EE 441 Data Structures

Lecture 2:

Arrays, Pointers, Argument Passing

Memory

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

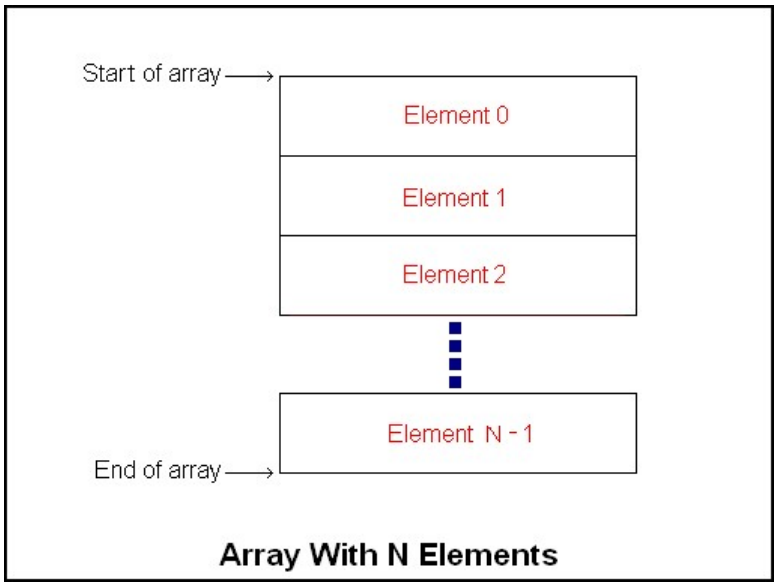
address	contents
1000	57
1004	31
1008	0
1012	1004
1016	

- 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	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)

Arrays

- An array is a consecutive group of memory locations (i.e., elements of the array).
- The contents of each element are of the same type.
  - Could be an array of int, double, char, …
- We can refer to individual elements by giving the position number (index, subscript) of the element in the array.



Arrays

- Arrays in C++ are indexed from zero and *not one!!*
  - *The first element is the 0th element !*
  - If you declare an array of *n* elements, the last one is number *n-1*.
  - *If you try to access element number n it is an error !*
- Subscripts are always integral types, e.g.,

```
int A[10];
```

- The number of elements in the array must be known at compile time

Declaring An Array

```
element_type array_name[number_of_elements];
```

element_type can be any C++ data type.

array_name can be any valid object name.

number_of_elements can be an expression.
Cannot be a variable that can be changed during runtime.

C++ Operations on Arrays

- Declaration:

```
const int ArraySize=50; //const variable
  cannot be changed runtime
float  A[ArraySize];
long  X[Arraysize+10];
Date Calendar[365];
```

- Assignment:

```
A[i]=z;
t=X[i];
x[i]=x[i+1]=t ;
/*this is equivalent to x[i+1]=t;
  x[i]=x[i+1];i.e., right to left*/
```

Initialization

- You can initialize an array when you declare it (just like with variables):

```
int grades[5] = { 1,8,3,6,12};
double d[2] = { 0.707, 0.707};
char s[] = { 'M', 'E', 'T', `U' };
```

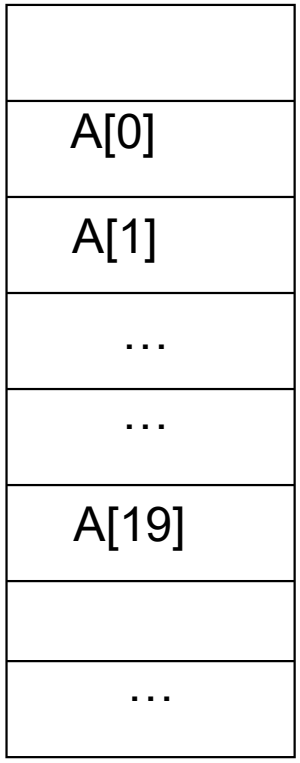
You don't need to specify a size when initializing, the compiler will count for you.

Array Example

```
int main(void)
{
  int facs[10];

  for (int i=0;i<10;i++)
     facs[i] = factorial(i);

  for (int i=0;i<10;i++)
     cout << "factorial(" << i << ") is " <<
          facs[i] << endl;
}
```

C++ Operations on Arrays

Most C++ compilers don't check array index range !!




```
int V=20;
int A[20]; /*  index range
  is 0-19  */
A[V]=0; /* index is out of
range, but most C++ compilers
don't check this */
```

Two Dimensional Arrays

```
int T[3] [4]={{20,5,-3,0},{-85,35,-
  1,2},{15,3,-2,-6}};
```

T:

20	5	-3	0	T[0]
-85	35	-1	2	T[1]
15	3	-2	-6	T[2]

e.g., `T[2] [3]=-6, T[2][0]=15` etc.

e.g., `int T[][5]`: a list of 5-element arrays.

Arrays of Objects

```
Rectangle room[100];
// constructor is called for room[0] ..room[99];
```

- This declaration creates an array of 100 objects. Each have different members, all (in this case) initialized to the default values.
- For declaring large array objects, a constructor with default values or with no parameters is preferred!
- Rectangle room[3]={Rectangle(10,15), Rectangle(5,8), Rectangle(2,30)}; may be practical, but
- Rectangle room[100]; simply initializes all length and with values to the default value of 0.

Variables: Example

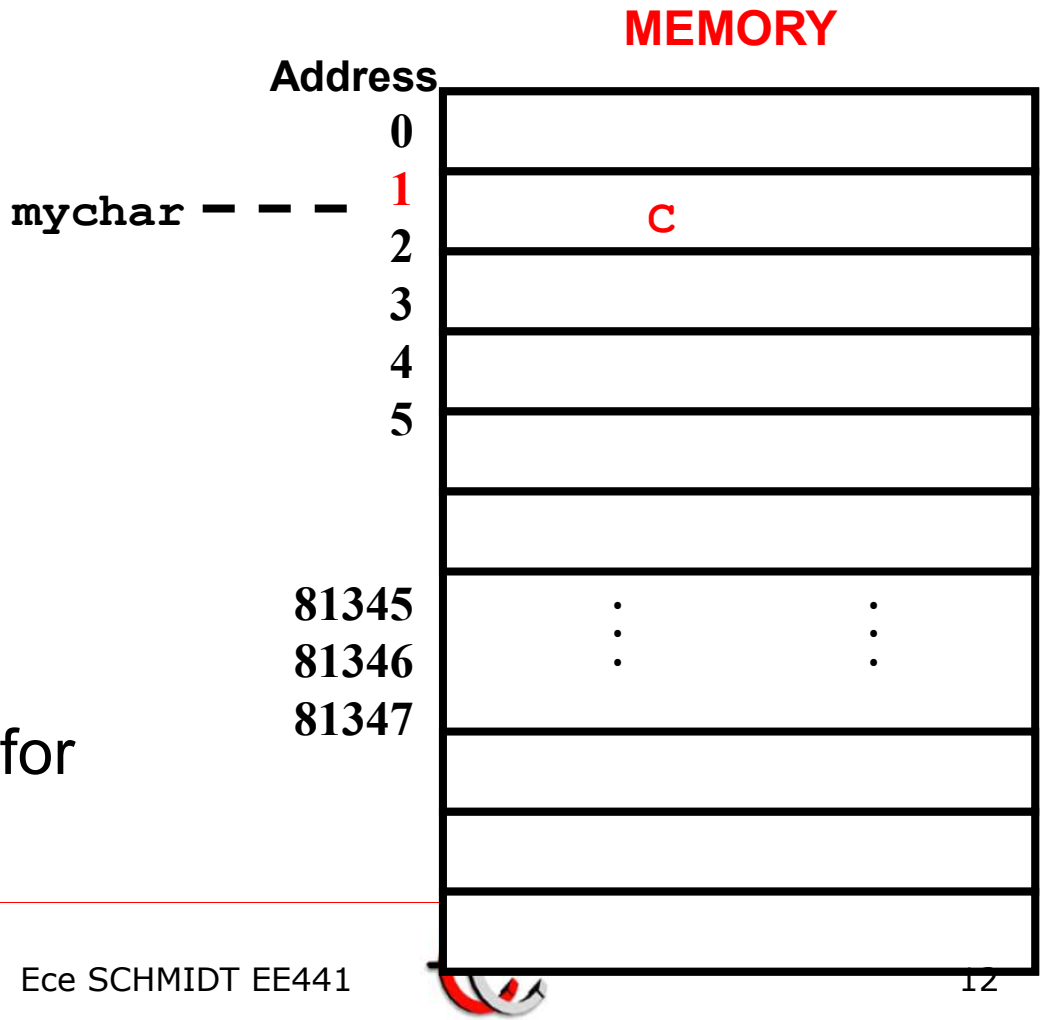
Declare a character type variable:

```
char mychar='C';
```

Allocated memory for **`mychar`** is at Location 1

Amount of memory allocated for **`mychar`** = **`sizeof(char)`**

Location 1 contains the binary representation for character **`C`**



Variables: Example

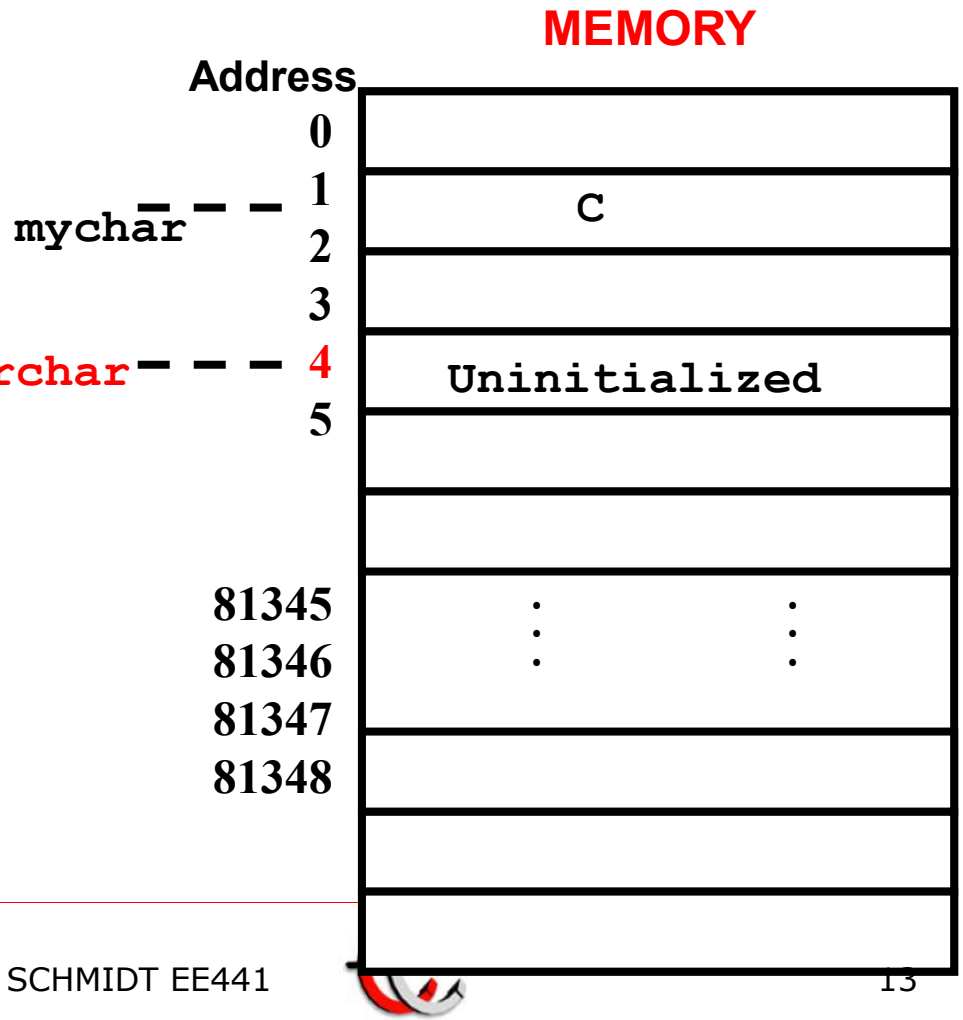
Declare another character type variable:

```
char yourchar;
```

Allocated memory for **`yourchar`** is at Location 4

Location 4
Is uninitialized

The programmer has no control on the selected location. You cannot tell the computer to store **`mychar`** or **`yourchar`** at location 5.



Pointers: Example

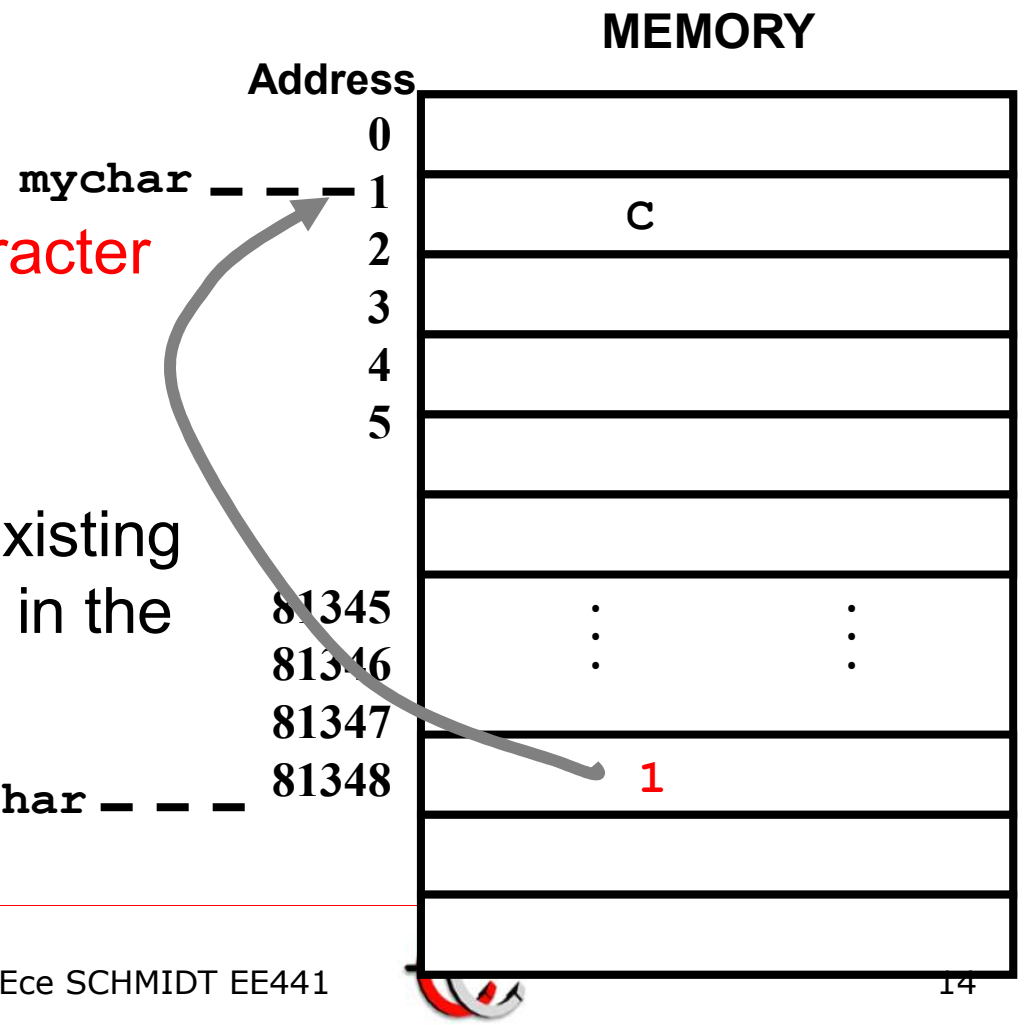
A pointer is a variable that holds the *address* of another object.

Declare a pointer to character type variable:

```
char* pchar;
```

Store the address of an existing variable of the same type in the pointer:

```
pchar=&mychar;
```



Pointers: Example

The variable of pointer changes according to the type of the pointed object

```
char* pchar;//pchar is a variable.
  Its type is pointer to character
int* intaddres;//intaddress's type
  is pointer to integer
pchar=intaddress;//will not work
  type mismatch!!
```

Pointers

- Typical uses of pointers are the creation of linked lists and the management of objects allocated during execution.

```
int* ip;                  // pointer decleration
int *ip;                  // the same as above
int *ip1, *ip2; // two pointers
int *ip1, ip2;            // a pointer, an integer
```

Pointers: Assigning a value

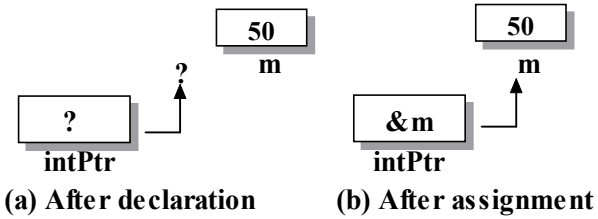
- "address-of" operator(&): Retrieves the memory address of a variable or object declared in the program.
- Example:

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

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

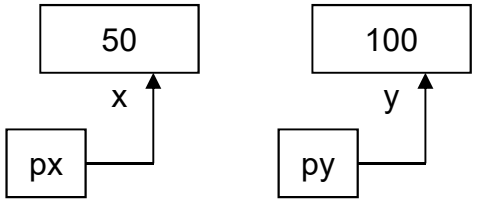
```
intPtr = &m;
```

- sets intPtr to point at an actual data item.

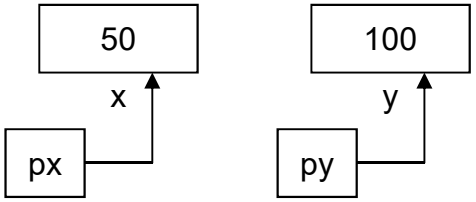


Pointers: Accessing data

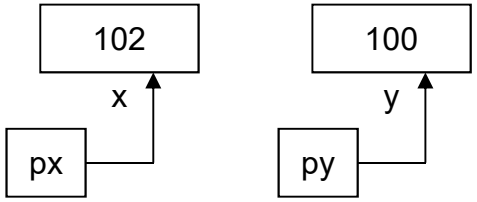
- "de-reference "operator(*): Allows access to the contents referenced by the pointer.
  - `*intPtr` is an alias form (alternative name) for `m`.
  - `*intPtr=60;` has the same effect as `m=60;`
- Examples:
  - `int x=50,y=100;`
  - `int*px =&x ,int*py =&y;`



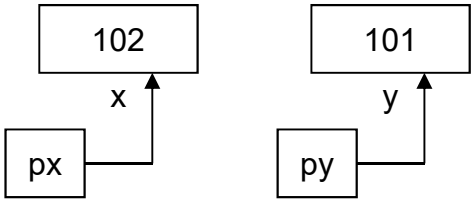
Pointers: Accessing data



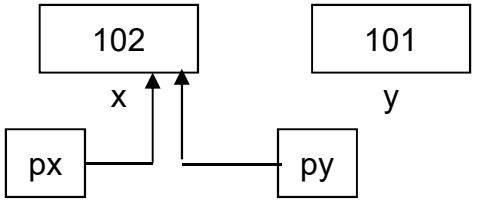

```
*px=*py+2;
```



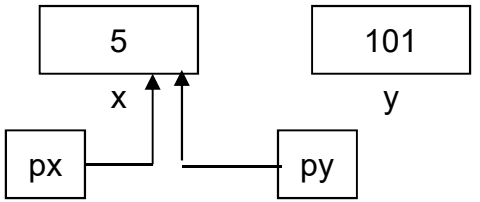

```
*py=(*py)++;
```




```
py=px;
```




```
*py=5;
```




```
cout<<x;
```

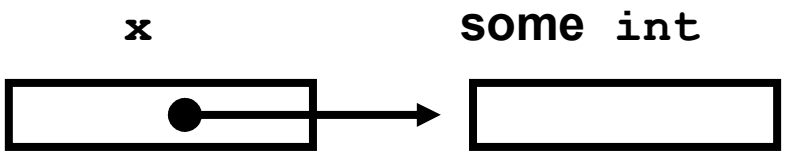
5

```
cout<<px<<" "<<py;
```

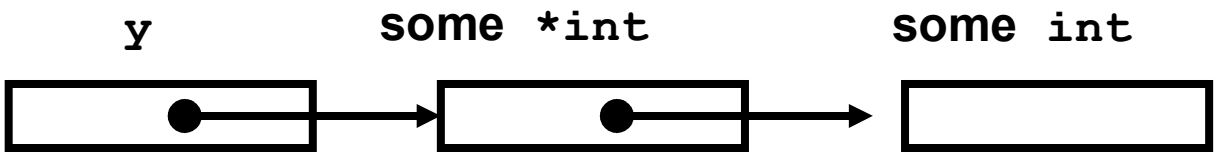
0x0045A7D0 0x0045A7D0

Pointers: Accessing data

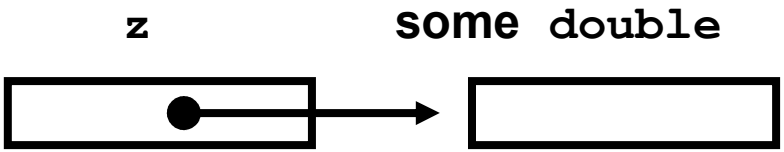
```
int *x;
```




```
int **y;
```




```
double *z;
```



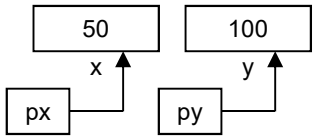
Pointers: Accessing data: Example

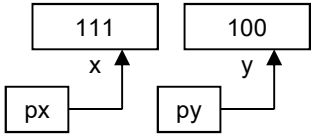
```
 int i=1024
//create an pointer that
   points to i

int *ip=i
// error, type mismatch


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

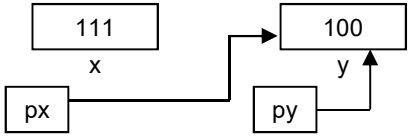
```
int *ip2=ip
// ok. now ip2 is another
   pointer that also
   addresses i

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
```

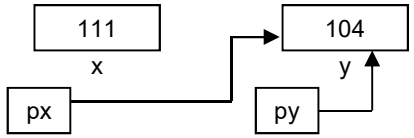
Pointers: Accessing data: Example RUN IT @home

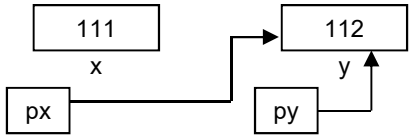
```
#include<iostream>
int main()
{

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";

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

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


*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";
return 0;
}
```











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

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

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

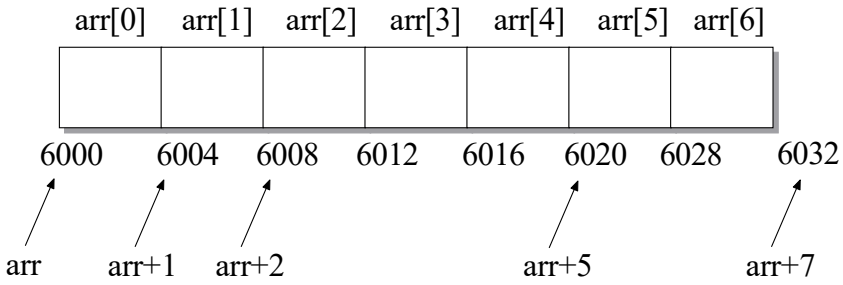
x:111 y:112
px:0x0012FF70 py:0x0012FF70
*px:112 *py:112

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];
```

- Runtime, the computer:
  - 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



- Pointer arithmetic:

`arr+i`

Points at the i^{th} location of the array

Pointers and Arrays: Example

```
main()
{

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";
```

```
int *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
```

Argument passing

- Write a function that takes:
  - Two integers as input
  - Swaps the integer values

```
swap(integer 1, integer 2)
```

- Desired operation

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

Before swap(): i:10 j:20
After swap(): i:20 j:10

Argument passing

- C++ offers three ways for argument passing:
  - pass by value
  - pass by address
  - pass by reference

Argument passing: Pass by value

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

Before swap(): i:10 j:20
After swap(): i:10 j:20

- Pass by Value:
  - Default argument passing mechanism
  - When a function is called the function makes a local 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.

Argument passing: Pass by address

```
void  pswap(int *v1, int *v2)
   // parameters are pointers
     {  int tmp=*v2;
        *v2=*v1;
        *v1=tmp;
     }
     main ()
     {.....
        pswap(&i, &j); // send
   address of i and j as
   parameter
     ...
     }
```

Before swap(): i:10 j:20
After swap(): i:20 j:10

- passing the *argument's address* to the callee.
- The function makes a local copy of the *address*
- Makes the changes on the original data stored in the address location
- The problem with this technique is that it's tedious and error-prone.

Argument passing: Pass by reference

```
void rswap(int &v1, int &v2)
    { int tmp=v2;
       v2=v1;
       v1=tmp;
    }

    main ()
    {.....
       rswap(i, j);
    ...
    }
```

When compiled and executed, we will have
Before swap(): i:10 j:20
After swap(): i:20 j:10

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 difficult *, & and -> notation

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;" the variable ref2 is an alias of m, not ref.

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).
- It offers a more intuitive syntax and requires less keystrokes from the programmer.
- 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.

Examples

```
#include <iostream>
main ()
{ int i;
  int numbers[6];
  int * p;
  p = numbers;
  for (i=0; i<6;i++){
 *(p+i)=i;
  }
  p = &numbers[2];  *p = 29;
  p = numbers + 3;  *p = 17;

  p= numbers; *p = 21;
  p++;  *p = 32;
  p = numbers;  *(p+4) = 16;
  for (int n=0; n<6; n++)
    cout <<"Line"<<n<<":"<<
numbers[n] <<"\n";
}
```

0	1	2	3	4	5

0	1	29	3	4	5

0	1	29	17	4	5

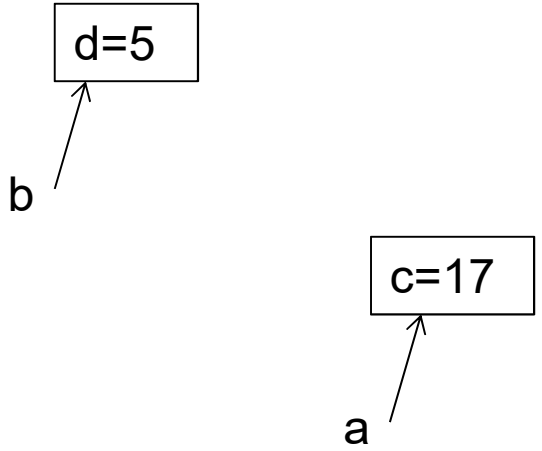
21	1	29	17	4	5

21	32	29	17	4	5

21	32	29	17	16	5

Examples

```
#include<iostream>
main()
{
int *a, c, d=5;
int *b=&d;
c=2;
a=&c;
*a=d+12;
cout<<"*a:"<<*a<<"*b:"<<*b<<"\n";
cout<<"c:"<<c<<"d:"<<d<<"\n";
cout<<"a:"<<a<<"b:"<<b<<"\n";
a=b;
*b=*a+5;
cout<<"*a:"<<*a<<"*b:"<<*b<<"\n";
cout<<"c:"<<c<<"d:"<<d<<"\n";
cout<<"a:"<<a<<"b:"<<b<<"\n";
}
```



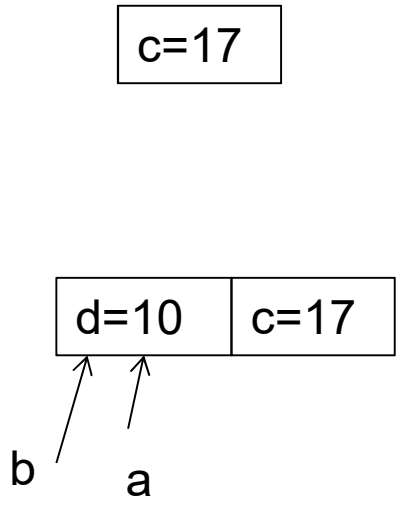
*a:17 *b:5

c:17 d:5

a:addres1 b:address2

Examples

```
#include<iostream>
main()
{
int *a, c, d=5;
int *b=&d;
c=2;
a=&c;
*a=d+12;
cout<<"*a:"<<*a<<"*b:"<<*b<<"\n";
cout<<"c:"<<c<<"d:"<<d<<"\n";
cout<<"a:"<<a<<"b:"<<b<<"\n";
a=b;
*b=*a+5;
cout<<"*a:"<<*a<<"*b:"<<*b<<"\n";
cout<<"c:"<<c<<"d:"<<d<<"\n";
cout<<"a:"<<a<<"b:"<<b<<"\n";
}
```



*a:10*b:10
c:17d:10
a:address b:address

Examples

```
#include <iostream>
void triple(double &num)
{
        num=3*num;
}
main()
{
        double d=10.0;
        triple(d);
        cout<<d;
        return 0;
}
```

Output: 30

```
#include <iostream.h>
void triple(double &num)
{
        num=3*num;
}


main()
{
        double d=10.0;
        triple(&d);
        cout<<d;
        return 0;
}
```

Error type mismatch in argument

Examples

```
#include <iostream>
void triple(double *num)
{
        *num=3**num;
}
main()
{
        double d=10.0;
        triple(&d);
        cout<<d;
        return 0;
}
```

Output: 30

```
#include <iostream.h>
void triple(double num)
{
        num=3*num;
}
main()
{
        double d=10.0;
        triple(d);
        cout<<d;
        return 0;
}
```

Output:10

Useful: Arrays and Characters

Extra Notes

Character Sequences:

As you may already know, the C++ Standard Library implements a **string** class, which is useful to handle and manipulate strings of characters. However, because strings are in fact sequences of characters, we can represent them also as plain arrays of char elements.

For example, declare Jenny as an array that can store up to 20 elements of type char.

char Jenny [20];

It can be represented as:

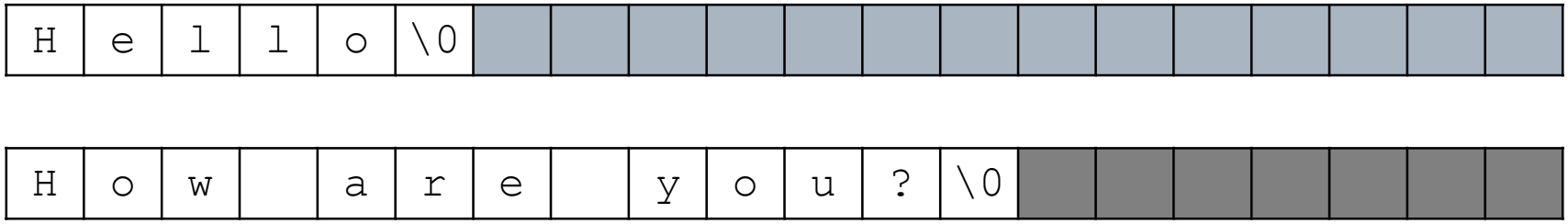
Jenny:

In the array Jenny, in theory, we can store sequences of characters up to 20 characters long. But we can also store shorter sequences. For example, Jenny could store at some point in a program either the sequence "Hello" or the sequence ”How are you?", since both are shorter than 20 characters.

Therefore, since the array of characters can store shorter sequences than its total length, a special character is used to signal the end of the valid sequence: the *null character*, whose literal constant can be written as '\0' (backslash, zero).

Our array of 20 elements of type char, called jenny, can be represented storing the characters sequences "Hello" and ”How are you?" as:

Jennny:



Notice how after the valid content a null character ('\0') has been included in order to indicate the end of the sequence. The panels in gray color represent char elements with undetermined values.

Initialization of null-terminated character sequences

Because arrays of characters are ordinary arrays they follow all their same rules. For example, if we want to initialize an array of characters with some predetermined sequence of characters we can do it just like any other array:

```
char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

In this case we would have declared an array of 6 elements of type char initialized with the characters that form the word "Hello" plus a null character '\0' at the end.

But arrays of char elements have an additional method to initialize their values: using string literals.

Double quoted strings (") are literal constants whose type is in fact a null-terminated array of characters. So string literals enclosed between double quotes always have a null character ('\0') automatically appended at the end.

Therefore we can initialize the array of char elements called myword with a null-terminated sequence of characters by either one of these two methods:

```
char myword [ ] = { 'H', 'e', 'l', 'l', 'o', '\0' };
char myword [ ] = "Hello";
```

In both cases the array of characters myword is declared with a size of 6 elements of type char: the 5 characters that compose the word "Hello" plus a final null character ('\0') which specifies the end of the sequence and that, in the second case, when using double quotes (") it is appended automatically.

Notice that we are talking about initializing an array of characters in the moment it is being declared, and not about assigning values to them once they have already been declared. In fact because this type of null-terminated arrays of characters are regular arrays we have the same restrictions that we have with any other array, so we are not able to copy blocks of data with an assignment operation.

Assuming mystext is a char[ ] variable declared previously. Assignment operations within a source code like:

```
mystext = "Hello";
```

```
mystext[ ] = "Hello";
```

would **not** be valid, like neither would be:

```
mystext = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

The reason for this may become more comprehensible once you know a bit more about pointers, since then it will be clarified that an array is in fact a constant pointer pointing to a block of memory.

- # Using null-terminated sequences of characters

Null-terminated sequences of characters are the natural way of treating strings in C++, so they can be used as such in many procedures. In fact, regular string literals have this type (char[ ]) and can also be used in most cases.
For example, cin and cout support null-terminated sequences as valid containers for sequences of characters, so they can be used directly to extract strings of characters from cin or to insert them into cout. For example:

```
// null-terminated sequences of characters
#include <iostream>;
void main ()
{ char question[] = "Please, enter your first name: "; char greeting[] =
"Hello, ";
char yourname [80];
 cout << question; cin >> yourname;
cout << greeting << yourname << "!";}
```

```
a sample run:
Please, enter your first name: John
Hello, John!
```

As you can see, we have declared three arrays of char elements. The first two were initialized with string literal constants, while the third one was left uninitialized. In any case, we have to specify the size of the array: in the first two (question and greeting) the size was implicitly defined by the length of the literal constant they were initialized to. While for yourname we have explicitly specified that it has a size of 80 chars.

Finally, sequences of characters stored in char arrays can easily be converted into string objects just by using the assignment operator:

```
string mystring;
char myntcs[ ]="some text";
mystring = myntcs;
```

EE 441 Data Structures

Lecture 2:

Arrays, Pointers, Argument Passing