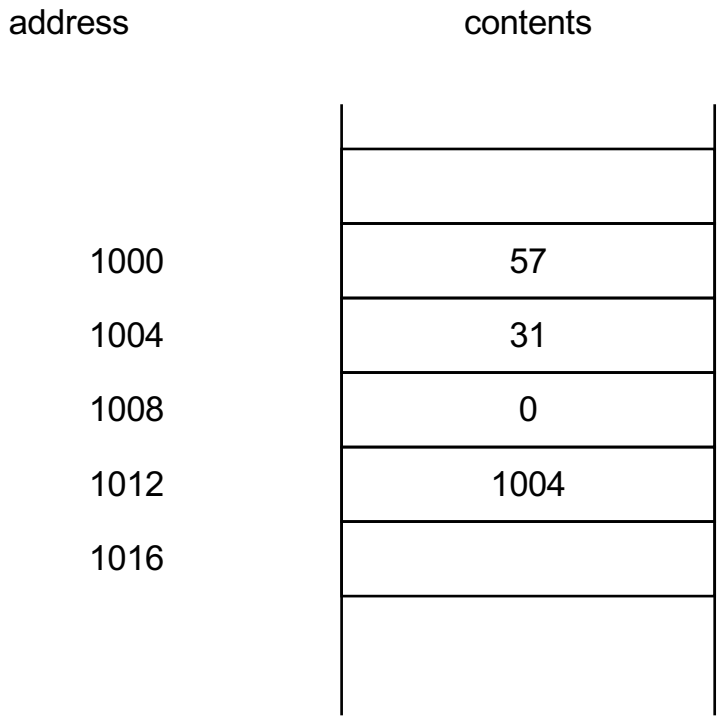
EE 441 Data Structures
Lecture 5
Pointers

İlkay Ulusoy

Memory

- The computer's memory is made up of bytes.
- Each byte has an address, associated with it.



- Different data types occupy different number of bytes in memory
- Examples:

Type Name	Bytes	Range of Values
char	1	–128 to 127
unsigned char	1	0 to 255
short	2	–32,768 to 32,767
unsigned short	2	0 to 65,535
Long / int	4	–2,147,483,648 to 2,147,483,647
unsigned long	4	0 to 4,294,967,295
enum	*	Same as int
float	4	3.4E +/- 38 (7 digits)
double	8	1.7E +/- 308 (15 digits)
long double	10	1.2E +/- 4932 (19 digits)

Variables

- A variable is a named object. It represents a storage location.

- A variable has the following attributes:
  1. name :label to identify a variable in the text of a program.
  2. address : the memory address of the storage location(s) occupied by that variable
  3. size :the amount of storage (in bytes) occupied by that variable.
  4. type :determines the set of values that the variable can have and the set of operations that can be performed on that variable.
  5. value :the content of the memory location(s) occupied by that variable.
  6. lifetime : the interval of time in the execution of a program during which a variable is said to exist.
  7. scope :the range of statements in the text of a program in which that variable can be referenced.

Variables

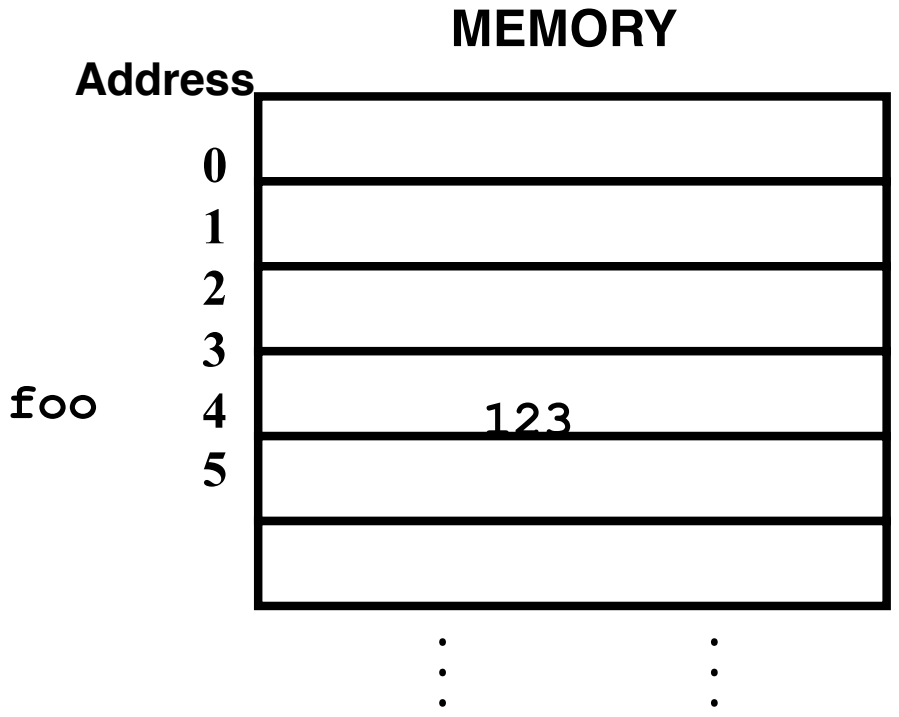
```
int i = 57;
```

- This statement defines a variable and *binds* various attributes with that variable.
- The name of the variable is i, the type of the variable is int, its size is `sizeof(int)` and its initial value is 57.
- Static binding: Attributes of a variable are bound at compile time.
- Dynamic binding: Attributes of a variable, such as its address and value, may be bound at run time.

Address-of

A pointer is a variable that holds the *address* of something else.

```
int foo;

foo = 123;
```

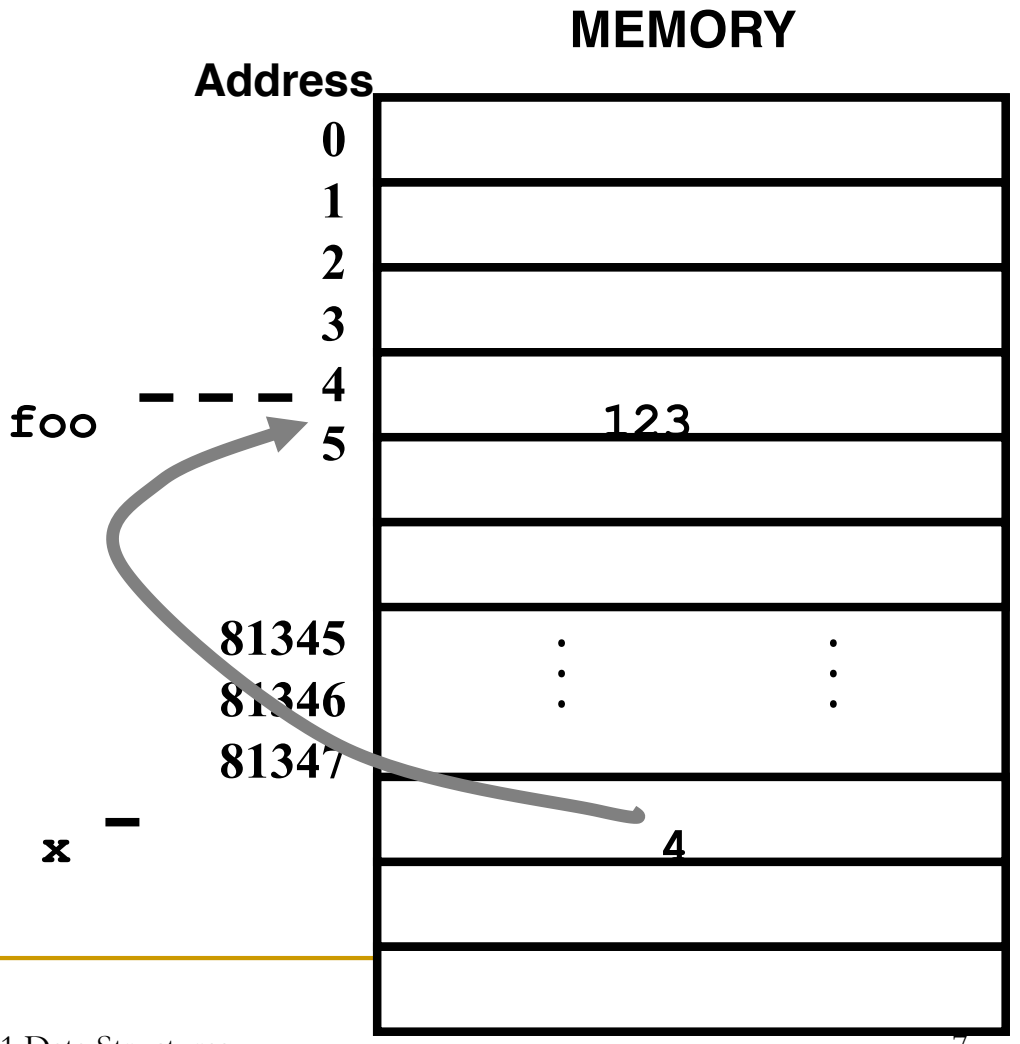


"the address of foo" = 4

Pointers

- In C++ you can define a pointer by “ * ” and get the address of a variable with the “&” operator.

```
int foo;
int *x;

foo = 123;
x = &foo;
```



Pointers (Examples)

e.g.

```
int *i1, i2;
```

here, i1 will contain a pointer to an integer

i2 will contain an integer

Similarly,

```
float *f1, *f2, f3;
```

f1 and f2 are pointers to floating point variables

f3 is itself a floating point variable

Some more:

```
int ip, *ip2;       // an integer and an integer pointer
long *lp, lp2;  // a long integer pointer and a long integer
float fp, *fp2; // a floating point and a floating point pointer
```

A pointer:

- An integer that represents a memory address.
- Serves as a reference to the data at the address.
- Holds the address of an object of the specified type
- A variable the type of which has the form *T** where *T* is an arbitrary type.

- Declaring a pointer:

`int* p;` p is a pointer to int.

  - The *value* of the int* pointer p is the address of another variable of type int.
  - Simple declaration does not give the pointer a value
  - Pointers do not initially reference an object in memory

- Why do we need pointers:
  - Through a pointer an object can be referenced directly
  - Dynamic memory management: the management of objects allocated during execution.
  -
- A typical use of pointers:
  - creation of linked lists
  - management of objects allocated during execution

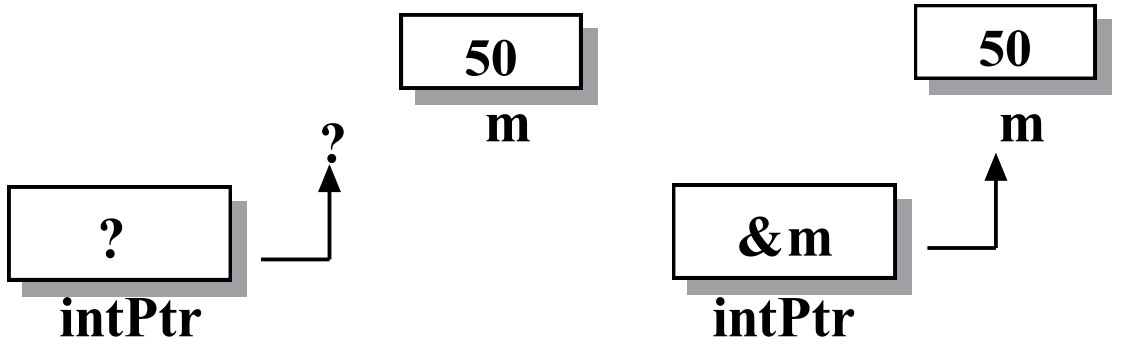
Pointers: Assigning a value

```
int m=50;
int* intPtr;
(int m, *intPtr; //is valid too)
```

- &m is the address of the integer in memory.

  `intPtr = &m;`

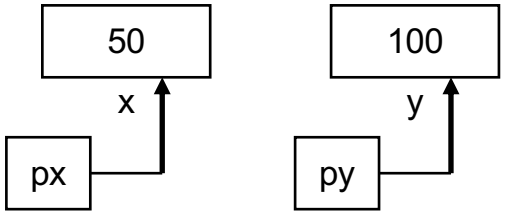
- sets intPtr to point at an actual data item.



(a) After declaration (b) After assignment

Pointers: Accessing data

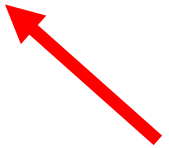
- “de-reference “operator(*): Allows access to the contents referenced by the pointer.
  - `*intPtr` is an alias form (alternative name) for `m`.
  - `*intPtr=60;` has the same effect as `m=60;`
- Examples:
  - `int x=50,y=100;`
  - `int *px, *py;`
  - `px =&x ,py =&y;`
  - `*px=x;`
  - `*py=y;`



Assigning Value to Pointers

- A pointer must have a value before you can *dereference* it (follow the pointer).

```
int *x;
*x=3;
```



ERROR!!!
x doesn't point to anything!!!

```
int foo;
int *x;
x = &foo;
*x=3;
```

this is fine
x points to foo

Pointers (Examples)

```
int i=856;
int *p=i;        // error! type mismatch.
int *p=&i;       // creates a pointer p that points to i
                 // address of integer i is stored in pointer p
int *p2=p;       // a second "pointer-to-integer" called p2
                 // is created and the address in p is copied
                 // into p2

int k=p;         // error! type mismatch
int k=*p;        // value pointed by p (i.e., 856) is copied into k
p2=&k;           // copy address of k into pointer p2
*p2=i;           // copy i into address pointed by p2
                 // i.e., value of k now becomes 856
```

Pointers (Examples)

e.g.:

```
*p=*p+2;      // add 2 to the value in the address pointed
              // by p (i.e., i now becomes 858)
 p=p+2;       // increment the address in p by 2
              // (p no longer points to i)
```

Hence:

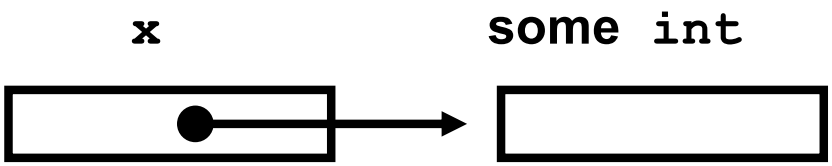
*p : “object pointed to by” p

&p : “address of” p

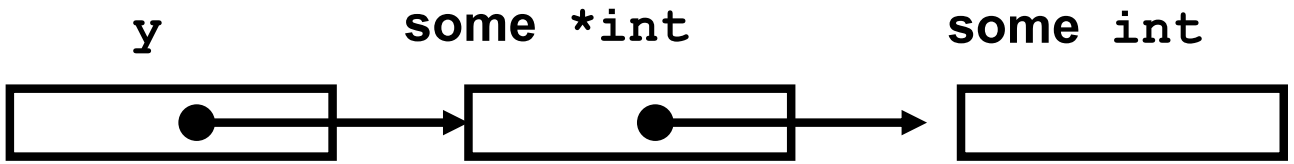
p : “the pointer itself”

Pointers to anything

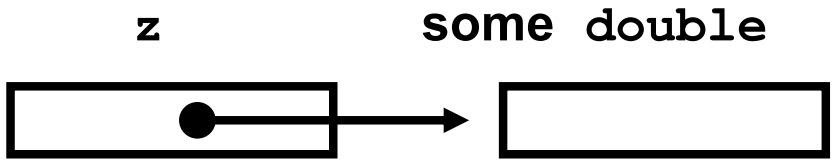
```
int *x;
```




```
int **y;
```

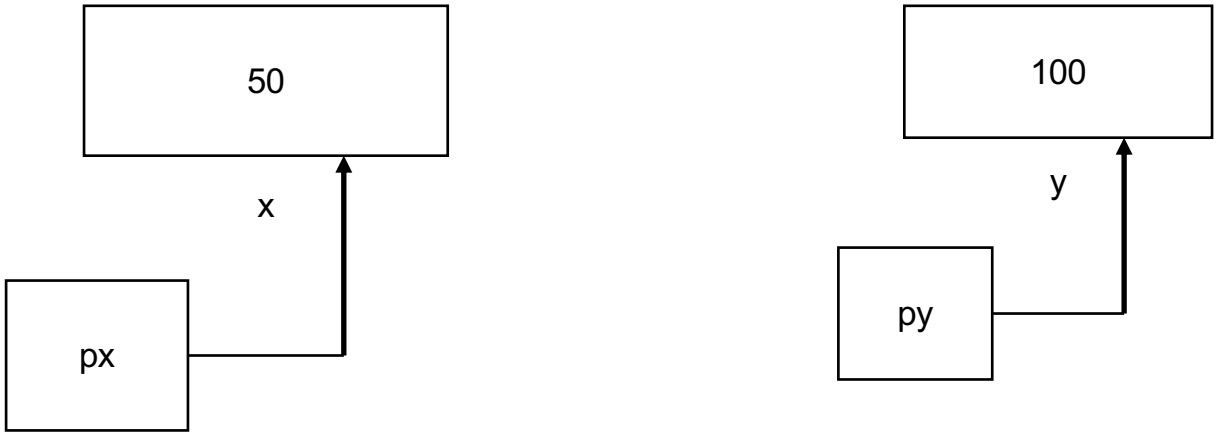



```
double *z;
```



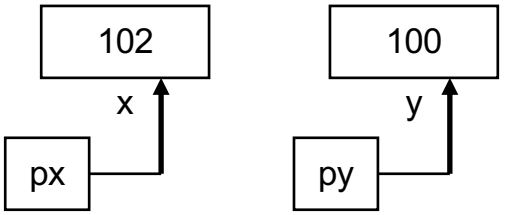
Pointers: Accessing data

```
int x=50, y=100;
int *px, *py;
px =&x;
py =&y;
```

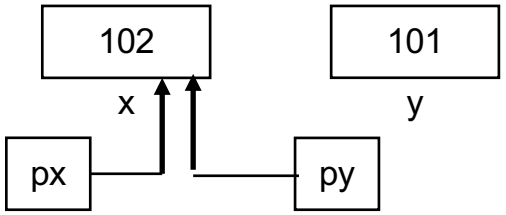


Pointers: Accessing data

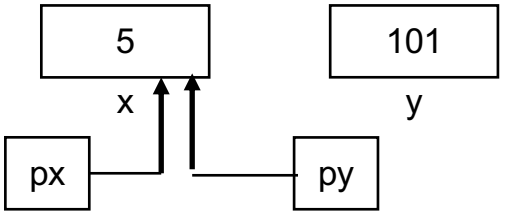
```
*px=*py+2;
```



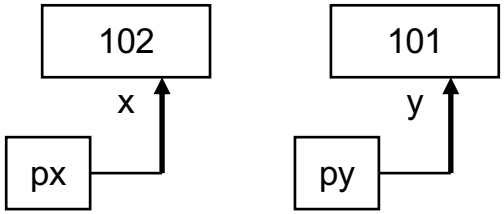

```
py=px;
```




```
*py=5;
```



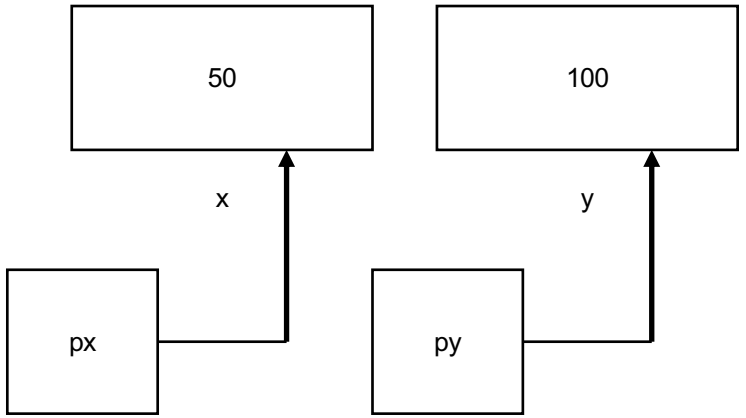

```
*py=(*py)++;
```




```
cout<<x;
5
```

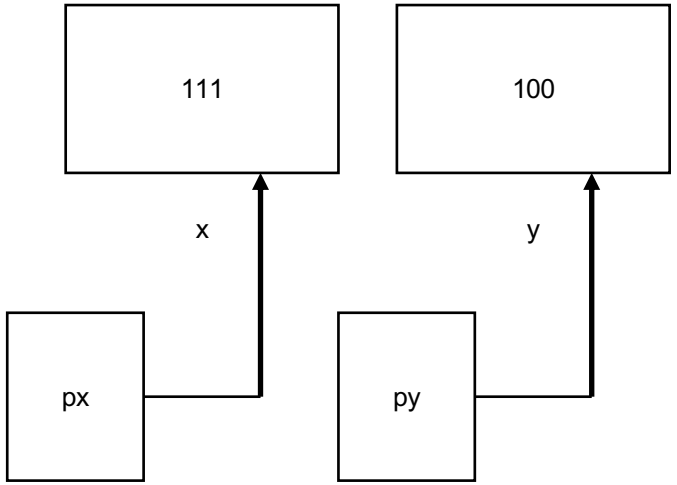
Pointers: Accessing data: Example

```
#include<stdio.h>
#include<iostream.h>
void main()
{

int *px, *py;
int x, y;

x=50;
y=100;
px=&x;
py=&y;
cout<<"x:"<<x<<" y:"<<y<<"\n";
cout<<"px:"<<px<<" py:"<<py<<"\n";
cout<<"*px:"<<*px<<"
   *py:"<<*py<<"\n";
cout<<"\n";
```



x:50 y:100
px:0x0012FF74 py:0x0012FF70
*px:50 *py:100

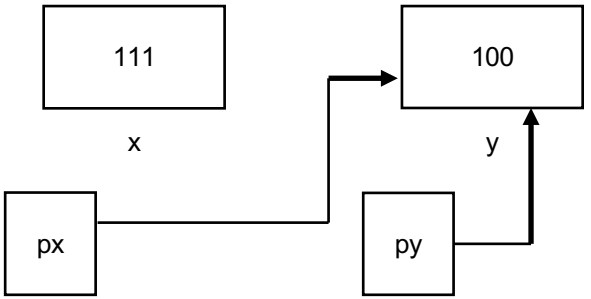
Pointers: Accessing data: Example




```
*px=y+11;
cout<<"x:"<<x<<" y:"<<y<<"\n";
cout<<"px:"<<px<<" py:"<<py<<"\n";
cout<<"*px:"<<*px<<"*py:"<<*py<<"\n";
cout<<"\n";
```

x:111 y:100
px:0x0012FF74 py:0x0012FF70
*px:111 *py:100

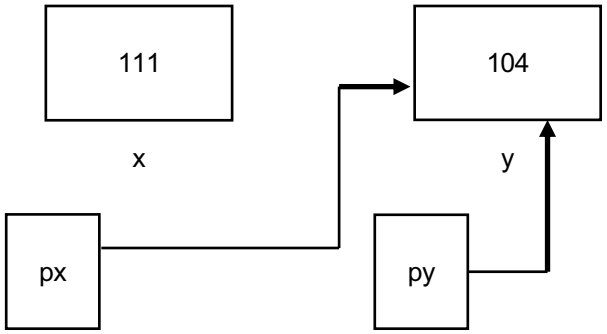
Pointers: Accessing data: Example

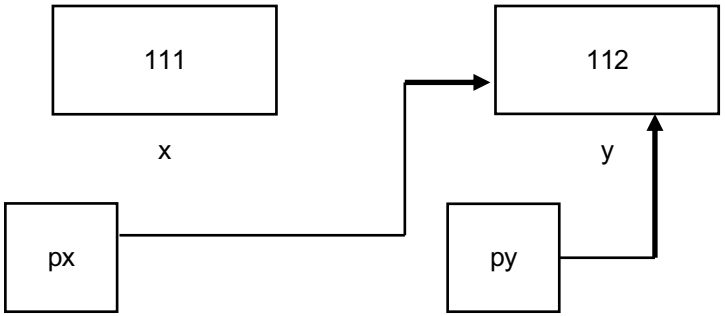



```
px=py;
cout<<"x:"<<x<<" y:"<<y<<"\n";
cout<<"px:"<<px<<" py:"<<py<<"\n";
cout<<"*px:"<<*px<<"*py:"<<*py<<"\n";
cout<<"\n";
```

x:111 y:100
px:0x0012FF70 py 0x0012FF70
*px:100 *py:100

Pointers: Accessing data: Example






```
*px=(*px)+4;
*py=(*py)+8;
cout<<"x:"<<x<<" y:"<<y<<"\n";
cout<<"px:"<<px<<" py:"<<py<<"\n";
cout<<"*px:"<<*px<<"*py:"<<*py<<"\n";
cout<<"\n";
}
```

x:111 y:112
px:0x0012FF70 py:0x0012FF70
*px:112 *py:112

Pointers: Accessing data: Example

```
int* ip // pointer declaration

int *ip // the same as above

int *ip1, *ip2 // two pointers

int* ip1, ip2 // a pointer, an integer

int ip, *ip2 // an integer and an integer pointer

long *lp, lp2 // a long integer pointer and a long
                //integer

float fp, *fp2 // a floating point and a floating
                //point pointer
```

```
int i=1024

int *ip=i // error, type mismatch

int *ip=&i // ok. the operator & is referred as
           //address-of operator

// create another pointer that also points to i
int *ip2=ip // ok. now ip2 also addresses i

// to create a pointer that points ip2
int *ip3=&ip2 // error, type mismatch

int **ip3=&ip2 // ok , it is a pointer to a pointer.
                  //Note that char
           //is 1,int 4 and double is 8 bytes long
```

Pointers: Accessing data: Example

```
int i=1024
int *ip=&i; //ip points to i

int k=ip; //error

int k=*ip; // k now contains 1024

int *ip=… // type decleration for the pointer and
             //initialization
*ip= … // the item pointed by ip

…=*ip // the item pointed by ip

=ip // the pointer itself
```

```
int k=-5

int *ip=&i;

*ip=k; // i=k

*ip=abs( *ip) // i=abs(i);

ip=ip+1//ip points to an integer that is 4 bytes so
      //increases the value of the address it contains
      // by the size of the object, so
      //it is incremented by 4
      //in some compilers ip+1*size of (int)
```

```
int i,j,k;

int *ip=&i;

*ip= *ip+2 // add two to i,
                //that is i=i+2
ip=ip+2; // add two to the address ip
          //contains.
//In some compilers ip +2*sizeof(int)
```

Specific C++ notations

Compound assignment:

```
int arraySum(int ia[ ], int sz)
{int sum=0;
for (int i=0; i<sz; ++i)
sum+=ia[i];
return sum;
}
```

```
sum+=ia[i];
 is same as
sum=sum+ia[i];
```

general syntax:
a op =b;
meaning
a=a op b;

so e.g. i*=j;
is the same as i=i*j;

e.g. ia[i]*=(b+c/d);
is the same as
ia[i]=ia[i]*(b+c/d);
etc.

Increment and decrement operators in the prefix and postfix forms:

Prefix: ++i

e.g. array [++i]=some_value;

prefix form increments i before using that value as index

Postfix: i++

e.g. array [i++]=some_value;

postfix form increments i after using that value as index

Note: alone i++; is the same as ++i;

E.g.

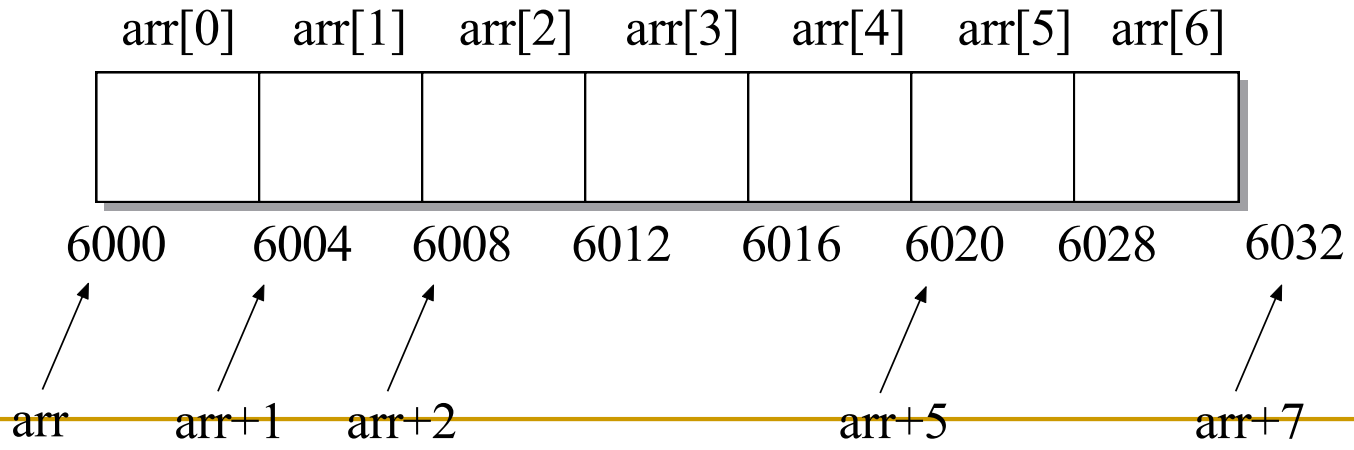
A[(*(*Q+1))++]=2.2 ????

Pointers and Arrays

- Array: Sequence of data items of the same type occupying consecutive memory locations.
- Array name is the starting address of the memory block

```
int arr[7];
```

- Compiler:
  - Allocates space for 7 integer objects
  - Assigns to `arr` the starting address of the memory block
  - Associates the pointer with the type and the size of the item it references
  - Knows that data item is a 4-byte integer



Pointers and Arrays: Example

```
void main()
{
int *p;
int arr[7];
int i;

for(i=0;i<7;i++)
   arr[i]=i;

for(i=0;i<7;i++)
   cout<<arr[i]<<" ";
cout<<"\n";

for(i=0;i<7;i++)
   cout<<&arr[i]<<" ";
cout<<"\n";
```

```
for(i=0;i<7;i++)
   cout<<*(arr+i)<<" ";
cout<<"\n";

p=arr;
for(i=0;i<7;i++)
{
   cout<<*p<<" ";
   p++;
}
cout<<"\n";
}
```

Pointers and Arrays: Example

```
void main()
{
int *p;
int arr[7];
int i;

for(i=0;i<7;i++)
   arr[i]=i;

for(i=0;i<7;i++)
   cout<<arr[i]<<" ";
cout<<"\n";

for(i=0;i<7;i++)
   cout<<&arr[i]<<" ";
cout<<"\n";
```

```
for(i=0;i<7;i++)
   cout<<*(arr+i)<<" ";
cout<<"\n";

p=arr;
for(i=0;i<7;i++)
{
   cout<<*p<<" ";
   p++;
}
cout<<"\n";
}
```

```
0 1 2 3 4 5 6
0x0012FF60
0x0012FF64
0x0012FF68
0x0012FF6C
0x0012FF70
0x0012FF74
0x0012FF78

0 1 2 3 4 5 6
0 1 2 3 4 5 6
```

Pointers and Arrays

- An array name is basically a *const* pointer.
- You can use the [ ] operator with a pointer:

```
int *x;
int a[10];
x = &a[2];
for (int  i=0;i<3;i++)
  x[i]++;
```

x is "the address of a[2] "

x[i] is the same as a[i+2]

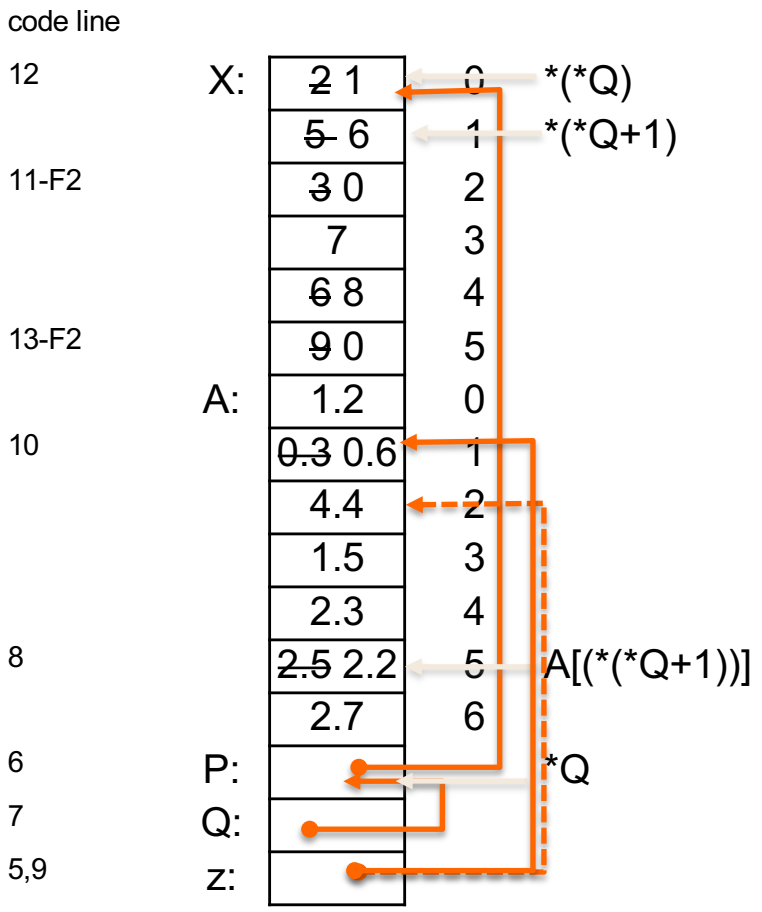
Ex: Write the contents of A and X at the end of execution of the following C++ program

```
      void  MyFUN(int* z)
F1.     {z += 2;
F2.       *z=0;  }

       main ()
1.   { int X[ ]={2,5,3,7,6,9}
2.   float A[ ]={1.2, 0.3, 4.4, 1.5, 2.3,2.5,2.7};
3.   int *P,**Q;
4.   float *z;

5.   z=&A[2];
6.   P=X;
7.   Q=&P;
8.   A[(*(*Q+1))++]=2.2;
9.   (*z)*=2;
10.   z-=1;
11.  MyFun(P);
12.  *P=1;
13.  MyFun(P+3);
14.  *(P+4)=8;
        }
```

Example

Write the contents of A and X at the end of execution of the following C++ program

```
        void  MyFUN(int* z)
F1.    {z += 2;
F2.     *z=0;
        }
        main ()
1.     { int X[ ]={2,5,3,7,6,9}
2.     float A[ ]={1.2, 0.3, 4.4, 1.5, 2.3,2.5,2.7};
3.     int *P,**Q;
4.     float *z;
5.     z=&A[2];
6.     P=X;
7.     Q=&P;
8.     A[(*(*Q+1))++]=2.2;
9.     (*z)*=2;
10.     z-=1;
11.    MyFun(P);
12.    *P=1;
13.    MyFun(P+3);
14.    *(P+4)=8;
        }
```



Solution

X: 1, 6, 0, 7, 8, 0

A:1.2, 0.6, 4.4, 1.5, 2.3, 2.2, 2.7

Information Passing Between Functions (Argument Passing)

- The "output" or return value of a function may (be of any type) belong to any class.

  e.g.

```
RandomNumber fr(int n)
{ // function fr returns an object that belongs to class RandomNumber;
  // input parameter "n" can have any use, to be defined within the
  // function }
```

- The input parameters (or "arguments") of a function may be passed
- BY VALUE

or

- BY REFERENCE.

Information Passing Between Functions (Argument Passing)

- BY VALUE: The calling program copies the actual full object value to the local data area of the called program.

- BY REFERENCE: The address of the object as stored by the calling program is passed into the called program which operates on the original data; not its local copy!!!

Argument Passing (Example)

- Suppose that we desire to swap the contents of two integer variables. Consider the following function:

```
void swap(int v1, int v2)
{
        int temp=v2;
        v2=v1;
        v1=temp;
}

int main()
{
        int i=10;
        int j=20;
        cout << "Before swap():\ti:"<<i<<"\tj:"<<j<<endl;
        swap(i,j);
        cout << "After swap():\ti:"<<i<<"\tj:"<<j<<endl;
}
```

Argument Passing (Example)

- When this program is executed the result is not as we desired, the output is

Before swap(): i:10 j:20

After swap(): i:10 j:20

- Why?

- Pass by Value:
  - When a function is called, the function gets a copy of the original argument.
  - This copy remains in scope until the function returns and is destroyed immediately afterwards.
  - Consequently, a function that takes value-arguments cannot change them, because the changes will apply only to local copies, not the actual caller's variables.
  - If you want the (function) callee to modify its arguments, you must override the default passing mechanism.
  - Before the variables inside the function have not been passed out, copies were swapped.
  -

- The parameters in the argument list must be passed by *reference*.

- There are two alternatives two solve this problem:
  - Use pointers as parameters
  - Use reference as parameters

Pointers as Parameters

- Alternative 1: Pass pointers to the variables

```
void pswap(int *v1, int *v2)     // input parameters shall be pointers
{
        int temp=*v2;
        *v2=*v1;
        *v1=temp;
}
void main()
{
        …
        pswap(&i,&j);            // caller passes address of i and j
        …
}
```

Pointers as Parameters

- When executed we obtain:

Before swap(): i:10 j:20

After swap(): i:20 j:10

If we change what the pointer points to, the caller will see the change!!

Call by Reference

- Alternative 2: Another syntax to pass pointers

```
// input parameters are variables but compiler should copy addrs. only

void rswap(int &v1, int &v2)
{
        int temp=v2;
        v2=v1;
        v1=temp;
}
void main()
{
        …
        rswap(i,j);      // caller passes variables themselves,
        …                // compiler converts this to copying addresses
}
```

When executed, correct results will be obtained!!!

References

- A reference is "an alias for an existing object."

```
int m=0;
int &ref=m;
```

  The reference ref serves as an alias for the variable m.

- ref and m behave as distinct names of the same object.

- Any change applied to m is reflected in ref and vice versa.

- You may define an infinite number of references to the same object:

```
int & ref2=ref;
int & ref3=m;
```

- Here ref2 and ref3 are aliases of m, too.

- There's no such thing as a "reference to a reference;
- References:
  - Syntactically behave like ordinary variables
  - Function as pointers from a compiler's point of view
  - Enable a (function) callee to alter its arguments without forcing programmers to use the unwieldy *, & and -> notation

Advantages of Pass by reference

- Combines the benefits of passing by address and passing by value.
-
- It's efficient, just like passing by address because the callee doesn't get a copy of the original value but rather an alias thereof (under the hood, all compilers substitute reference arguments with ordinary pointers).
- In addition, it offers a more intuitive syntax and requires less keystrokes from the programmer.
- Finally, references are usually safer than ordinary pointers because they are always bound to a valid object -- C++ doesn't have null references so you don't need to check whether a reference argument is null before examining its value or assigning to it.
- Passing objects by reference is usually more efficient than passing them by value because no large chunks of memory are being copied and no constructor and destructor calls are performed in this case.