by the pointer sign '*'.	The reference variable returns the address of the variable preceded by the reference sign '&'.
Operators	*, ->	&
Null Reference	The pointer variable can refer to NULL.	The reference variable can never refer to NULL.
Initialization	An uninitialized pointer can be created.	An uninitialized reference can never be created.
Time of Initialization	The pointer variable can be initialized at any point of time in the program.	The reference variable can only be initialized at the time of its creation.
Reinitialization	The pointer variable can be reinitialized as many times as required.	The reference variable can never be reinitialized again in the program.

```
int main()
{
int x = 10, y = 20;
int *ptr = &x;
int &ref = y;
*ptr++;
ref++;
cout<< x << " " << y;
return 0;
}
```

(a) 10 21
(b) 10 20
(c) 11 20
(d) 11 21

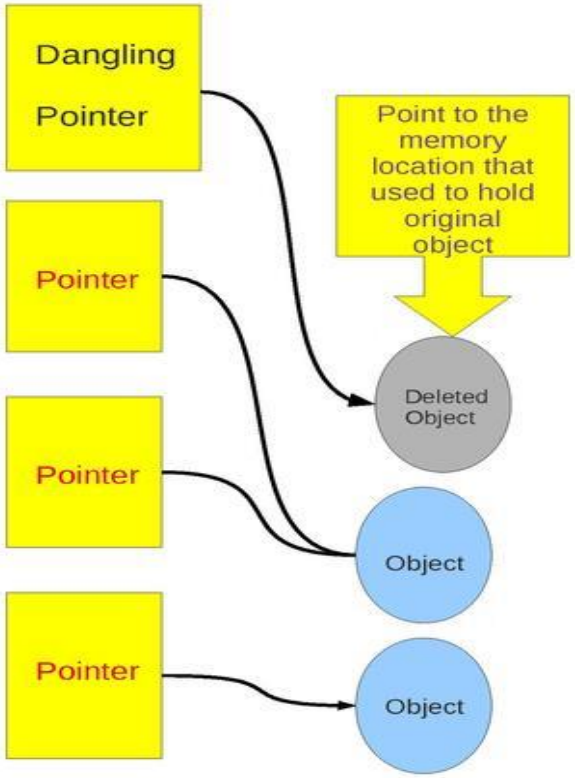
```
#include<iostream>
using namespace std;
void print(int *p,int m)
{
  m=m+5;
  *p=*p+m;
}
int main()
{
int i=5,j=10;
print(&i,j);
cout<<i+j;
return 0;
}
```

a)15
b)25
c)30
d)20

Dangling Pointers

- Dangling pointers arise when an object is deleted or de allocated, without modifying the value of the pointer, so that the pointer still points to the memory location of the de allocated memory.

Dangling Pointer



Example

```
#include<iostream>
using namespace std;
int main()
{
  int *ptr;
  {
     int v=23;
     ptr = &v;
     cout<<"Value is(inside block):"<<*ptr<<"\n";
     cout<<"Address is(inside block):"<<ptr<<"\n";
  }
  // Here ptr is dangling pointer as v is no longer existing
cout<<"Value is(outside block):"<<*ptr<<"\n";
cout<<"Address is(outside block):"<<ptr;//ptr is dangling pointer(same address)
     ptr=NULL;//Solution to dangling pointer(assign null address)

}
```

Wild Pointer

- Wild pointer arise when pointer is used prior to initialization some to known slot.
- They show the same unpredictable behaviour like dangling pointer but they are less likely to go undetected.
- Created by omitting necessary initialization prior to its first use. So every pointer in programming language begins as a wild pointer.

Pointer within class

```
#include<iostream>
using namespace std;
class Array
{
int *arr;
int size;
public:
void get_data(int n)
{
size=n;
arr=new int[size];
cout<<"\nEnter elements:";
for(int i=0;i<size;i++)
{
cin>>*(arr+i);
}
}
void add()
{
int sum=0;
for(int i=0;i<size;i++)
{
sum+=*(arr+i);
}
cout<<"\n Sum of elements="<<sum;
}
};
```

```
int main()
{
Array a;
int n;
cout<<"\n Enter the number of elements:"<<endl;
cin>>n;
a.get_data(n);
a.add();
return 0;
}
```

Pointer to Objects

- A pointer to a C++ class is done exactly the same way as a pointer to a structure and to access members of a pointer to a class you use the member access operator **->** operator, just as you do with pointers to structures. Also as with all pointers, you must initialize the pointer before using it.

Example

```
#include<iostream>
using namespace std;
class A
{
int x;
public:
void getdata()
{
cout<<"\n Enter value for x:"<<endl;
cin>>x;
}
void showdata()
{
cout<<"\n Entered value is:"<<x<<endl;
}
};
int main()
{
A obj1;
A *ptr;
ptr=&obj1;//Pointer to object
ptr->getdata();
ptr->showdata();
(*ptr).getdata();
(*ptr).showdata();
}
```

Pointers to Members

- It is possible to take address of a member of a class and assign it to a pointer.
- The address of a member can be obtained by applying the operator & to a “fully qualified “ class member name.
- A class member pointer can be declared using the operator ::* with the class name.

```
Ex:  class A
     {
       private:
         int m;
       public:
         void  show();
      };
```

- Define a pointer to the member m as follows:

  int A ::* ip = &A :: m;

  – ip pointer created thus acts like a class member in that it must be invoked with a class object.
  – A::* means “pointer-to-member” of A class.
  – &A::m means the “address of the m member of A class”.
  – ip now be used to access the member m inside member functions.

- The dereferencing operator ->* is used to access a member when we use pointers to both the object and the member.
- The dereferencing operator .* is used when object itself is used with the member pointer.

Example

```
#include <iostream>

using namespace std;

class X
 {
 public:
 int a;
void f(int b)
 {
 cout << "The value of b is "<< b << endl;
 }
 };
 int main()
{
// declare pointer to data member
 int X::*ptiptr = &X::a;

 // declare a pointer to member function
 void (X::* ptfptr) (int) = &X::f;

// create an object of class type X

 X xobject;

 // initialize data member
 xobject.*ptiptr = 10;

cout << "The value of a is " << xobject.*ptiptr<<
endl;

// call member function
(xobject.*ptfptr) (20);
 }
```

```
#include<iostream>
using namespace std;
class M
{
int x;
int y;
public:
void set_xy(int a, int b)
{   x=a;   y=b;    }
friend int sum(M m);
};
int sum(M m)
{
// declare pointer to data member
int M :: * px =&M :: x;
int M :: * py =&M :: y;
M *pm=&m;
int S = m.*px + pm->*py;
return S;
}
```

```
int main()
{
M n;
//declare pointer to member function
void (M :: *pf)(int,int) = &M :: set_xy;

// call member function
(n.*pf)(10,20);
cout<<“SUM=“<<sum(n)  ;

M *op = &n;
(op->*pf)(30,40);
cout<<“SUM=“<<sum(n);

return 0;
}
```

this pointer

- Every object in C++ has access to its own address through an important pointer called **this** pointer.
- The **this** pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.
- Friend functions do not have a **this** pointer, because friends are not members of a class. Only member functions have a **this** pointer.
- We cannot declare this pointer or make assignments to it.
- A static member function does not have a this pointer.

Example

```
#include <iostream>
using namespace std;
 class X
{
private:
 int a;
 public:
void Set_a(int a)
 {
 // The 'this' pointer is used to retrieve 'xobj.a'
// hidden by the automatic variable 'a'
this->a = a;
 }
void Print_a()
{
cout << "a = " << a << endl;
 }
 };
int main()
 {
X xobj;
 int a = 5;
xobj.Set_a(a);
xobj.Print_a();
}
```

this pointer

(A) implicitly points to an object.

(B) can be explicitly used in a class.

(C) can be used to return an object.

(D) All of the above.

Which of the following ways are legal to access a class data member using this pointer?

(A) this.x

(B) *this.x

(C) *(this.x)

(D) (*this).x