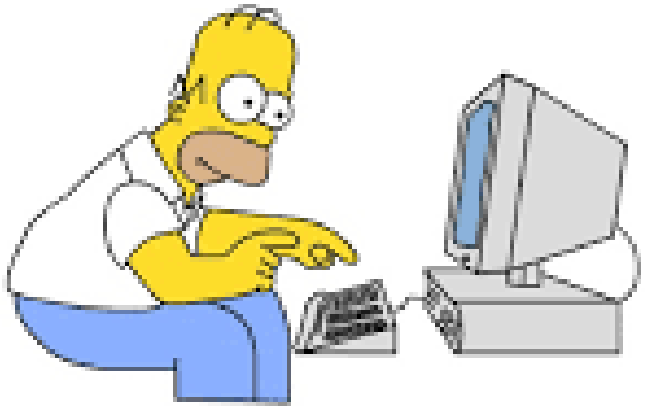
- Pointers
- Dynamic Variables
- Memory Management
- Pointers and Arrays
- Dynamic Arrays
- Pointers to structs

FUNDAMENTALS OF STRUCTURED PROGRAMMING

Pointers

Quote of the Day!

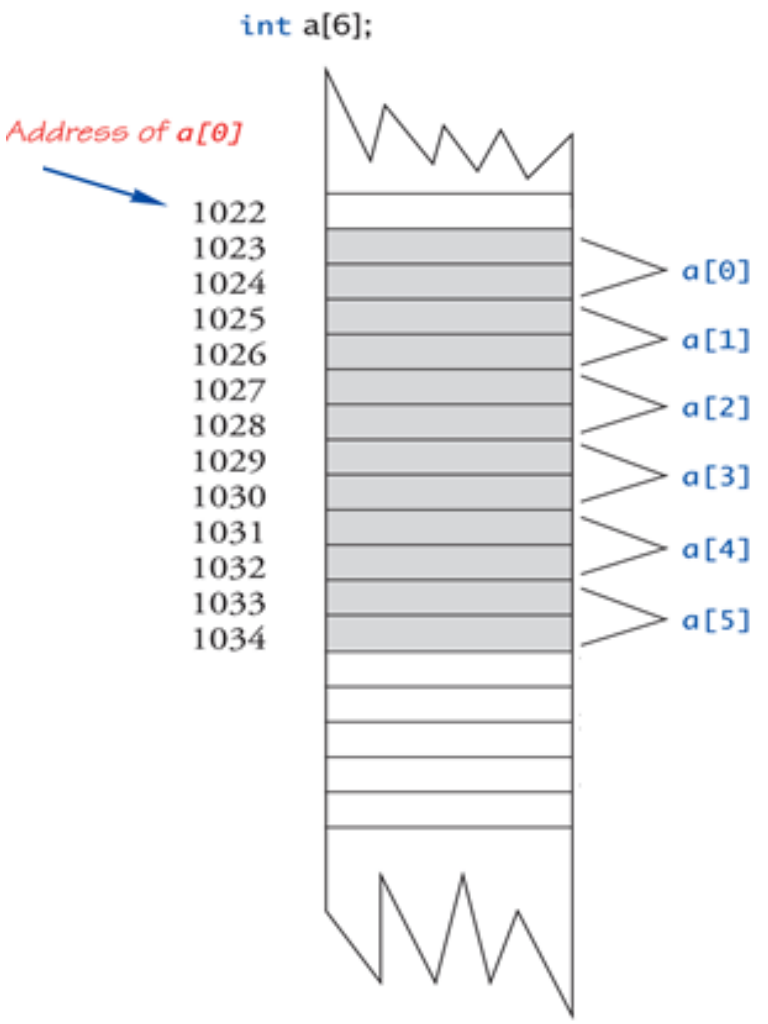
Any fool can write code that a computer can understand. Good programmers write code that humans can understand.



-Martin Fowler-

1. Pointers

- A pointer is the memory address of a variable.
- A pointer variable holds a pointer value. A pointer value is the address of a variable in memory.
- Pointer variables are typed.



1. Pointers – (cont.)

Declaration

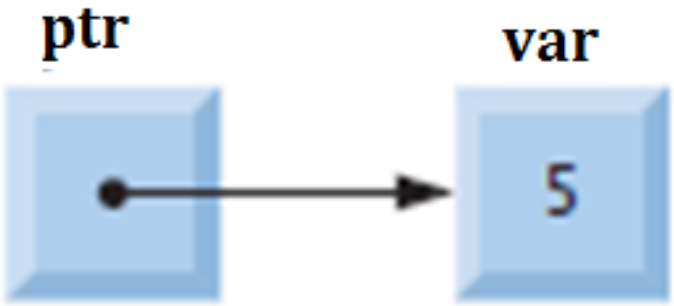
- Pointers declared like other types.
  - Add * before variable name. `int *p;`
  - Produces "pointer to" that type.
- * must be before each pointer variable.

```
int count = 7;
int *p1, *p2, v1, v2;   double *p;
```

- p1 and p2 hold pointers to int variables.
- count, v1, and v2 are ordinary int variables.
- p holds pointer to a double variable.

1. Pointers – (cont.)

- Diagrams typically represent a pointer as an arrow from the variable that contains an address to the variable located at that address in memory.
- A variable name directly references a value, and a pointer indirectly references a value (indirection).



- Terminology:

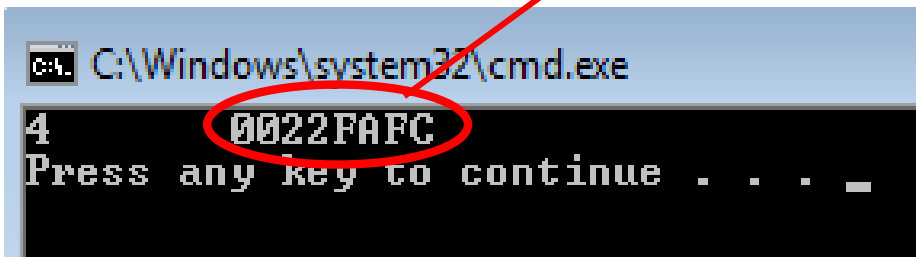
  Pointer variable "points to" ordinary variable.

1. Pointers – (cont.)

(Initialization) Address operator

```
// variables
int num = 4;
int* ptr = & num;

// output
cout<<num<<"\t"<<ptr<<endl;
```

A HEX representation of a memory location

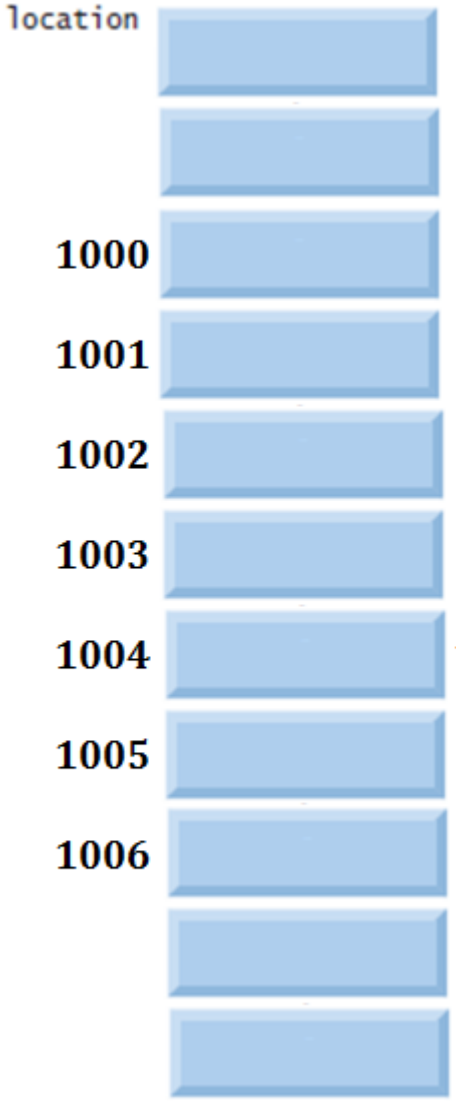


- The address operator **&** determines "address of" variable.
- Read like this:
  - "ptr equals address of num" Or "ptr points to num"

1. Pointers – (cont.)

Address operator

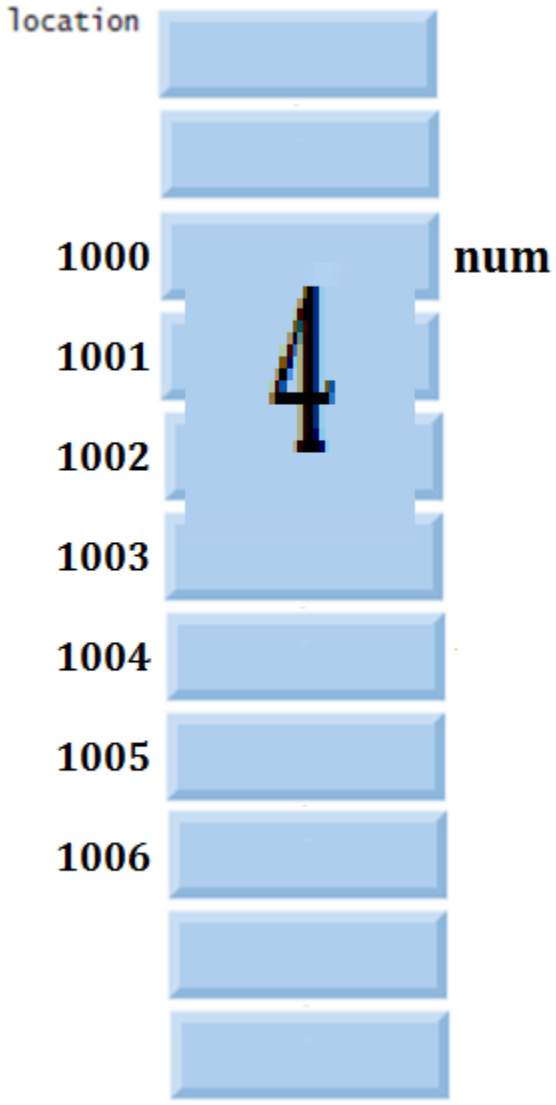
```
// variables
int num = 4;
int* ptr = & num;

// output
cout<<num<<"\t"<<ptr<<endl;
```



1. Pointers – (cont.)

Address operator

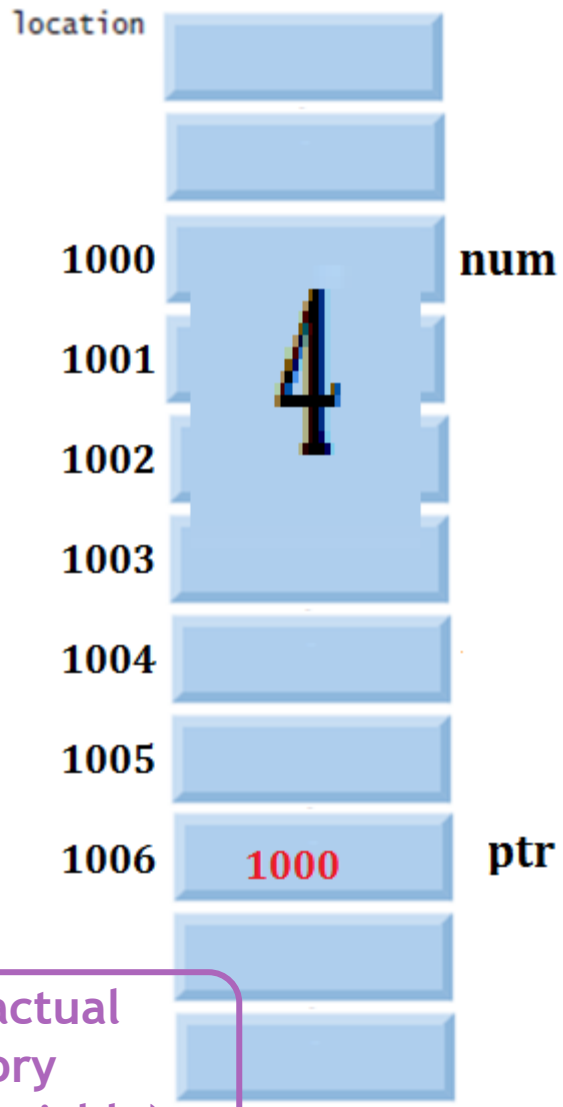
```
// variables
int num = 4;
int* ptr = & num;

// output
cout<<num<<"\t"<<ptr<<endl;
```



1. Pointers – (cont.)

Address operator

```
// variables
int num = 4;
int* ptr = & num;

// output
cout<<num<<"\t"<<ptr<<endl;
```

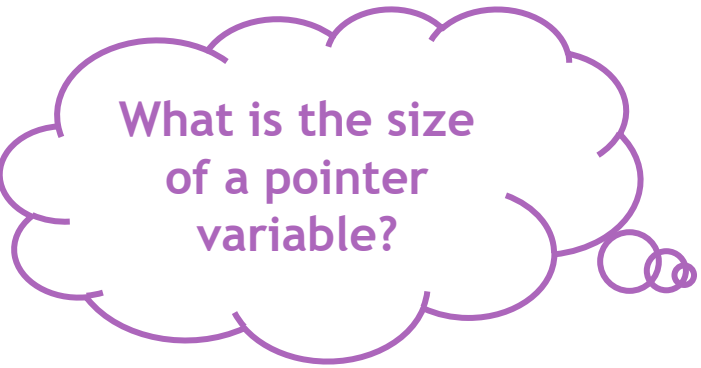


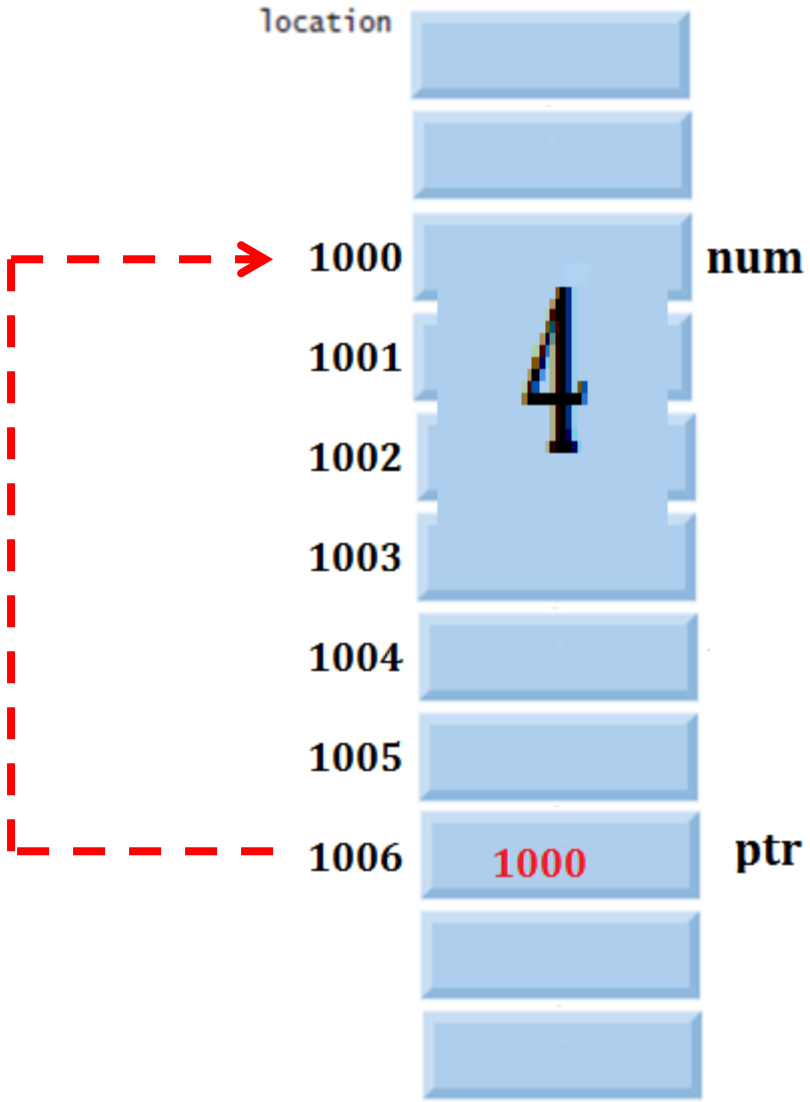
Note that ptr is an actual variable in memory (Unlike a reference variable)

1. Pointers – (cont.)

Address operator

```
// variables
int num = 4;
int* ptr = & num;

// output
cout<<num<<"\t"<<ptr<<endl;
```

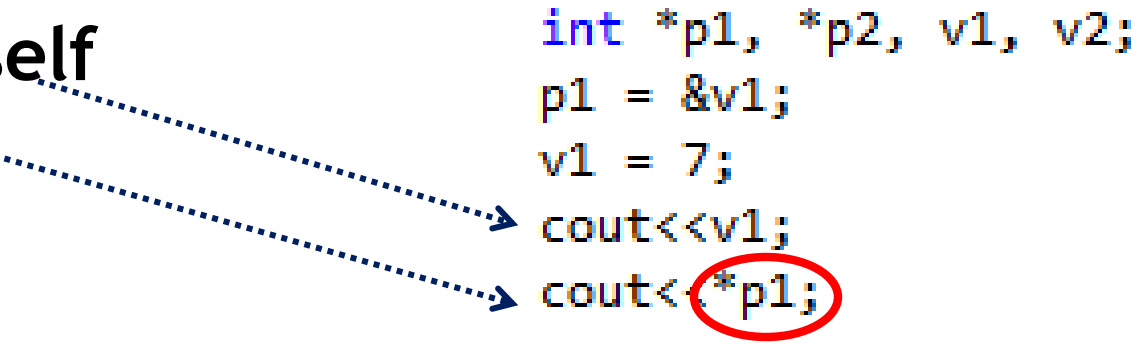




1. Pointers – (cont.)

Dereferencing operator

- Two ways to refer to v1 now:
  - Variable v1 itself
  - Via pointer p1



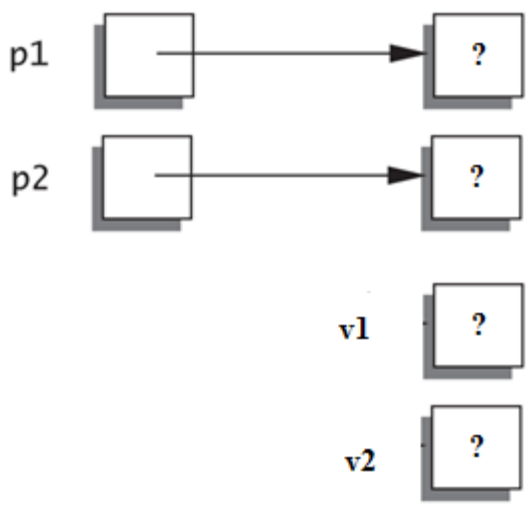
NOTE that this is NOT a declaration.

- The dereference operator * retrieves the value pointed to by the variable.
  - Pointer variable "dereferenced" means: "get data that pointer points to".

1. Pointers – (cont.)

Dereferencing operator

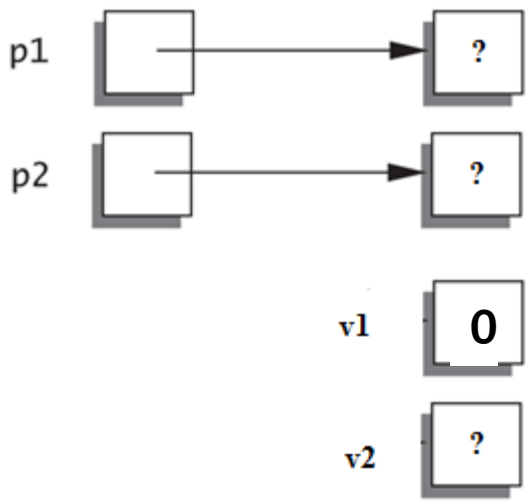
```
int *p1, *p2, v1, v2;
v1 = 0;
p1 = &v1;
*p1 = 42;
cout << v1 << endl;
cout << *p1 << endl;
```



1. Pointers – (cont.)

Dereferencing operator

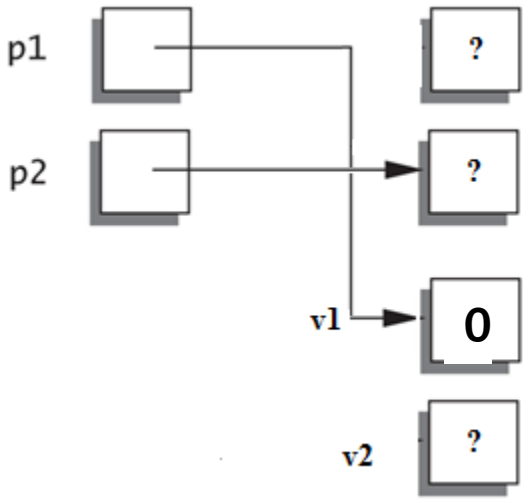
```
int *p1, *p2, v1, v2;
v1 = 0;
p1 = &v1;
*p1 = 42;
cout << v1 << endl;
cout << *p1 << endl;
```



1. Pointers – (cont.)

Dereferencing operator

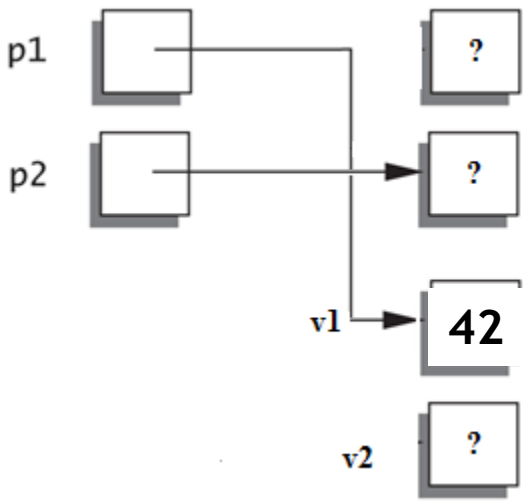
```
int *p1, *p2, v1, v2;
v1 = 0;
p1 = &v1;
*p1 = 42;
cout << v1 << endl;
cout << *p1 << endl;
```



1. Pointers – (cont.)

Dereferencing operator

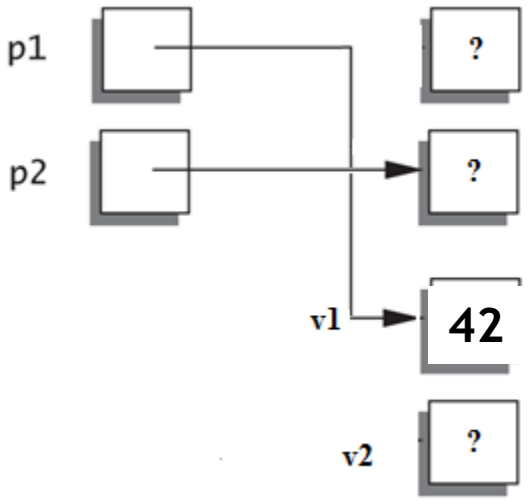
```
int *p1, *p2, v1, v2;
v1 = 0;
p1 = &v1;
*p1 = 42;
cout << v1 << endl;
cout << *p1 << endl;
```



1. Pointers – (cont.)

Dereferencing operator

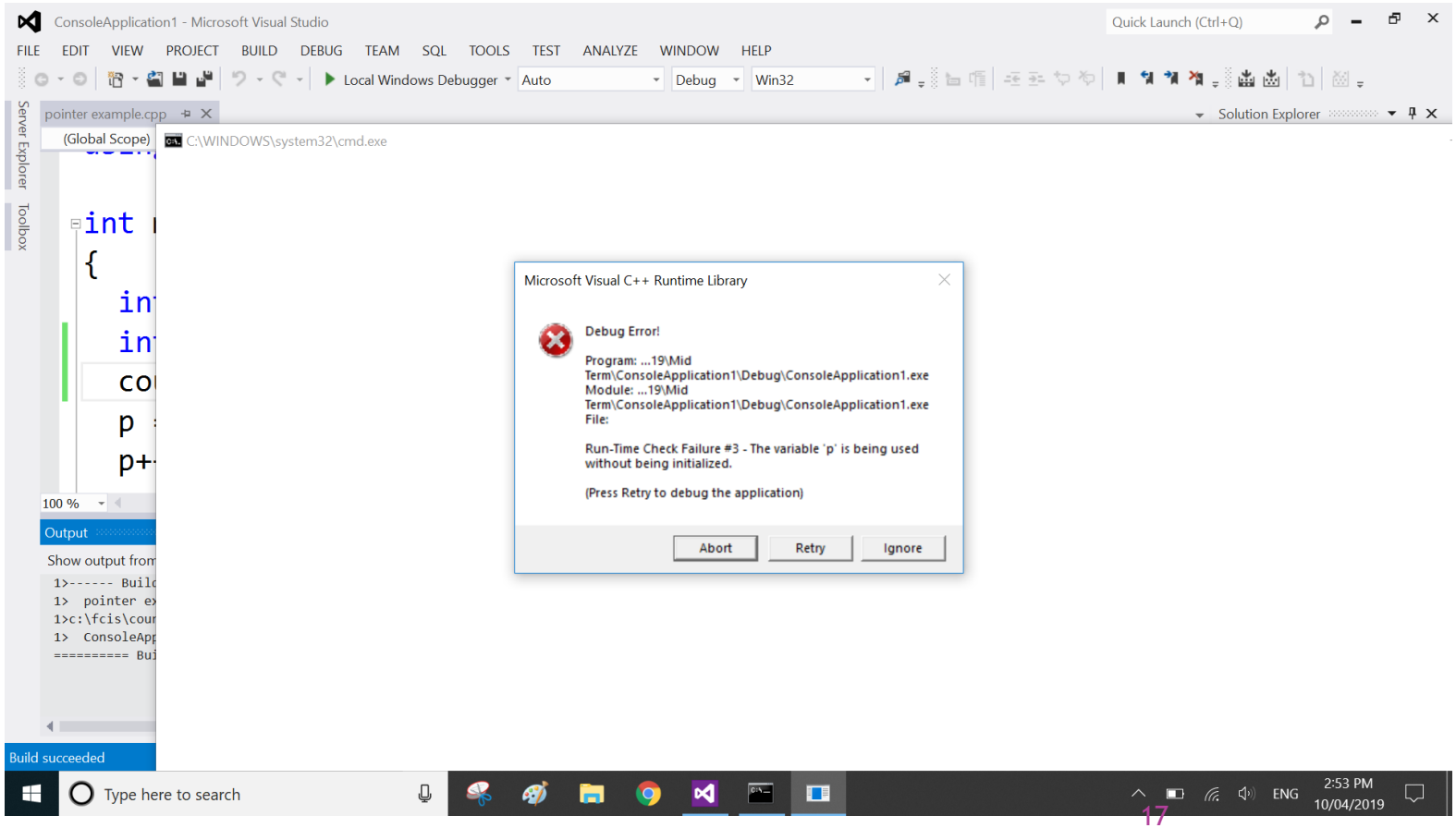
```
int *p1, *p2, v1, v2;
v1 = 0;
p1 = &v1;
*p1 = 42;
cout << v1 << endl;
cout << *p1 << endl;
```



- Produces output:
  42
  42
- p1 and v1 refer to same variable now.

1. Pointers – (cont.)

```
int* p;
cout << *p;
```



\\Summary:

```
int i=17;
int* ptr;   // define a ptr to an integer variable
ptr= &i;  // assign the address of i to pointer
cout << *ptr;  // prints contents of variable i
```

Practice

```
int v;     // defines variable v of type int
int w;     // defines variable w of type int
int *p;    // defines variable p of type pointer to int
p=&v;      // assigns address of v to pointer p
v=3;       // assigns value 3 to v
*p=7;      // assigns value 7 to v (pointed to by p)
p=&w;      // assigns address of w to pointer p
*p=12;     // assigns value 12 to w (pointed to by p)
```

1. Pointers – (cont.)

- Pointer is an address and address is a number, but pointer is NOT an integer! Cannot be used as numbers.

```
int add = ptr+1;
```

- Although it can be incremented (What does that mean?)

✓ `ptr++;`

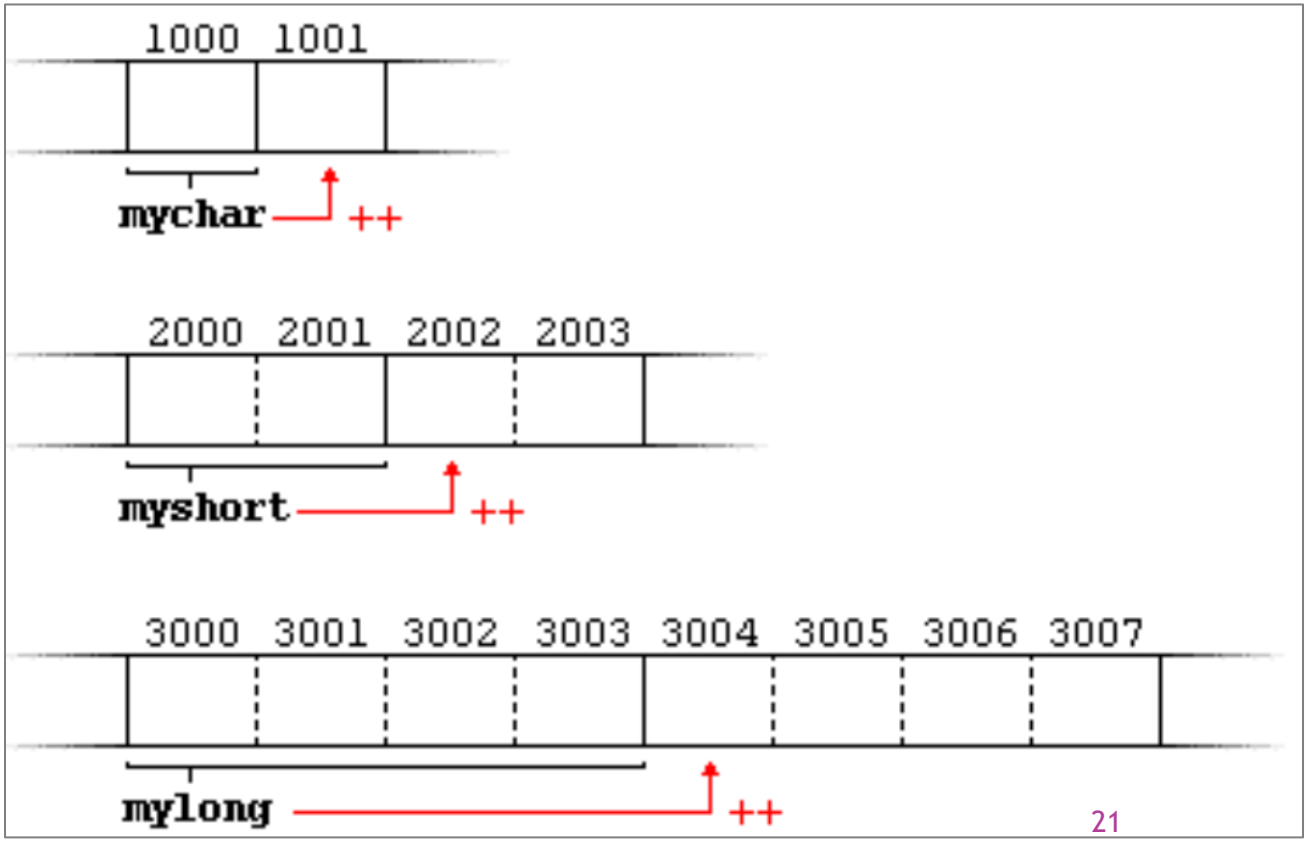
`ptr++;` is same as `++ptr;`, that is ptr starts pointing to the next location after increment.

`++* ptr` means to increment the value pointed by ptr , same as: `* ptr = * ptr + 1;`

Pointer Arithmetic

```
1 char *mychar;
2 short *myshort;
3 long *mylong;
```

```
1 ++mychar;
2 ++myshort;
3 ++mylong;
```

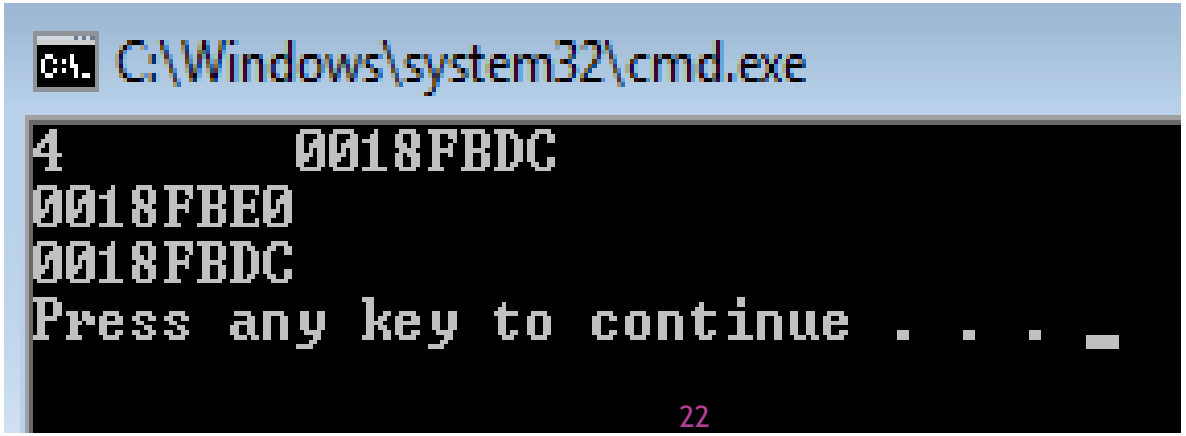


1. Pointers (cont.)

Pointer Arithmetic

```
int num = 4, *ptr = &num;
cout<<num<<"\t"<<ptr<<endl;

ptr = ptr + 1; cout<<ptr<<endl;
ptr = ptr - 1; cout<<ptr<<endl;
```

Here, the address will be altered by four bytes.

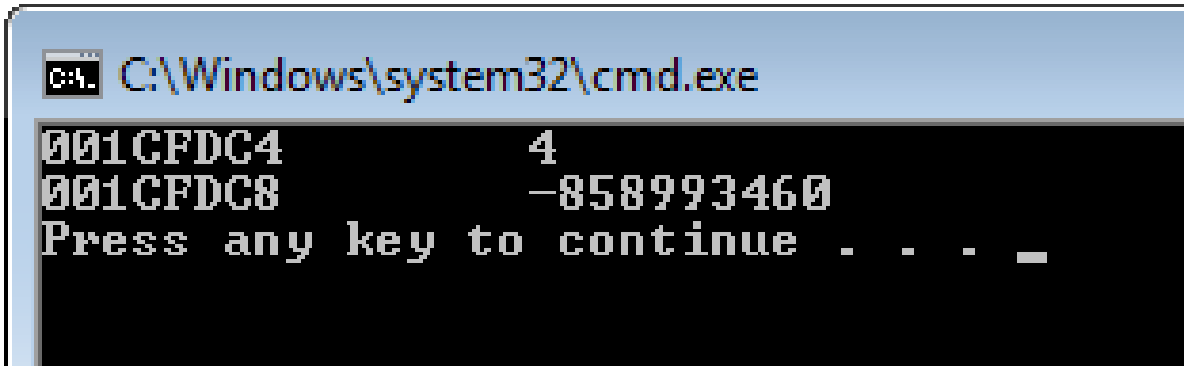


1. Pointers – (cont.)

Pointer Arithmetic

```
int num = 4, *ptr = &num;
cout<<ptr<<"\t"<<*ptr<<endl;

ptr = ptr + 1;
cout<<ptr<<"\t"<<*ptr<<endl;
```

Here, the address will be advanced by four bytes and you can VIEW the RANDOM value in that new address.



1. Pointers – (cont.)

- A pointer variable can be **assigned** to **any variable type**. *Pointer to integer, to float, to double, to struct, to array.*

- A pointer variable, like any other variable, **can be assigned** a value, or another variable of the same type.

- **Incrementing** Pointer Variable Depends Upon data type of the Pointer variable. The effect of applying the **increment** operator to a **pointer** of the type **pointer**-type* is to add sizeof(**pointer**-type) to the address that is contained in the **pointer** variable.

- This differs from compiler to compiler as memory required to store specific data types **vary compiler to compiler.**

1. Pointers – (cont.)

Pointers in Assignments

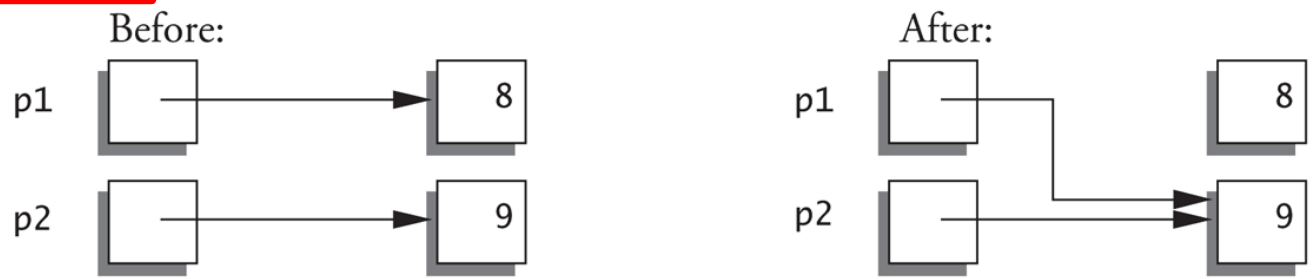
- Pointer variables can be "assigned":
  p2 = p1;
  - Assigns one pointer to another
  - Make p2 point to where p1 points

- Do not confuse with:
  *p2 = *p1;
  - Assigns "value pointed to" by p1 to "value pointed to" by p2

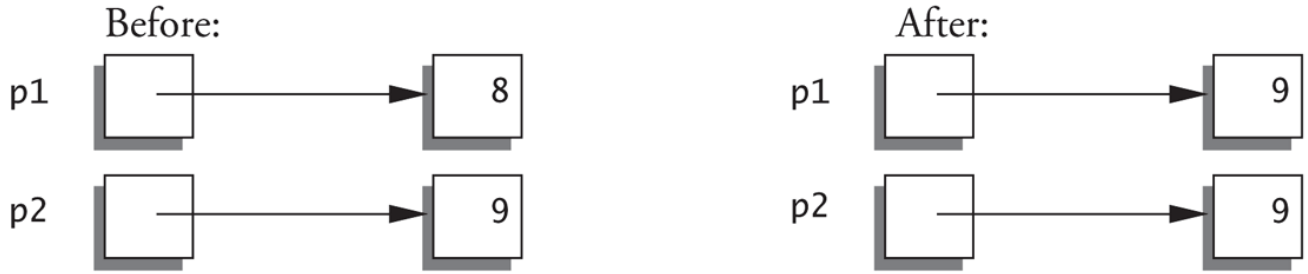
POINTER ASSIGNMENTS GRAPHIC: USES OF THE ASSIGNMENT OPERATOR WITH POINTER VARIABLES

Display 10.1 Uses of the Assignment Operator with Pointer Variables

```
p1 = p2;
```




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



1. Pointers – (cont.)

Dynamic Variables

- p1 = new int;
  - Creates new "nameless" variable, and assigns p1 to "point to" it.
  - Can access it with *p1
  - Used just like ordinary variable

1. Pointers – (cont.)

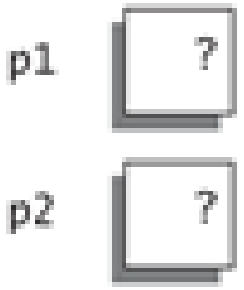
Dynamic Variables

- **Since pointers can refer to variables…**
  - **No "real" need to have a standard identifier.**
- **Can dynamically allocate variables**
  - For dynamic memory allocation, C++ offers operator **new.** Operator new returns the pointer to the newly allocated space.
    - **No identifiers to refer to them**
    - **Just a pointer!**

```
1  //Program to demonstrate pointers and dynamic variables.
2  #include <iostream>
3  using std::cout;
4  using std::endl;


5  int main( )
6  {
7      int *p1, *p2;


8      p1 = new int;
9      *p1 = 42;
10     p2 = p1;
11     cout << "*p1 == " << *p1 << endl;
12     cout << "*p2 == " << *p2 << endl;


13     *p2 = 53;
14     cout << "*p1 == " << *p1 << endl;
15     cout << "*p2 == " << *p2 << endl;


16     p1 = new int;
17     *p1 = 88;
18     cout << "*p1 == " << *p1 << endl;
19     cout << "*p2 == " << *p2 << endl;


20     cout << "Hope you got the point of this example!\n";
21     return 0;
22 }
```

(a)

```
int *p1, *p2;
```

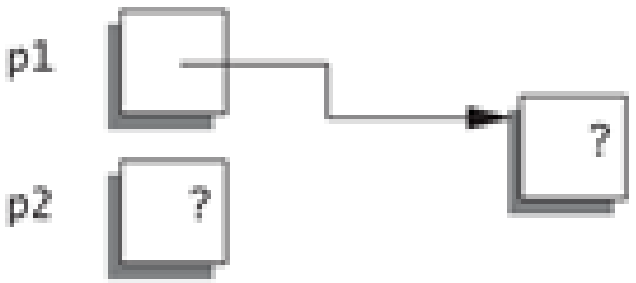


SAMPLE DIALOGUE

```
1  //Program to demonstrate pointers and dynamic variables.
2  #include <iostream>
3  using std::cout;
4  using std::endl;


5  int main( )
6  {
7      int *p1, *p2;


8      p1 = new int;
9      *p1 = 42;
10     p2 = p1;
11     cout << "*p1 == " << *p1 << endl;
12     cout << "*p2 == " << *p2 << endl;


13     *p2 = 53;
14     cout << "*p1 == " << *p1 << endl;
15     cout << "*p2 == " << *p2 << endl;


16     p1 = new int;
17     *p1 = 88;
18     cout << "*p1 == " << *p1 << endl;
19     cout << "*p2 == " << *p2 << endl;


20     cout << "Hope you got the point of this example!\n";
21     return 0;
22 }
```

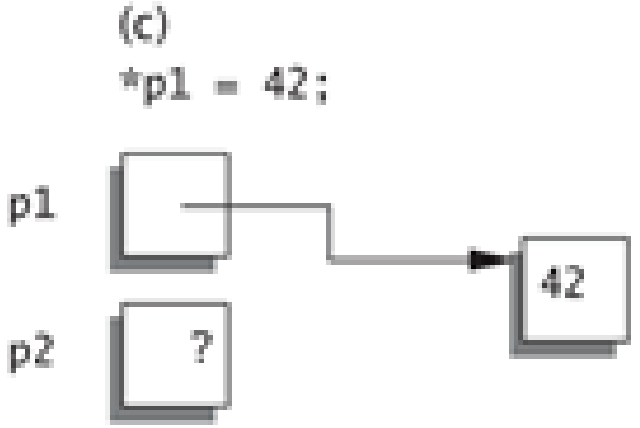
(b)

```
p1 = new int;
```



SAMPLE DIALOGUE

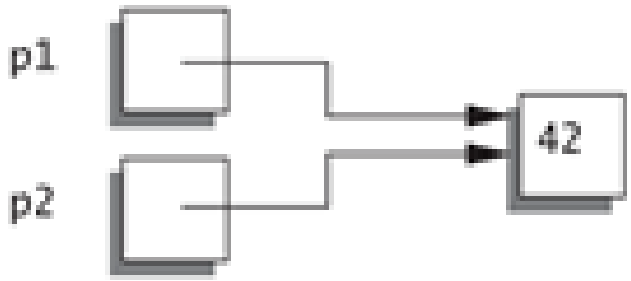
```
1  //Program to demonstrate pointers and dynamic variables.
2  #include <iostream>
3  using std::cout;
4  using std::endl;


5  int main( )
6  {
7      int *p1, *p2;


8      p1 = new int;
9      *p1 = 42;
10     p2 = p1;
11     cout << "*p1 == " << *p1 << endl;
12     cout << "*p2 == " << *p2 << endl;


13     *p2 = 53;
14     cout << "*p1 == " << *p1 << endl;
15     cout << "*p2 == " << *p2 << endl;


16     p1 = new int;
17     *p1 = 88;
18     cout << "*p1 == " << *p1 << endl;
19     cout << "*p2 == " << *p2 << endl;


20     cout << "Hope you got the point of this example!\n";
21     return 0;
22 }
```



SAMPLE DIALOGUE

```
1  //Program to demonstrate pointers and dynamic variables.
2  #include <iostream>
3  using std::cout;
4  using std::endl;


5  int main( )
6  {
7      int *p1, *p2;


8      p1 = new int;
9      *p1 = 42;
10     p2 = p1;
11     cout << "*p1 == " << *p1 << endl;
12     cout << "*p2 == " << *p2 << endl;


13     *p2 = 53;
14     cout << "*p1 == " << *p1 << endl;
15     cout << "*p2 == " << *p2 << endl;


16     p1 = new int;
17     *p1 = 88;
18     cout << "*p1 == " << *p1 << endl;
19     cout << "*p2 == " << *p2 << endl;


20     cout << "Hope you got the point of this example!\n";
21     return 0;
22 }
```

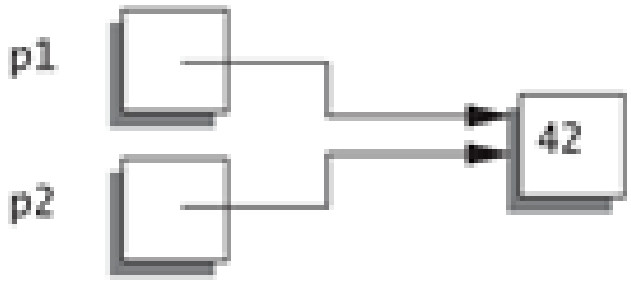
(d)
p2 = p1;



SAMPLE DIALOGUE

```
1  //Program to demonstrate pointers and dynamic variables.
2  #include <iostream>
3  using std::cout;
4  using std::endl;


5  int main( )
6  {
7      int *p1, *p2;


8      p1 = new int;
9      *p1 = 42;
10     p2 = p1;
11     cout << "*p1 == " << *p1 << endl;
12     cout << "*p2 == " << *p2 << endl;


13     *p2 = 53;
14     cout << "*p1 == " << *p1 << endl;
15     cout << "*p2 == " << *p2 << endl;


16     p1 = new int;
17     *p1 = 88;
18     cout << "*p1 == " << *p1 << endl;
19     cout << "*p2 == " << *p2 << endl;


20     cout << "Hope you got the point of this example!\n";
21     return 0;
22 }
```

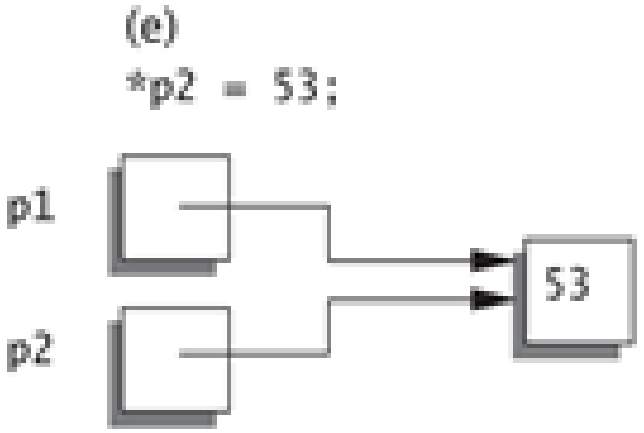
(d)
p2 = p1;



SAMPLE DIALOGUE

```
*p1 == 42
*p2 == 42
```

```
1  //Program to demonstrate pointers and dynamic variables.
2  #include <iostream>
3  using std::cout;
4  using std::endl;


5  int main( )
6  {
7      int *p1, *p2;


8      p1 = new int;
9      *p1 = 42;
10     p2 = p1;
11     cout << "*p1 == " << *p1 << endl;
12     cout << "*p2 == " << *p2 << endl;


13     *p2 = 53;
14     cout << "*p1 == " << *p1 << endl;
15     cout << "*p2 == " << *p2 << endl;


16     p1 = new int;
17     *p1 = 88;
18     cout << "*p1 == " << *p1 << endl;
19     cout << "*p2 == " << *p2 << endl;


20     cout << "Hope you got the point of this example!\n";
21     return 0;
22 }
```

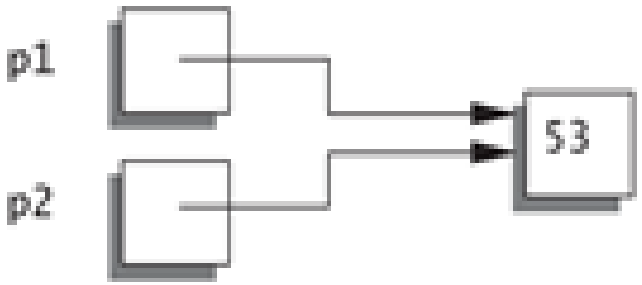


SAMPLE DIALOGUE

```
*p1 == 42
*p2 == 42
```

```
1  //Program to demonstrate pointers and dynamic variables.
2  #include <iostream>
3  using std::cout;
4  using std::endl;


5  int main( )
6  {
7      int *p1, *p2;


8      p1 = new int;
9      *p1 = 42;
10     p2 = p1;
11     cout << "*p1 == " << *p1 << endl;
12     cout << "*p2 == " << *p2 << endl;


13     *p2 = 53;
14     cout << "*p1 == " << *p1 << endl;
15     cout << "*p2 == " << *p2 << endl;


16     p1 = new int;
17     *p1 = 88;
18     cout << "*p1 == " << *p1 << endl;
19     cout << "*p2 == " << *p2 << endl;


20     cout << "Hope you got the point of this example!\n";
21     return 0;
22 }
```

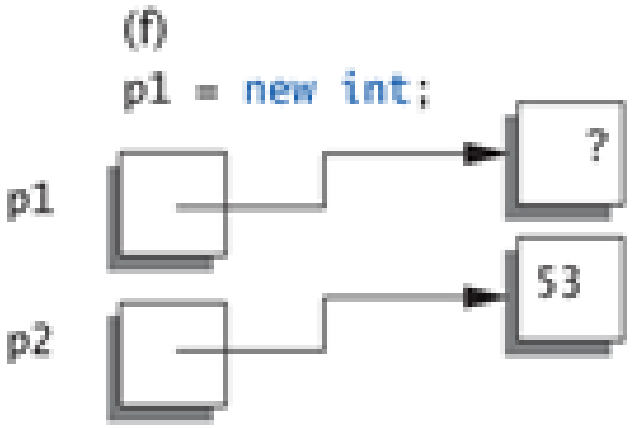
(e)
*p2 = 53;



SAMPLE DIALOGUE

```
*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
```

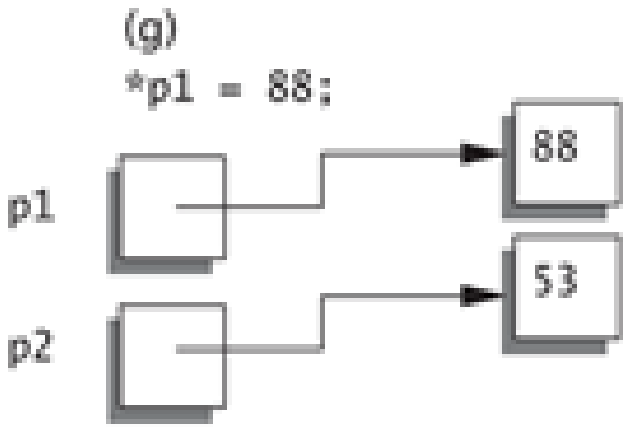
```
1  //Program to demonstrate pointers and dynamic variables.
2  #include <iostream>
3  using std::cout;
4  using std::endl;


5  int main( )
6  {
7      int *p1, *p2;


8      p1 = new int;
9      *p1 = 42;
10     p2 = p1;
11     cout << "*p1 == " << *p1 << endl;
12     cout << "*p2 == " << *p2 << endl;


13     *p2 = 53;
14     cout << "*p1 == " << *p1 << endl;
15     cout << "*p2 == " << *p2 << endl;


16     p1 = new int;
17     *p1 = 88;
18     cout << "*p1 == " << *p1 << endl;
19     cout << "*p2 == " << *p2 << endl;


20     cout << "Hope you got the point of this example!\n";
21     return 0;
22 }
```



SAMPLE DIALOGUE

```
*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
```

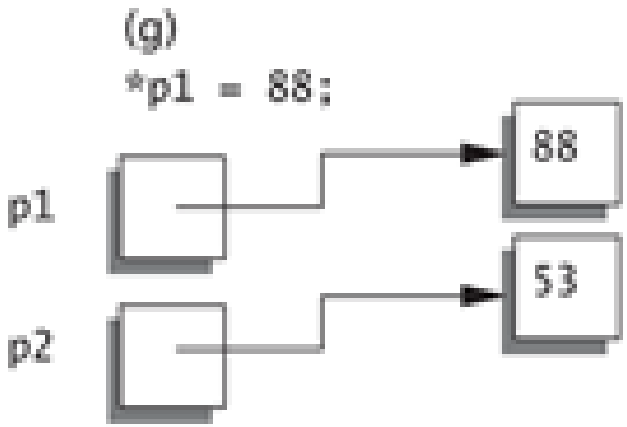
```
1  //Program to demonstrate pointers and dynamic variables.
2  #include <iostream>
3  using std::cout;
4  using std::endl;


5  int main( )
6  {
7      int *p1, *p2;


8      p1 = new int;
9      *p1 = 42;
10     p2 = p1;
11     cout << "*p1 == " << *p1 << endl;
12     cout << "*p2 == " << *p2 << endl;


13     *p2 = 53;
14     cout << "*p1 == " << *p1 << endl;
15     cout << "*p2 == " << *p2 << endl;


16     p1 = new int;
17     *p1 = 88;
18     cout << "*p1 == " << *p1 << endl;
19     cout << "*p2 == " << *p2 << endl;


20     cout << "Hope you got the point of this example!\n";
21     return 0;
22 }
```



SAMPLE DIALOGUE

```
*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
```

```
1  //Program to demonstrate pointers and dynamic variables.
2  #include <iostream>
3  using std::cout;
4  using std::endl;


5  int main( )
6  {
7      int *p1, *p2;


8      p1 = new int;
9      *p1 = 42;
10     p2 = p1;
11     cout << "*p1 == " << *p1 << endl;
12     cout << "*p2 == " << *p2 << endl;


13     *p2 = 53;
14     cout << "*p1 == " << *p1 << endl;
15     cout << "*p2 == " << *p2 << endl;


16     p1 = new int;
17     *p1 = 88;
18     cout << "*p1 == " << *p1 << endl;
19     cout << "*p2 == " << *p2 << endl;


20     cout << "Hope you got the point of this example!\n";
21     return 0;
22 }
```



SAMPLE DIALOGUE

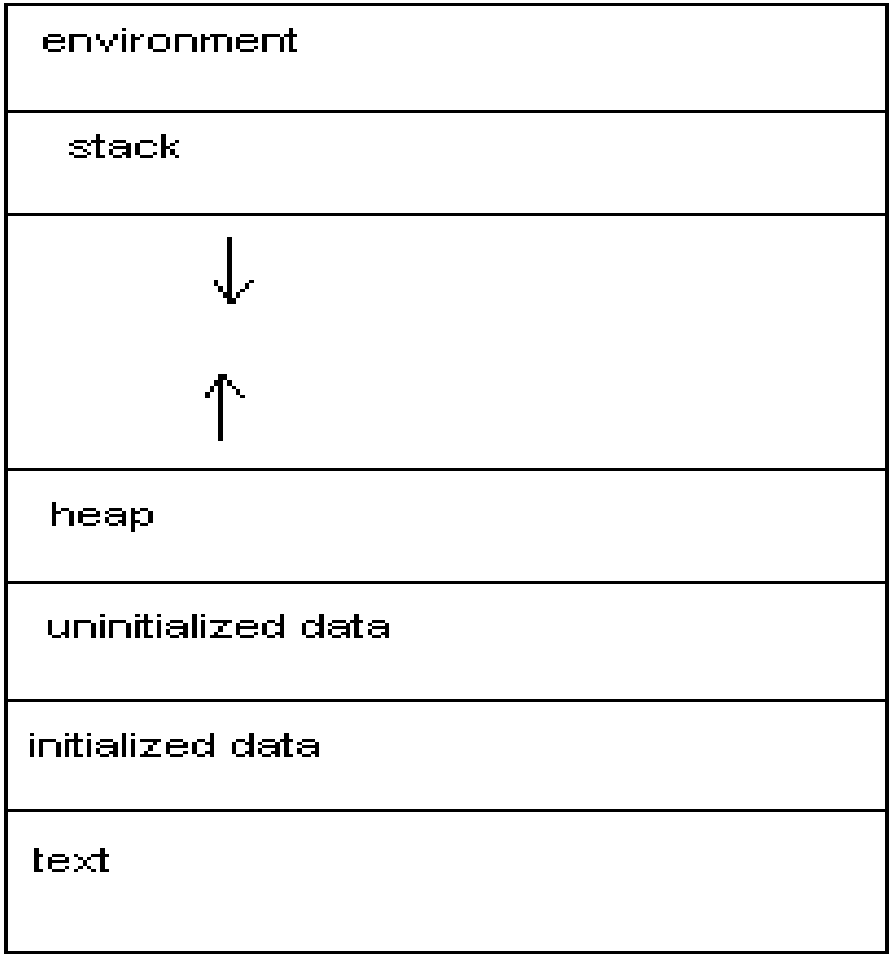
```
*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
*p1 == 88
*p2 == 53
Hope you got the point of this example!
```

1. Pointers – (cont.)

Initialization

```
int *iptr = new int (33); //init *iptr to 33

double *dptr;
dptr = new double (11.2); // ptr points to
nameless double var of value 11.2

int num(5);  // exactly as int num = 5;
int* ptr = new int(num);
```

2. Memory Management



Virtual memory organization

2. Memory Management – (cont.)

- **Text** segment is one of the sections of a program in an object file or in memory, which contains executable instructions.
- **Initialized Data** segment contains the global variables and static variables that are initialized by the programmer.
- **Uninitialized Data** contains all global variables and static variables that do not have explicit initialization in source code.

2. Memory Management – (cont.)

Stack

- All the variables, declared in functions (including main()) will be placed in **stack.**
- We call it stack memory allocation because the allocation happens in function call stack.
- The size of memory to be allocated is known to compiler and whenever a function is called, its variables get memory allocated on the stack.
- Whenever the function call is over, the memory for the variables is deallocated.
- This all happens using some predefined routines in compiler. Programmer does not have to worry about memory management of stack variables.

2. Memory Management

The Heap (free-store)

- Reserved for dynamically-allocated variables.
- The memory is allocated during execution of instructions written by programmers.
- All new dynamic variables consume memory in free-store.
- If too many → could use all free-store memory.
- Future "new" operations will fail if free-store is "full".

2. Memory Management – (cont.)

Insufficient Memory Test

- Older compilers required to test the return of the new operator.

```
int* ptr = new int;
if(ptr==NULL)
{
    cout<<"Failed to allocate memory.\n";
    exit(1);
}
else
    cout<<"successful new allocation.\n";
```

2. Memory Management – (cont.)

Insufficient Memory Test

- Newer compilers:
  - If new operation fails:
    - Program terminates automatically
    - Produces error message
- Still good practice to use NULL check

2. Memory Management – (cont.)

The delete Operator

- De-allocate dynamic memory pointed to by pointer variable
  - When no longer needed
  - Returns memory to free-store

```
// variables
int* ptr = new int(5);
// processing
//...
delete ptr;
```

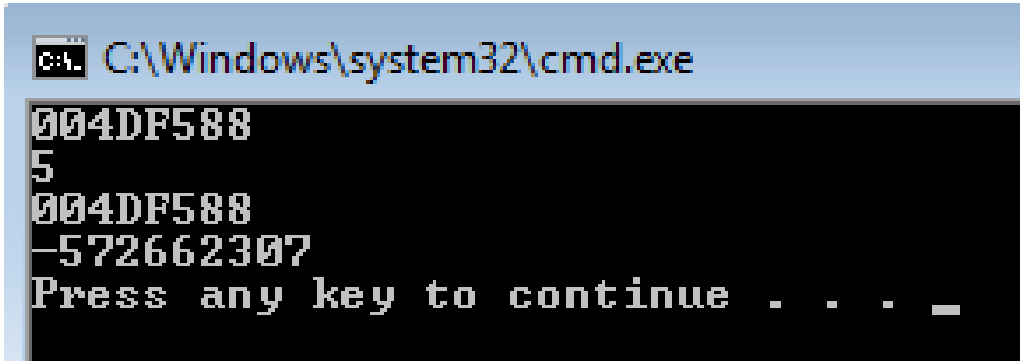
2. Memory Management – (cont.)

Dangling Pointers

`delete ptr` destroys dynamic memory but ptr still points there! Called dangling pointer.

- If ptr is then dereferenced by *ptr it may cause unpredictable results!

```
int* ptr = new int(5);
// processing
cout<<ptr<<endl;
cout<<*ptr<<endl;

delete ptr;

cout<<ptr<<endl;
cout<<*ptr<<endl;
```



2. Memory Management – (cont.)

Dangling Pointers

- **Avoid dangling pointers by assigning pointer to NULL after delete: pointer not pointing to anything.**

```
int* ptr = new int(5);
// processing
cout<<ptr<<endl;
cout<<*ptr<<endl;

delete ptr;
ptr = NULL;

cout<<ptr<<endl;
cout<<*ptr<<endl;
```

Same as:

```
ptr= nullptr;
```

You cannot dereference a null pointer. Runtime Error!

2. Memory Management – (cont.)

Dynamic and Automatic Variables

- **Dynamic variables**
  - Created with new operator
  - Created and destroyed while program runs

- **Automatic (local) variables**
  - Declared within function definition
  - Not dynamic
    - **Created when function is called**
    - **Destroyed when function call completes**
    - **Properties controlled for you**

POINTERS USAGE

- Pointers are used to:
  - Access array elements
  - Passing arguments to functions when the function needs to modify the original argument
    - by value : void f(int x);
    - by reference : void f(int& x);
    - by pointer : void f(int* x);
  - Passing arrays and strings to functions
  - Obtaining memory from the system
  - Creating data structures whose size can grow shrink dynamically such as linked lists


Practice
Practice
Practice

Practice (Try it first)

```
firstvalue is 10
secondvalue is 20
```

```
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue = 5, secondvalue = 15;                    5 , 15
  int * p1, * p2;

  p1 = &firstvalue;  // p1 = address of firstvalue
  p2 = &secondvalue; // p2 = address of secondvalue
  *p1 = 10;          // value pointed to by p1 = 10        10 , 15
  *p2 = *p1;         // val pointed to by p2 = val pointed 10 , 10
                        to by
  p1 = p2;           // p1=p2 (value of pointer is copied)
                        both points to “secondvalue”
  *p1 = 20;          // value pointed to by p1 = 20        10 , 20

  cout << "firstvalue is " << firstvalue << '\n';
  cout << "secondvalue is " << secondvalue << '\n';
  return 0;
```

Practice

- ptr++;

// use it then pointer moves to the next int position

- ++ptr;

// pointer moves to the next int position then use it

- ++*ptr;

// the value is incremented by 1 then used

- *++ptr;

// pointer moves to the next int position then uses value

- *ptr++;

// uses value then pointer moves to the next int position

Practice (self study)

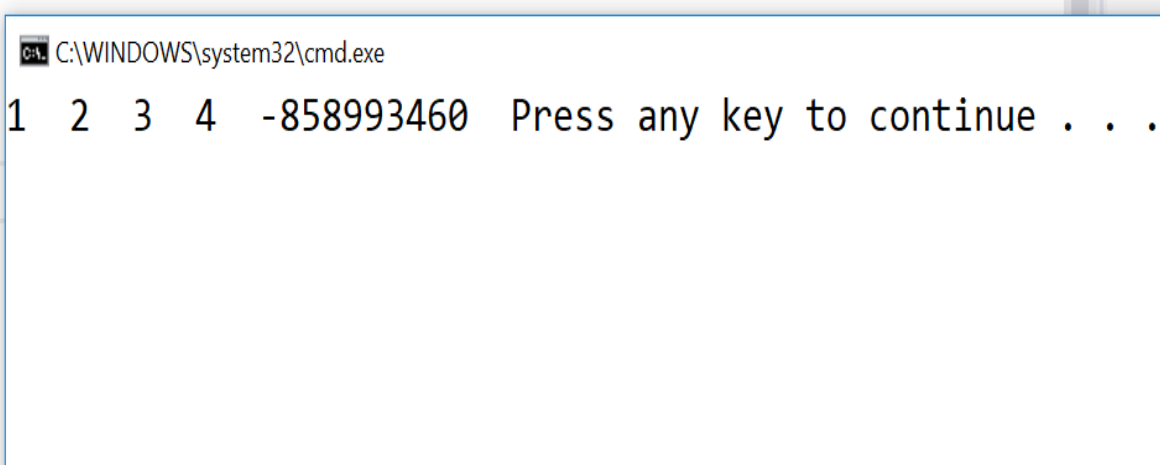
```
int *ptr = new int (33);
cout <<"Initial" << endl;
cout << ptr << '\t' << *ptr << endl;

cout << "++ptr;" << endl;
cout << ++ptr << '\t' << *ptr << endl;

cout << "ptr++;" << endl;
cout << ptr++ << '\t' << *ptr << endl;
cout << "ptr" << '\t' << ptr << endl;

cout << "++*ptr;" << endl;
cout << ++*ptr << endl;
cout << ptr << '\t' << *ptr << endl;

cout << "*++ptr;" << endl;
cout << *++ptr << endl;
cout << ptr << '\t' << *ptr << endl;

cout << "*ptr++;" << endl;
cout << *ptr++ << endl;
cout << ptr << '\t' << *ptr << endl;
```

```
Initial
0100D150        33
++ptr;
0100D154        -33686019
ptr++;
0100D154        -1607221541
ptr     0100D158
++*ptr;
-1607221540
0100D158        -1607221540
*++ptr;
-2147478819
0100D15C        -2147478819
*ptr++;
-2147478819
0100D160        -572662307
Press any key to continue . .
```

garbish

PRACTICE

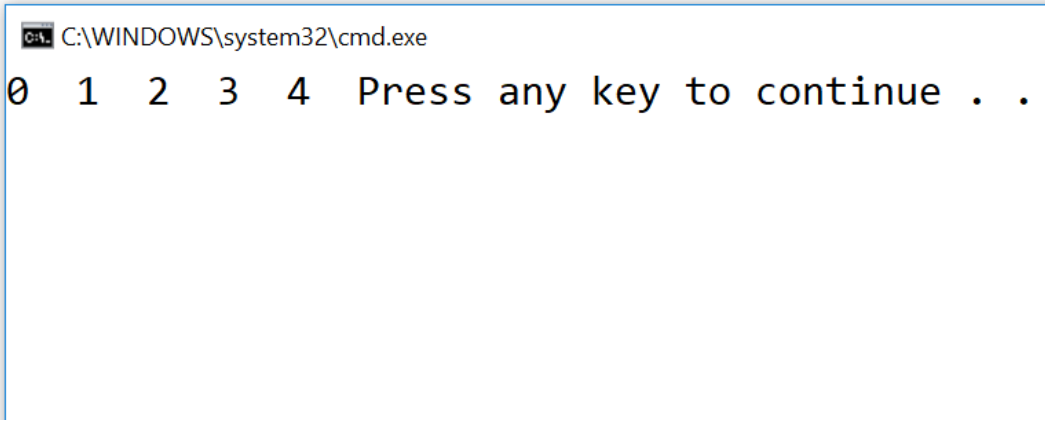
```
#include <iostream>
using namespace std;

void main ()
{
    int arr[5] = {0,1,2,3,4};
    int *ptr = arr;
    for (int i = 0; i < 5; i++)
        cout << *++ptr <<"  ";

}
```



*++ptr; // pointer moves to the next int position then uses value

PRACTICE

```
#include <iostream>
using namespace std;

void main ()
{
    int arr[5] = {0,1,2,3,4};
    int *ptr = arr;
    for (int i = 0; i < 5; i++)
        cout << *ptr++ <<"  ";

}
```



*ptr++; // uses value then pointer moves to the next int position

3. Examples – (cont.)

Trace Exercise

- **What is the output of the following code segment:**

```
int i=3,*j;
j=&i;
cout<<i**j*i+*j<<endl;
```

30

Pointers And Dynamic Arrays

3. Pointers and Arrays

- Our previous Array variables
  - Really pointer variables!
- Recall: arrays stored in memory addresses, sequentially
  - Array variable "refers to" first indexed variable
  - So array variable is a kind of pointer variable!
- Example:

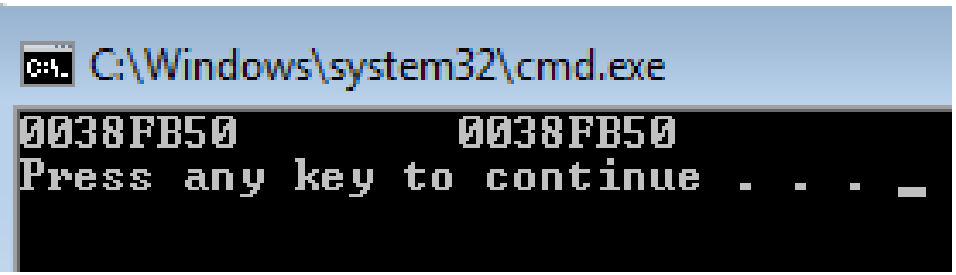
```
int arr[10];
int *p;
```

  - `arr` and `p` are both pointer variables!

3. Pointers and Arrays – (cont.)

- So, can they be assigned to each other?

```
int num = 4;
int arr[5] = {0};
int *ptr;

ptr = arr;
cout<<arr<<"\t"<<ptr<<endl;
```



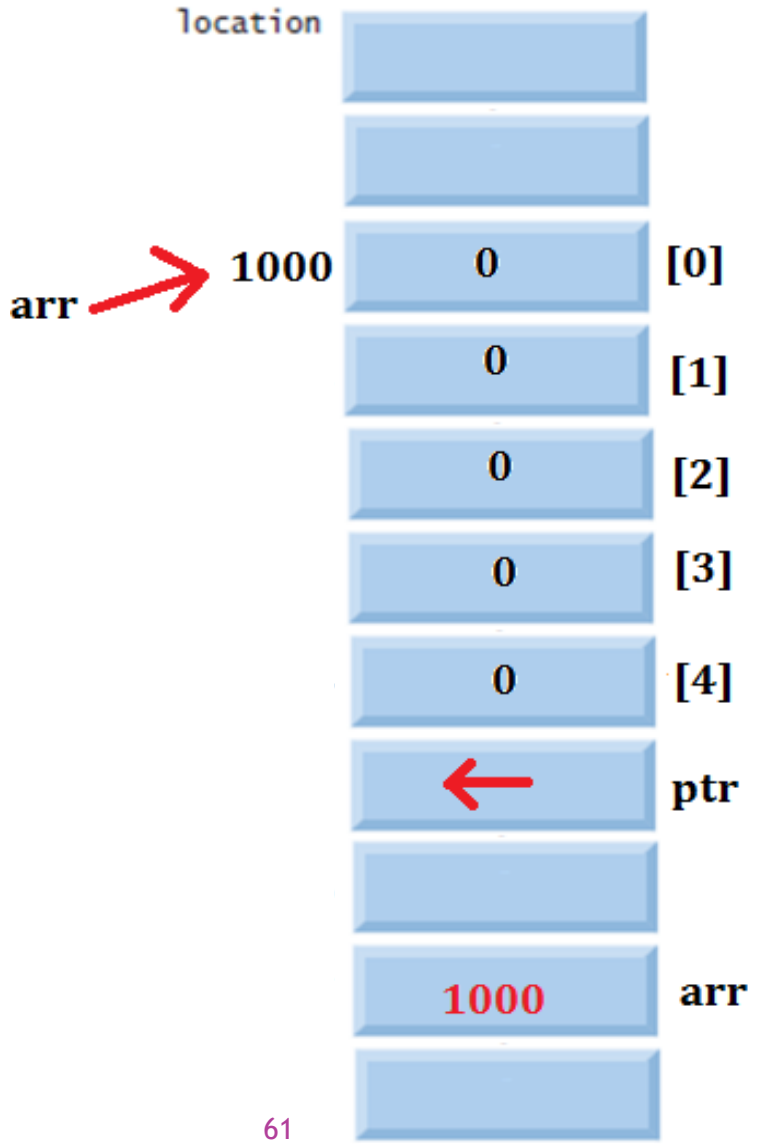

```
arr = ptr;
cout<<arr<<"\t"<<ptr<<endl;
```

ILLEGAL !
Note that an array name is exactly as a const int * type.

Main difference is that pointers can be assigned new addresses, while arrays cannot.
arr can never be assigned anything, will always represent same block of 5 int elements.

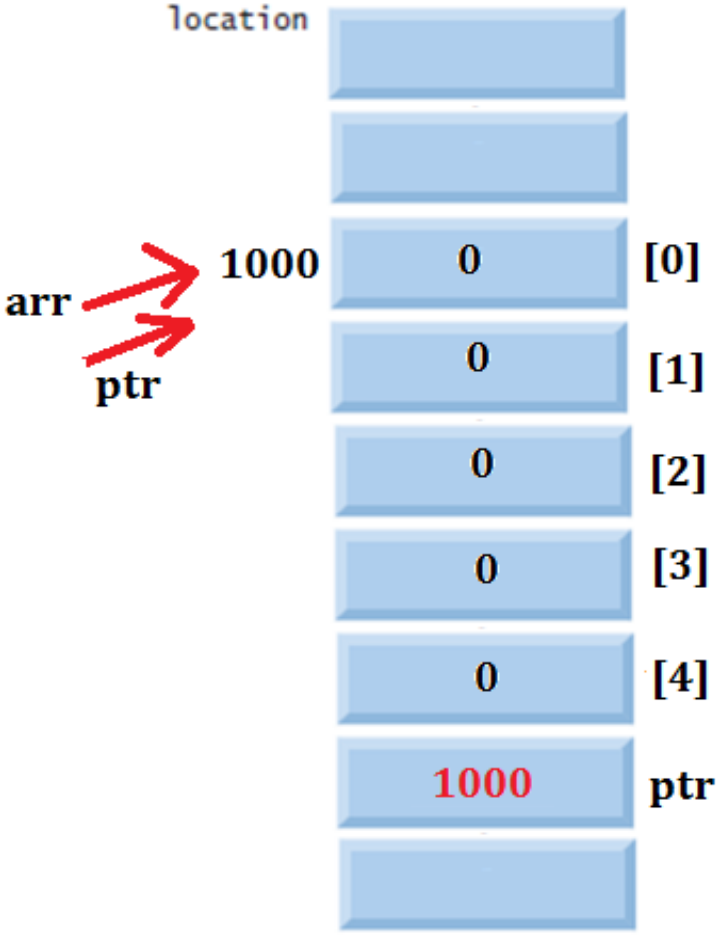
3. Pointers and Arrays – (cont.)

```
int arr[5] = {0};
int *ptr;
```



3. Pointers and Arrays – (cont.)

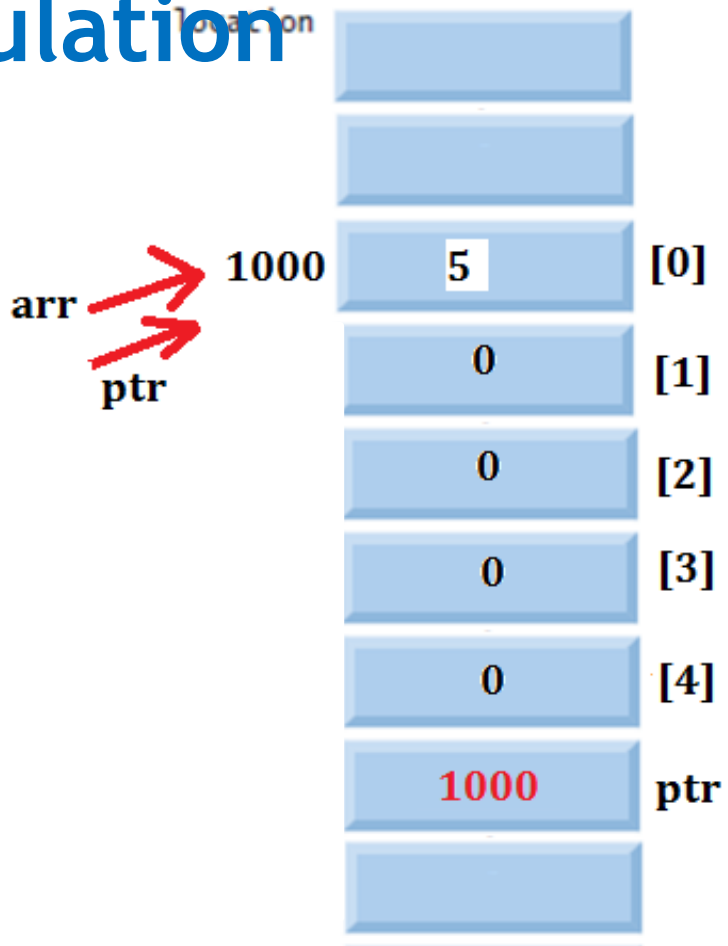
```
int arr[5] = {0};
int *ptr;

ptr = arr;
```

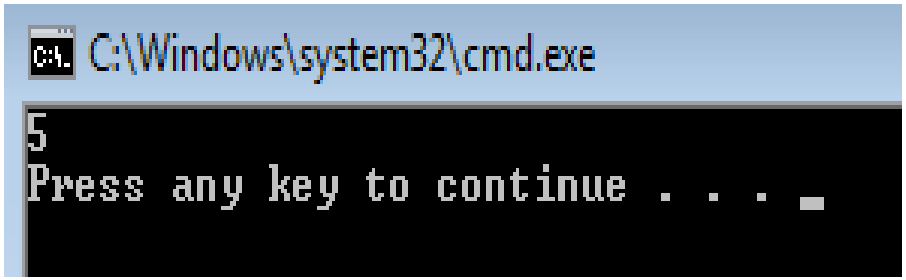


3. Pointers and Arrays – (cont.)

Alternative Arrays Manipulation

```
int arr[5] = {0};
int *ptr;

ptr = arr;

*ptr = 5;
cout<<arr[0]<<endl;
```



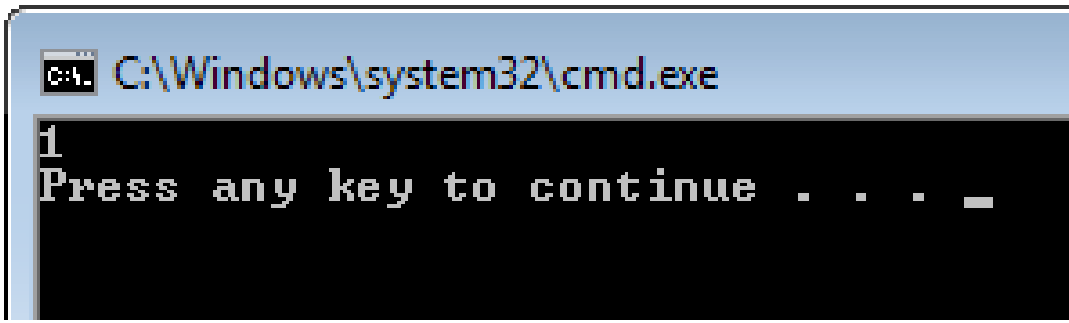


3. Pointers and Arrays – (cont.)

Alternative Arrays Manipulation

- **Using address arithmetic**

```
int arr[5] = {0, 1, 2, 3, 4};
int *ptr;

ptr = arr;
ptr++;
cout<<*ptr<<endl;
```



This is equivalent to

```
ptr = &arr[1];
```

3. Pointers and Arrays – (cont.)

Different ways to access array elements using a pointer

```
#include <iostream>
using namespace std;

void main ()
{
  int numbers[5];
  int * p;
  p = numbers;  *p = 10;
  p++;  *p = 20;
  p = &numbers[2];  *p = 30;
  p = numbers + 3;  *p = 40;
  p = numbers;  *(p+4) = 50;
  for (int n=0; n<5; n++)
    cout << numbers[n] << ", ";
}
```

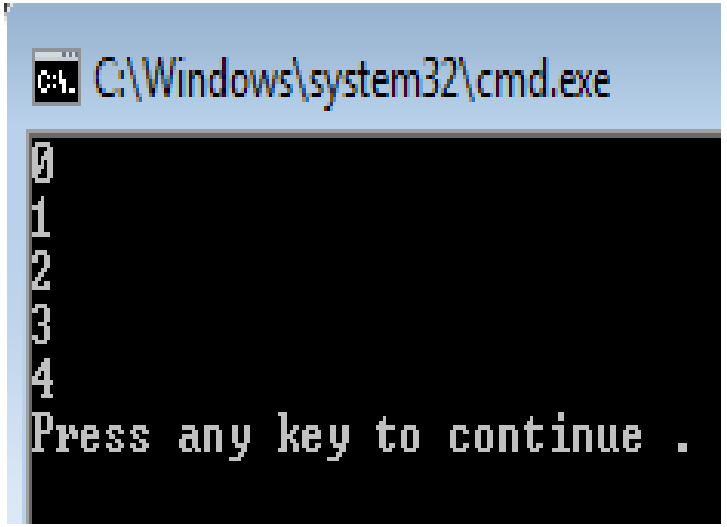
```
10, 20, 30, 40, 50,
```

3. Pointers and Arrays – (cont.)

Alternative Arrays Manipulation

- To display all elements

```
int arr[5] = {0, 1, 2, 3, 4};
int *ptr;

ptr = arr;
for(int i=0; i<5; i++)
    cout<<    ?    <<endl;
```

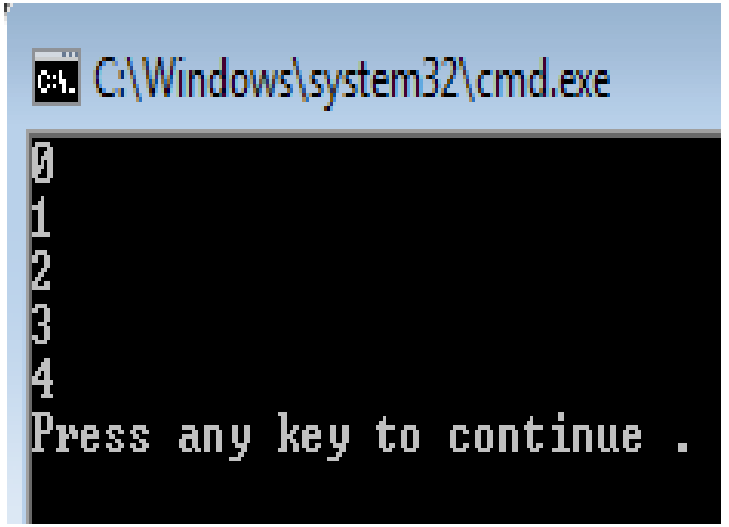


3. Pointers and Arrays – (cont.)

Alternative Arrays Manipulation

- To display all elements

```
int arr[5] = {0, 1, 2, 3, 4};
int *ptr;

ptr = arr;
for(int i=0; i<5; i++)
    cout<<*(ptr+i)<<endl;
```



3. Pointers and Arrays – (cont.)

Alternative Arrays Manipulation

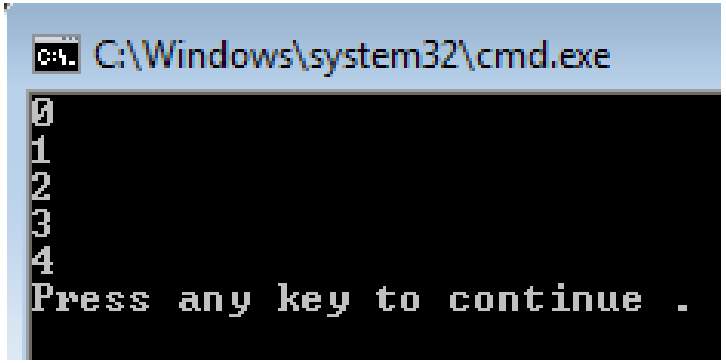
- To display all elements

```
int arr[5] = {0, 1, 2, 3, 4};
int *ptr;

ptr = arr;
for(int i=0; i<5; i++)
     cout<<*(ptr+i)<<endl;
```

This is equivalent to

```
 cout<<*(arr+i)<<endl;
```

Ptr didn't move
it still points to arr[0]

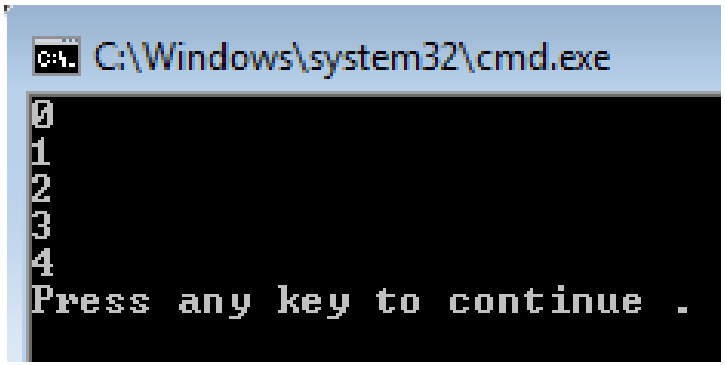


3. Pointers and Arrays – (cont.)

Alternative Arrays Manipulation

- To display all elements

```
int arr[5] = {0, 1, 2, 3, 4};
int *ptr;

ptr = arr;
for(int i=0; i<5; i++)
    cout<<   ?   <<endl;
```

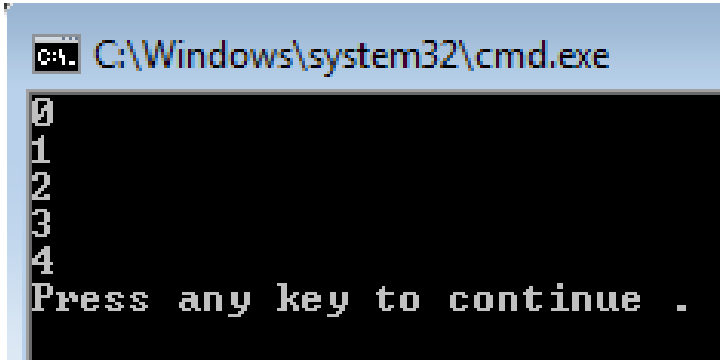


3. Pointers and Arrays – (cont.)

Alternative Arrays Manipulation

- To display all elements

```
int arr[5] = {0, 1, 2, 3, 4};
int *ptr;

ptr = arr;
for(int i=0; i<5; i++)
    cout<<ptr[i]<<endl;
```



*ptr++ can be used too (moves ptr itself)

3. Pointers and Arrays – (cont.)

- Brackets ([]) were explained as specifying the index of an element of the array.
- Well, these brackets are a **dereferencing** operator known as offset operator.
- They dereference the variable they follow just as * does, but they also add the number between brackets to the address being dereferenced.

```
1 a[5] = 0;       // a [offset of 5] = 0
2 *(a+5) = 0;     // pointed to by (a+5) = 0
```

4. Dynamic Arrays

Standard array

- Must specify size first - Fixed size
- May not know actual used size until program runs!
- Must "estimate" maximum size needed
  - "Wastes" memory

Dynamic array

- Size not specified at program design time
- Determined while program runs
- Can grow and shrink as needed

4. Dynamic Arrays – (cont.)

Allocating Dynamic Array Variable

- Use `new` operator
  - Dynamically allocates memory with pointer variable at run time.
- Int *p = new int[10]
  - Creates dynamically allocated array variable *p*, with ten elements, base type int.

4. Dynamic Arrays – (cont.)

Allocating Dynamic Array Variable

- The advantage here is that we can assign a variable as the array size. Something we couldn't do in automatic array variables.

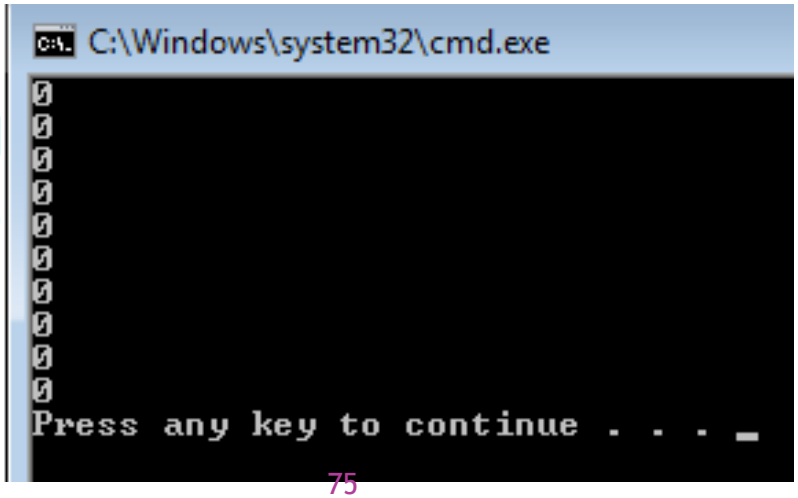
```
int size  = 4; // or cin>>size from user
int arr[size];
int *dArr = new int[size];
```

4. Dynamic Arrays – (cont.)

Initializing Dynamic Array Variable

- Recall array variable zero initialization

```
int arr[5] = {}; // same as = {0}
for(int i=0; i<5; i++)
    cout<<arr[i]<<endl;
```



4. Dynamic Arrays – (cont.)

check

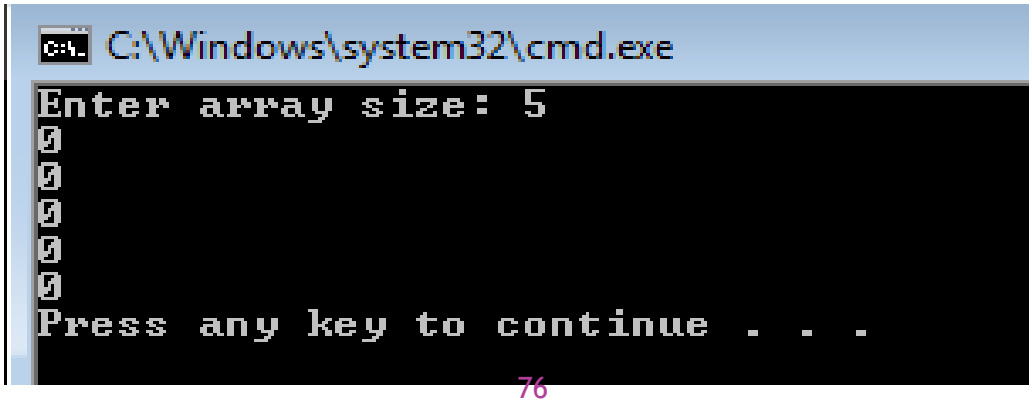
Initializing Dynamic Array Variable

- Dynamic arrays are zero initialized only

```
int size;
cout<<"Enter array size: "; cin>>size;

int *dArr = new int[size]();
for(int i=0; i<size; i++)
    cout<<dArr[i]<<endl;
```

Remember you can also write

```
cout<<*(dArr+i)<<endl;
```



4. Dynamic Arrays – (cont.)

Processing Dynamic Array Variable

- Treated like any standard array.

```
dArr = new double[10];
```

`dArr` contains address of `dArr[0]`

`dArr+1` evaluates to address of `dArr[1]`

`dArr+2` evaluates to address of `dArr[2]`

… and so on.

4. Dynamic Arrays – (cont.)

De-allocating Dynamic Array Variable

```
delete [] dArr;
```

- Returns memory to OS.
- Brackets indicates array;
- Remember `dArr` pointer still exists

```
dArr = NULL;
```

```
#include <iostream>
using namespace std;
void main()
{
int* a;           // Pointer to int
int n;            // Size needed for array
cin >> n;         // Read in the size
a = new int[n];   // Allocate n ints, save ptr in a

for (int i=0; i<n; i++)
{
     a[i] = i;
     cout << a[i] << endl;
}

delete [] a;  // Free memory pointed to by a
a = NULL;  // Prevent using invalid memory reference
}
```

5. Examples – (cont.)

Example:

- Creating an array of student structures with variable size. Compute the average of grades for each student and the average of all students.

Sample Run:

```
C:\WINDOWS\system32\cmd.exe
Name: Ahmed
Subject#1: 23
Subject#2: 43
Subject#3: 43
ID: 20
Name: Sarah
Subject#1: 55
Subject#2: 44
Subject#3: 55
ID: 30
Name: Ola
Subject#1: 65
Subject#2: 61
Subject#3: 62
ID: 40
Name: John
Subject#1: 87
Subject#2: 76
Subject#3: 77
Average of All Students57.5833
ID        Name     Average Score
10        Ahmed    36.3333
20        Sarah    51.3333
30        Ola      62.6667
40        John     80
Press any key to continue . . .
```

```
#include <iostream>
using namespace std;
#define GRADES  3
struct Std
{
   int ID;
   char name[20];
   double grades[Grades];
   double avg;
};
void fillArray(Std* studs,int size);
void ComputeAvgofAll(Std* studs,int size);
double ComputeAvgofOneStud(double grades[],int size);
void display(Std* studs,int size);
void main()
{
int numStd;
cout<< "How many students: ";
cin>> numStd;
```

```
// processing
// allocate
Std *students = new Std[numStd];

//fill data
fillArray (students, numStd);

//compute average
for(int i=0; i<numStd; i++)
  students[i].avg=computeAvgofOneStud (students[i].grades,
GRADES);

//compute average of all students
computeAvgofAll (students, numStd);

// output
display (students, numStd);

// dellocate
delete [] students;
students = nullptr;
}
```

Check GRADES

```
void ComputeAvgofAll(Std* studs,int size)
{
double Average=0.0;
for(int i=0;i<size;i++)
   Average += studs[i].avg;
Average = Average/size;
cout << "Average of All Students"<<Average<<endl;
}

double ComputeAvgofOneStud(double grades[],int size)
{
double Average=0.0;
for(int j=0;j<size;j++)
    Average += grades[j];
Average = Average/size;
return Average;
}
```

```
void display(Std* studs, int size)
{
cout<<"ID\tName\tAverage Score\n";
for(int i=0; i<size; i++)
   cout<< studs[i].ID <<"\t"<<
studs[i].name<<"\t"<<studs[i].avg <<endl;
}
void fillArray(Std* studs, int size)
{
for(int i=0;i<size;i++)
{
  cout<<"ID: "; cin>> studs[i].ID;
  cout<<"Name: "; cin>> studs[i].name;
  for(int j=0; j<GRADES; j++)
  {
    cout<<"Subject#"<<j+1<<": ";
    cin>> studs[i].grades[j];
  }
}}
```

Can we create a pointer to struct ??

Pointers to structures

```
1 struct movies_t {
2   string title;
3   int year;
4 };
5
6 movies_t amovie;
7 movies_t * pmovie;
```

```
pmovie = &amovie;
```

Structures can be pointed to by its own type of pointers.

The value of the pointer pmovie would be assigned the address of object amovie.

The arrow operator (->) is a dereference operator that is used exclusively with pointers to objects that have members.
It accesses the member of an object directly from its address.

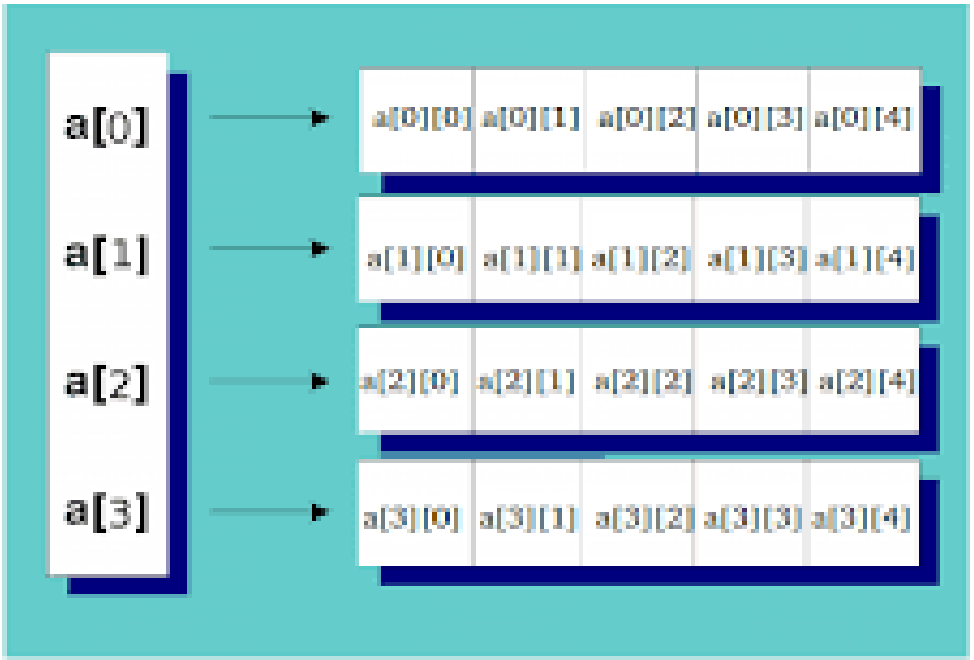
pmovie -> title
is equivalent to: (*pmovie).title

CAN WE CREATE A 2D DYNAMIC ARRAY ??

- A 2D array is basically a 1D array of pointers, where every pointer is pointing to a 1D array, which holds the actual data. Number of rows and columns can be obtained from user.

- Google it !

(Not in Exam)



HOME ASSIGNMENT 6

Update the last assignment to match the following requirement:

- Instead of using a fixed 1D array of structures, use a dynamic array instead that will be created according to the size the user specifies. And delete the array at the end of the program.

```
:Enter the number of products:5

Please enter the 5 products:

Enter values of product 1

1231

3

20

50

Enter values of product 2

2313

10

10

40

Enter values of product 3

3331

4

12

40

Enter values of product 4

3345

14

15

100

Enter values of product 5

1212

20

30

55

-----------------------------------------------
----------------

Please enter a number:
Press 1 to get products that has less quantity
than a certain value
Press 2 to Get Product with the highest sales
Press 3 to Apply 50% discount for products
that has quantity less than 5
Press 4 to count the number of products with
prices less than a certain amount
Press 5 to Display all the products
1

Please enter the quantity: 15

Product 1

Product 2

Product 3

Product 4

Do you want to Apply Another function, Press
'Y' or 'y' for yes, any other key to stop :y
```

```
Please enter a number:
Press 1 to get products that has less quantity than a certain value
Press 2 to Get Product with the highest sales
Press 3 to Apply 50% discount for products that has quantity less than 5
Press 4 to count the number of products with prices less than a certain amount
Press 5 to Display all the products


2
Product 4


Do you want to Apply Another function, Press 'Y' or 'y' for yes, any other key to stop :y

-----------------------------------------------------------
Please enter a number:
Press 1 to get products that has less quantity than a certain value
Press 2 to Get Product with the highest sales
Press 3 to Apply 50% discount for products that has quantity less than 5
Press 4 to count the number of products with prices less than a certain amount
Press 5 to Display all the products
3
Done
Do you want to Apply Another function, Press 'Y' or 'y' for yes, any other key to stop :y

-----------------------------------------------------------
Please enter a number:
Press 1 to get products that has less quantity than a certain value
Press 2 to Get Product with the highest sales
Press 3 to Apply 50% discount for products that has quantity less than 5
Press 4 to count the number of products with prices less than a certain amount
Press 5 to Display all the products
4
Enter the amount :50
The number of products with price less than 50 is 3
Do you want to Apply Another function, Press 'Y' or 'y' for yes, any other key to stop :y

-----------------------------------------------------------
Please enter a number:
Press 1 to get products that has less quantity than a certain value
Press 2 to Get Product with the highest sales
Press 3 to Apply 50% discount for products that has quantity less than 5
Press 4 to count the number of products with prices less than a certain amount
Press 5 to Display all the products
5
Serial num :1231
quantity:3
sales :20
price :25
Serial num :2313
quantity:10
sales :10
price :40
Serial num :3331
quantity:4
sales :12
price :20
Serial num :3345
quantity:14
sales :15
price :100
Serial num :1212
quantity:20
sales :30
price :55
Do you want to Apply Another function, Press 'Y' or 'y' for yes, any other key to stop :n
Program ended with exit code: 0
```

Sheet 7 will be available on Drive