

FUNDAMENTALS OF STRUCTURED PROGRAMMING

Lecture 9

Pointers I

Course Coordinator: Prof. Zaki Taha Fayed

Presented by: Dr. Sally Saad

SallySaad@gmail.com

DropBox folder link

<https://www.dropbox.com/sh/85vnrgkfqgrzhwn/AABdwKLJZqZs26a7u-y0AFwia?dl=0>

Credits to Dr.Salma Hamdy for content preparation

Quote of the Day!

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



-Martin Fowler-

Pointers I

Contents

1. Pointers

Variable, declaration, initialization, dereferencing operator, address operator, pointers in assignment.

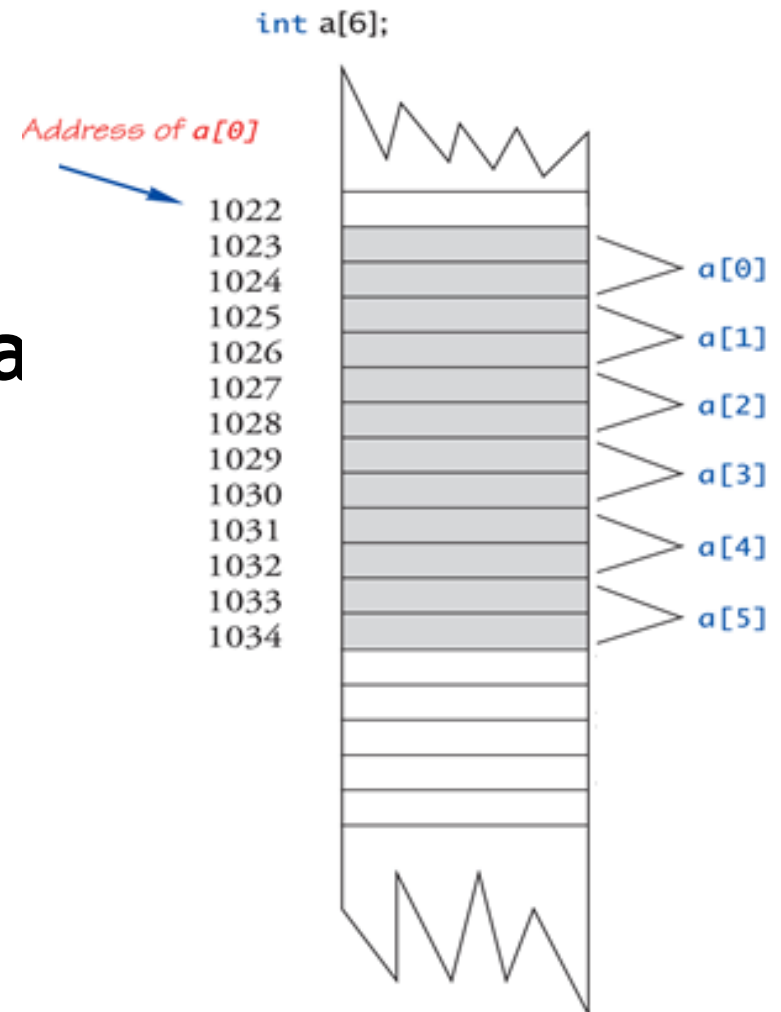
2. Memory Management and the Heap

Deleting a pointer variable

3. Examples

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

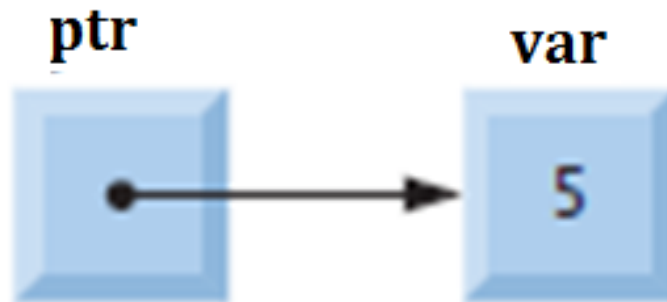
```
int count = 7;
```

```
int *p1, *p2, v1, v2;    double *p;
```

`p1`, and `p2` hold pointers to an 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.

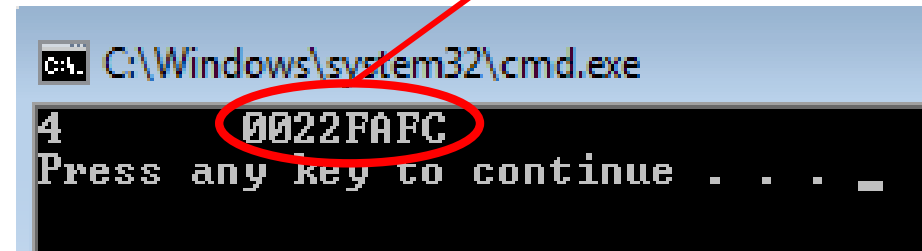
1. Pointers – (cont.)

Address operator

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

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

A HEX representation of a memory location



```
C:\Windows\system32\cmd.exe
4
0022FAFC
Press any key to continue . . . _
```

- 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

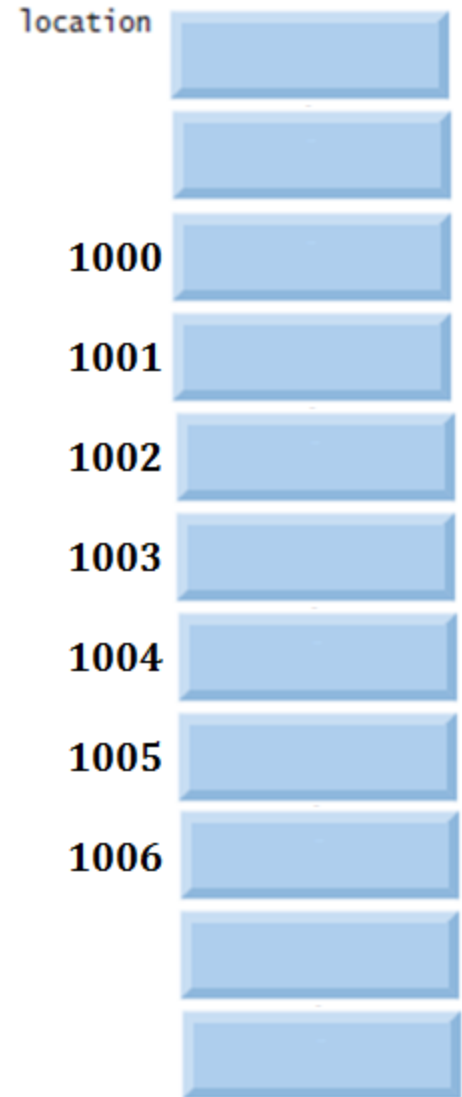
```
// variables
```

```
int num = 4;
```

```
int* ptr = & num;
```

```
// output
```

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

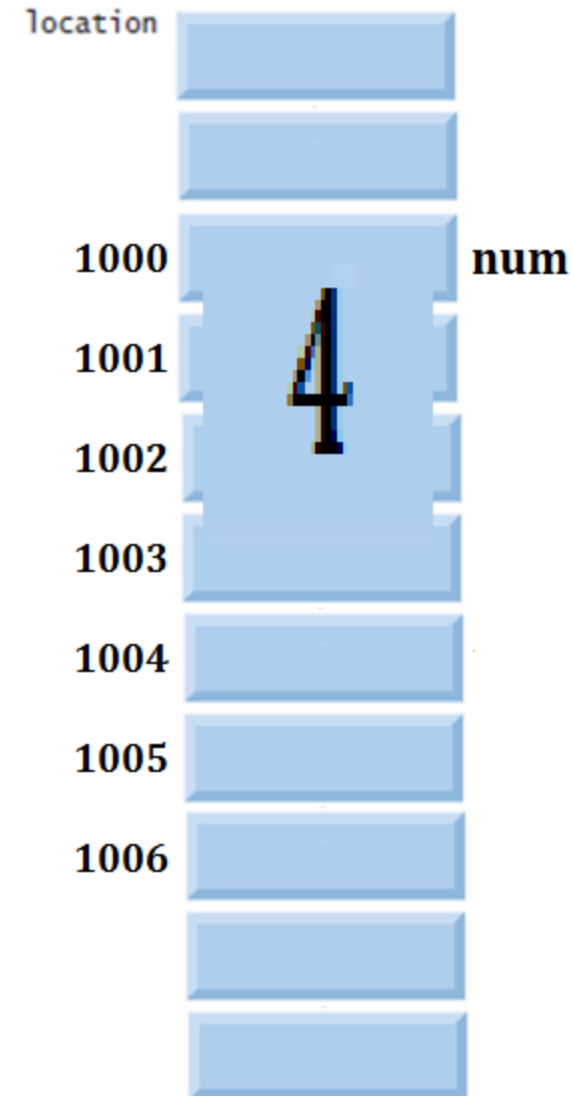


1. Pointers – (cont.)

Address operator

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

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

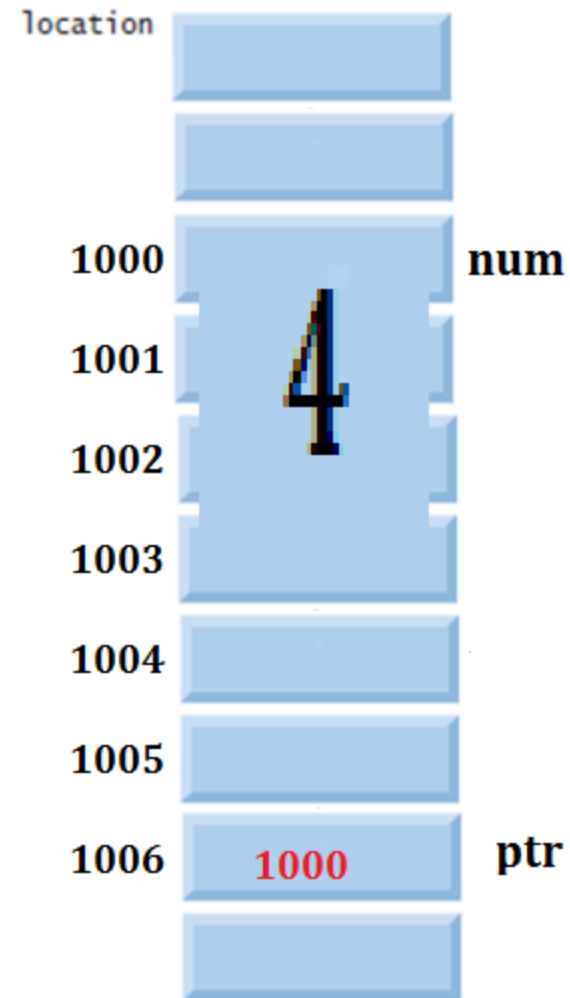


1. Pointers – (cont.)

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.

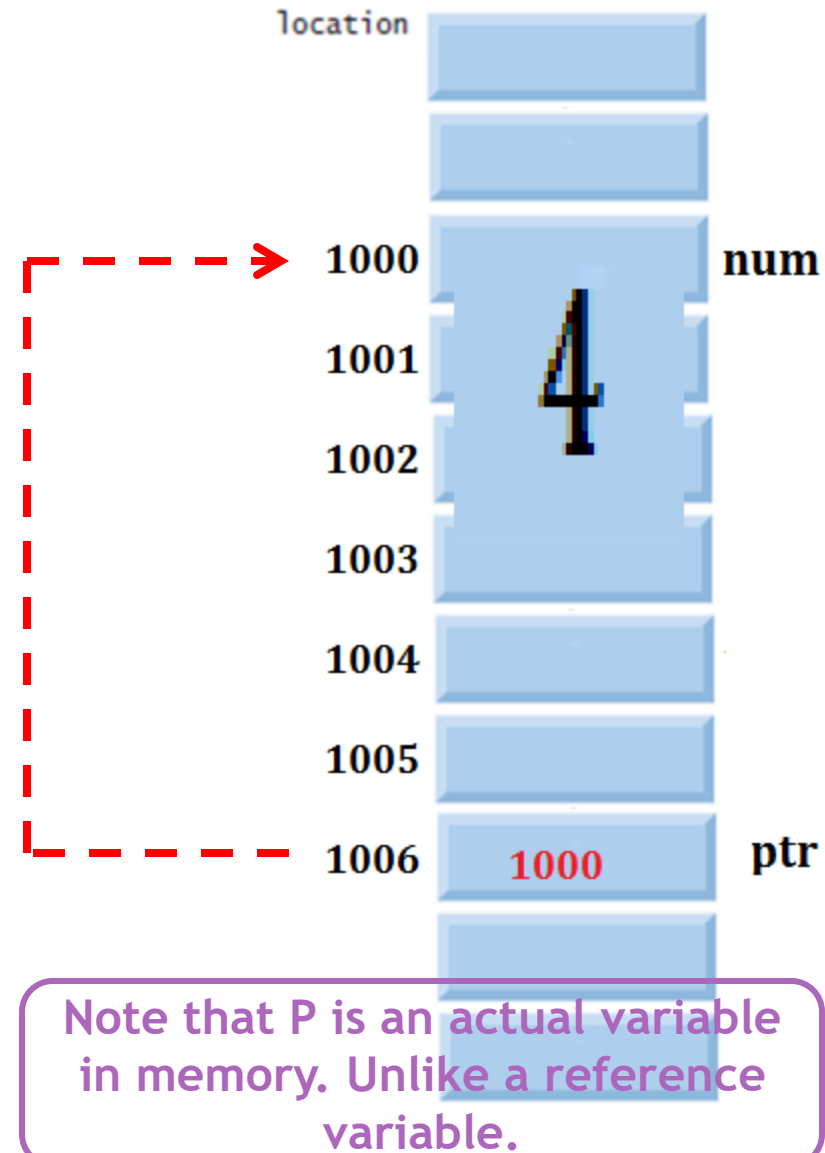
1. Pointers – (cont.)

Address operator

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

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

What is the size
of a pointer
variable?



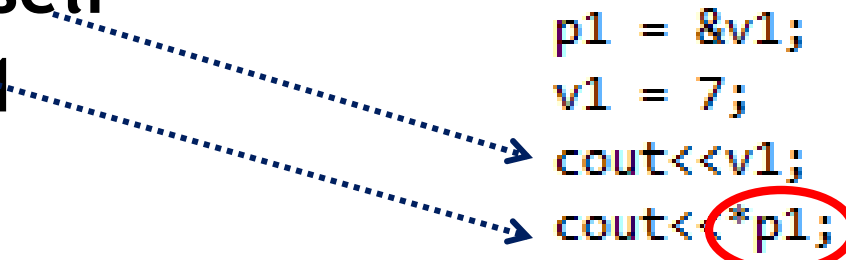
1. Pointers – (cont.)

Dereferencing operator

- Two ways to refer to v1 now:

- Variable v1 itself
- Via pointer p1

```
int *p1, *p2, v1, v2;  
p1 = &v1;  
v1 = 7;  
cout<<v1;  
cout<<(*p1);
```



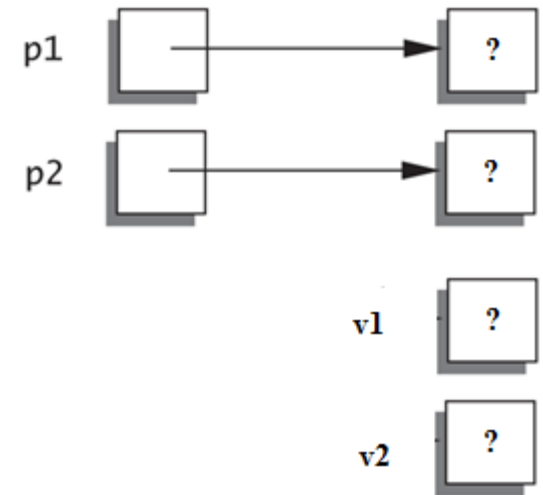
NOTE that this is NOT a declaration.

- The dereference operator ***** retrieves the value pointed to by the variable.
 - Pointer variable “dereferenced” means: “**get data that ptr 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;
```

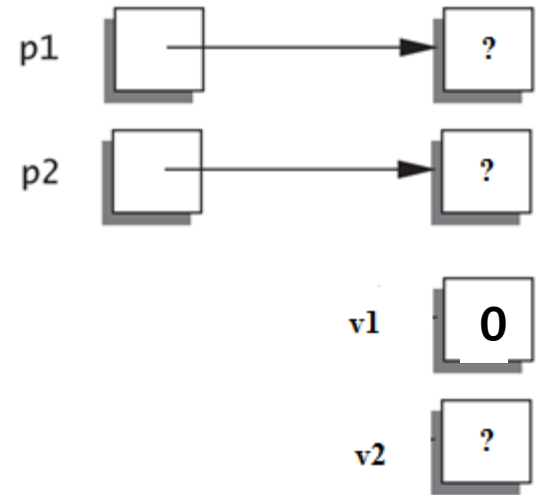
```
v1 = 0;
```

```
p1 = &v1;
```

```
*p1 = 42;
```

```
cout << v1 << endl;
```

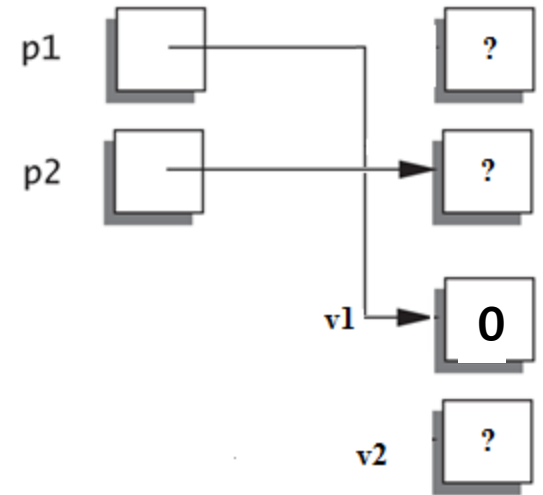
```
cout << *p1 << endl;
```



1. Pointers – (cont.)

Dereferencing operator

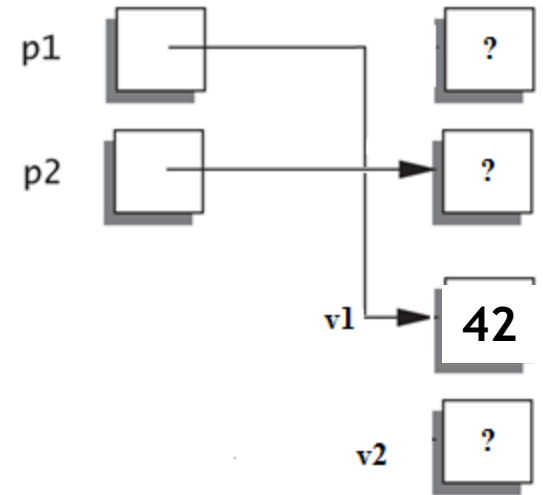
```
int *p1, *p2, v1, v2;  
v1 = 0;  
p1 = &v1;  
*p1 = 42;  
cout << v1 << endl;  
cout << *p1 << endl;
```



1. Pointers – (cont.)

Dereferencing operator

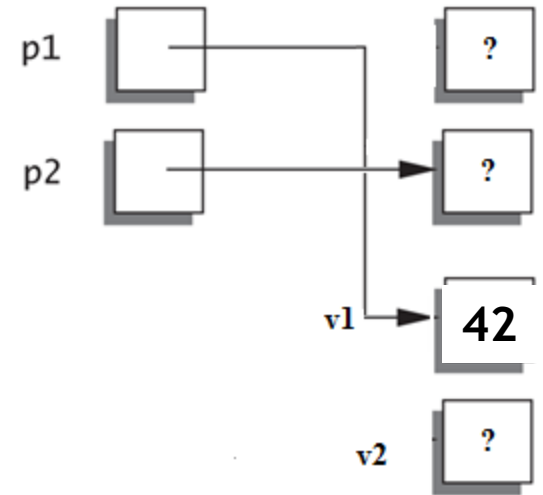
```
int *p1, *p2, v1, v2;  
v1 = 0;  
p1 = &v1;  
● *p1 = 42;  
cout << v1 << endl;  
cout << *p1 << endl;
```



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.)

- Pointer is an address and address is an integer, 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++;`
- A pointer variable can be assigned to any variable type.
Pointer to integer, to float, to double, to struct, to pointer, to array.
- A pointer variable, like any other variable, can be assigned a value, or another variable of the same type.

1. Pointers – (cont.)

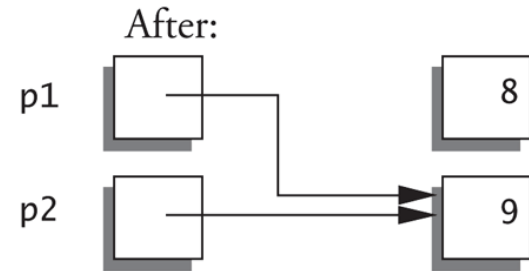
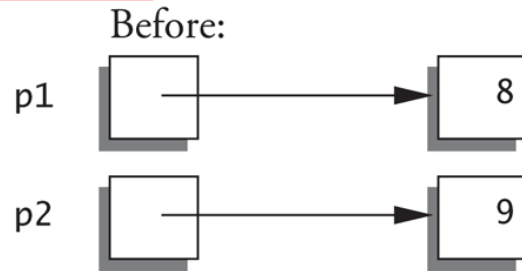
Pointers in Assignments

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

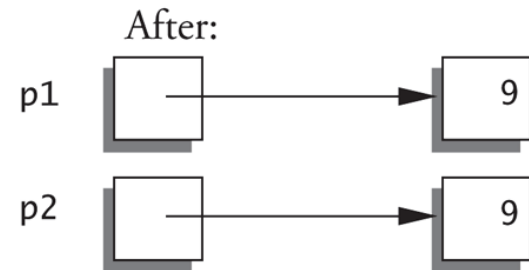
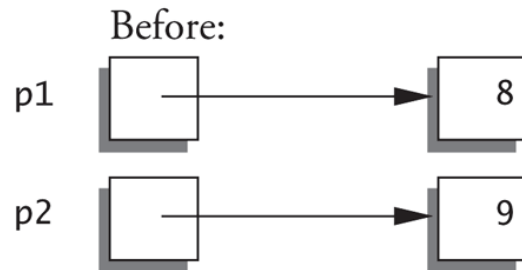
POINTER ASSIGNMENTS GRAPHIC: USES OF THE ASSIGNMENT OPERATOR WITH POINTER VARIABLES

Display 10.1 Uses of the Assignment Operator with Pointer Variables

`p1 = p2;`



`*p1 = *p2;`



1. Pointers – (cont.)

Dynamic Variables

- Since pointers can refer to variables...
 - No "real" need to have a standard identifier.
- Can dynamically allocate variables
 - Operator **new** creates variables
 - No identifiers to refer to them
 - Just a pointer!
- **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 //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";
```

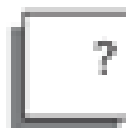
```
21    return 0;
```

```
22 }
```

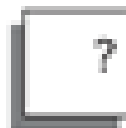
(a)

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

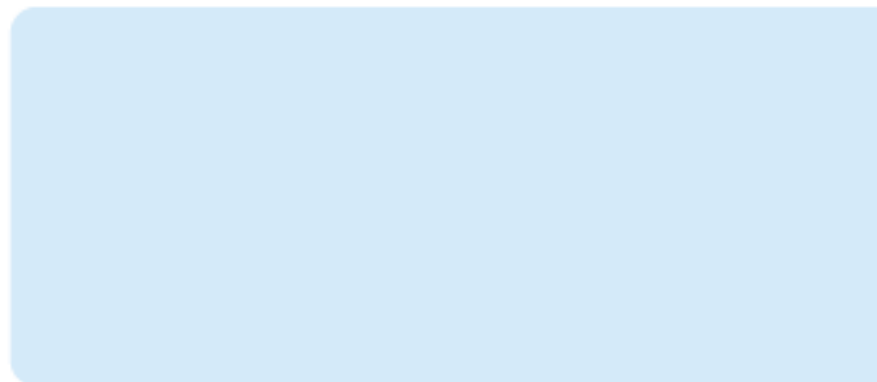
p1



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;
```

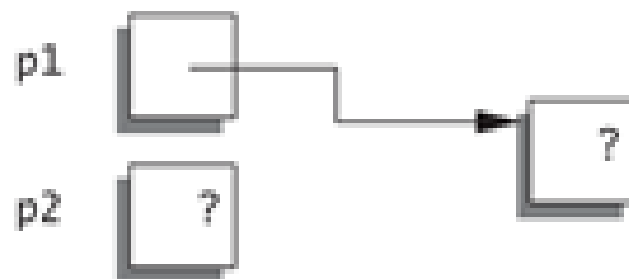
```
20    cout << "Hope you got the point of this example!\n";
```

```
21    return 0;
```

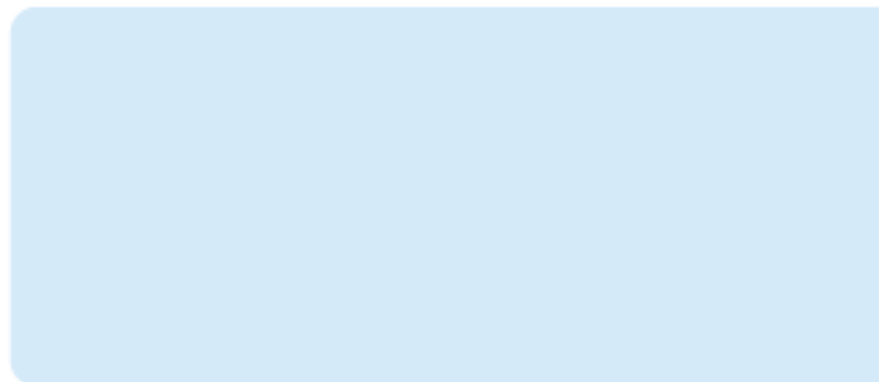
```
22 }
```

(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;

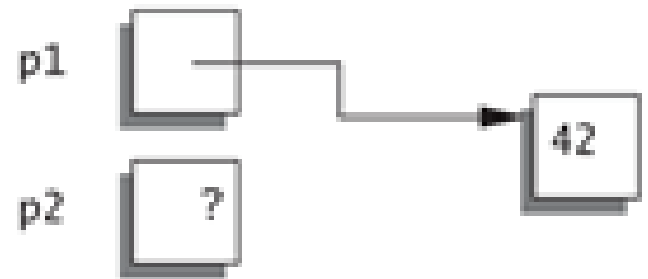
13    *p2 = 53;
14    cout << "*p1 == " << *p1 << endl;
15    cout << "*p2 == " << *p2 << endl;

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

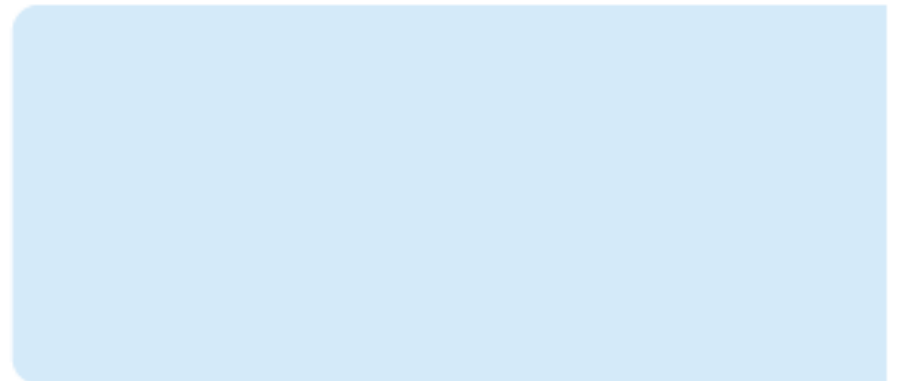
20    cout << "Hope you got the point of this example!\n";
21    return 0;
22 }

```

(c)
*p1 = 42;



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;

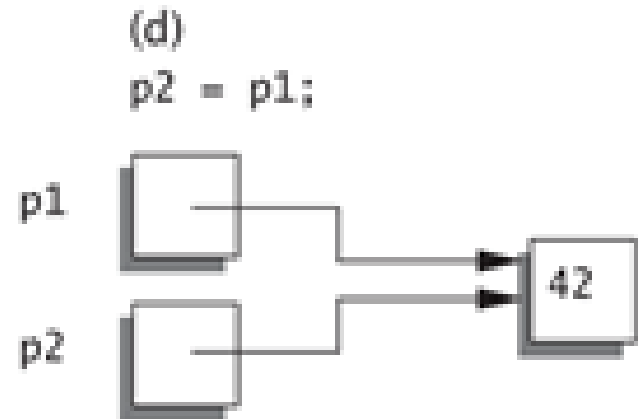
8     p1 = new int;
9     *p1 = 42;
10    p2 = p1;
11    cout << "*p1 == " << *p1 << endl;
12    cout << "*p2 == " << *p2 << endl;

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

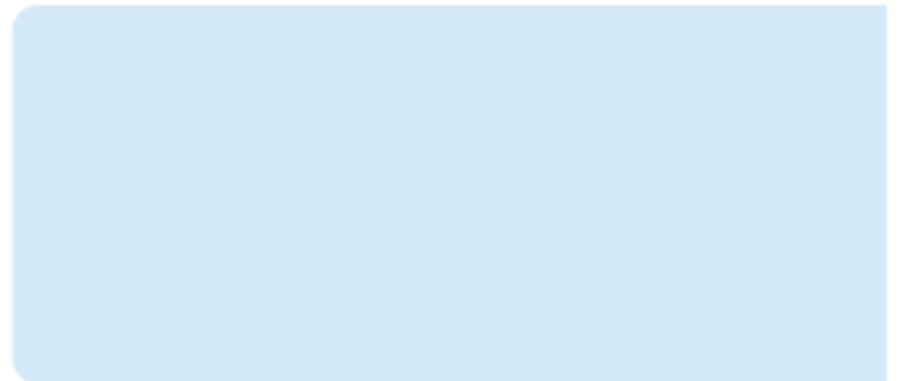
16    p1 = new int;
17    *p1 = 88;
18    cout << "*p1 == " << *p1 << endl;
19    cout << "*p2 == " << *p2 << endl;

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;

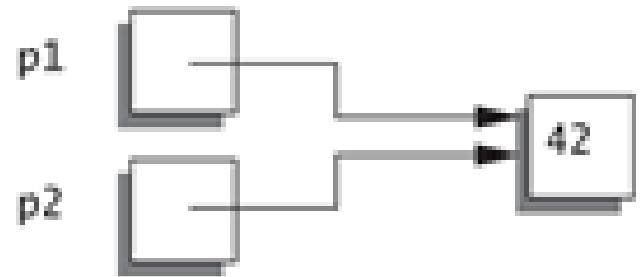
16    p1 = new int;
17    *p1 = 88;
18    cout << "*p1 == " << *p1 << endl;
19    cout << "*p2 == " << *p2 << endl;

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

```

(d)

p2 = p1;



SAMPLE DIALOGUE

*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;
```

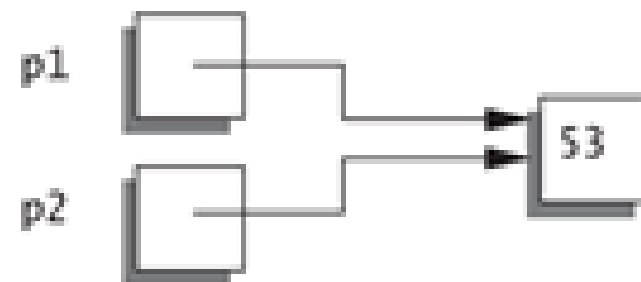
```
20    cout << "Hope you got the point of this example!\n";
```

```
21    return 0;
```

```
22 }
```

(e)

*p2 = 53;



SAMPLE DIALOGUE

*p1 == 42

*p2 == 42

```
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;
```

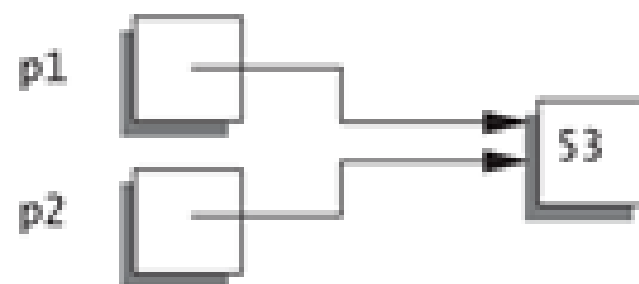
```
20    cout << "Hope you got the point of this example!\n";
```

```
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;

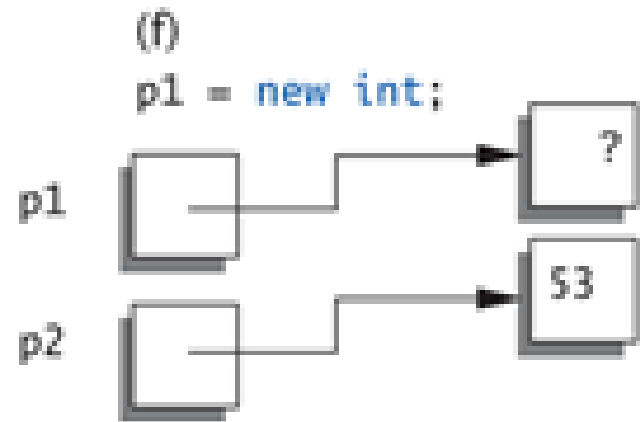
8     p1 = new int;
9     *p1 = 42;
10    p2 = p1;
11    cout << "*p1 == " << *p1 << endl;
12    cout << "*p2 == " << *p2 << endl;

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;

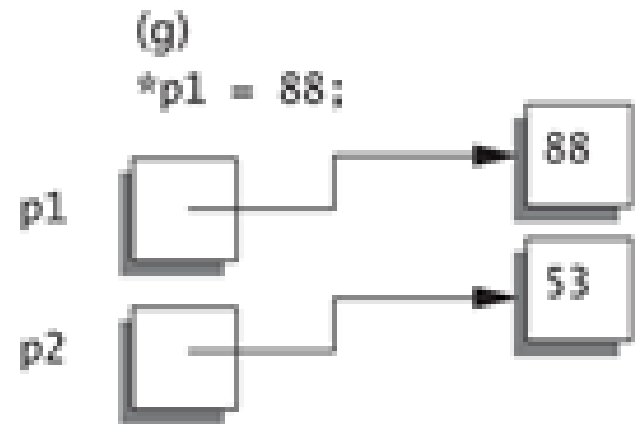
8     p1 = new int;
9     *p1 = 42;
10    p2 = p1;
11    cout << "*p1 == " << *p1 << endl;
12    cout << "*p2 == " << *p2 << endl;

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;
```

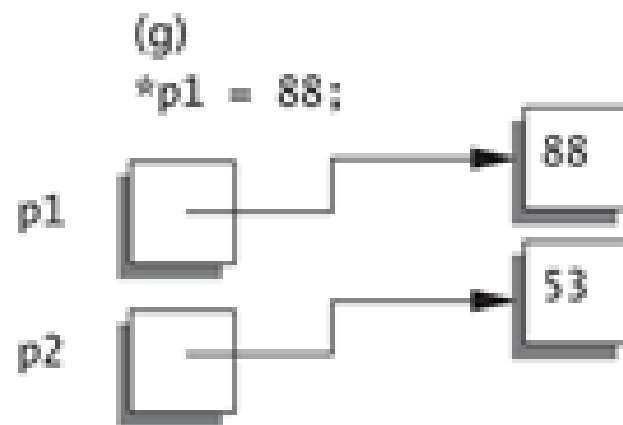
```
18    cout << "*p1 == " << *p1 << endl;
```

```
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(17);    //Initializes *iPtr to 17
```

```
double *dPtr;
```

```
dPtr = new double(98.6); // Initializes *dPtr to 98.6.
```

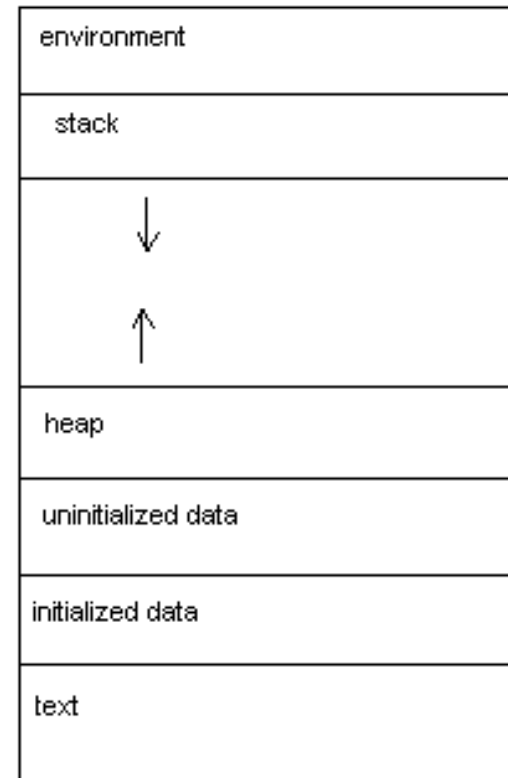
```
int num(5);    // exactly as int num = 5;
```

```
int* ptr = new int(num);
```

2. Memory Management

The Heap (freestore)

- Reserved for dynamically-allocated variables.
- All new dynamic variables consume memory in freestore.
- If too many → could use all freestore memory.
- Future "new" operations will fail if freestore is "full".



Virtual memory organization

2. Memory Management – (cont.)

The Heap (freestore) Size

- **Varies with implementations**
- **Typically large**
 - Most programs won't use all memory
- **Memory management**
 - Still good practice
 - Solid software engineering principle
 - Memory IS finite
 - Regardless of how much there is!

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);
} // end if
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 freestore

```
// 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
```

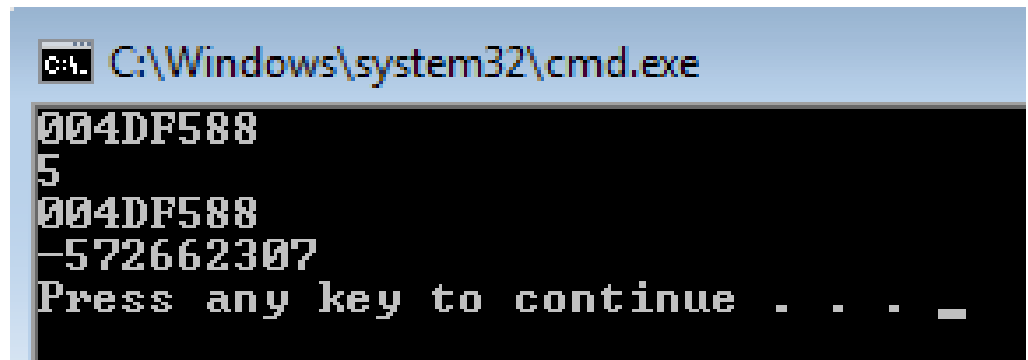
```
cout<<ptr<<endl;
```

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

```
delete ptr;
```

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

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



```
C:\Windows\system32\cmd.exe
004DF588
5
004DF588
-572662307
Press any key to continue . . . _
```

2. Memory Management – (cont.)

Dangling Pointers

- Avoid dangling pointers by assigning pointer to NULL after delete:

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

```
delete ptr;  
ptr = NULL;
```

Same as:

```
ptr= nullptr;
```

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

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

⦿ **Local** variables

- Declared within function definition
- Not dynamic
 - Created when function is called
 - Destroyed when function call completes
- Often called "**automatic**" variables
 - Properties controlled for you

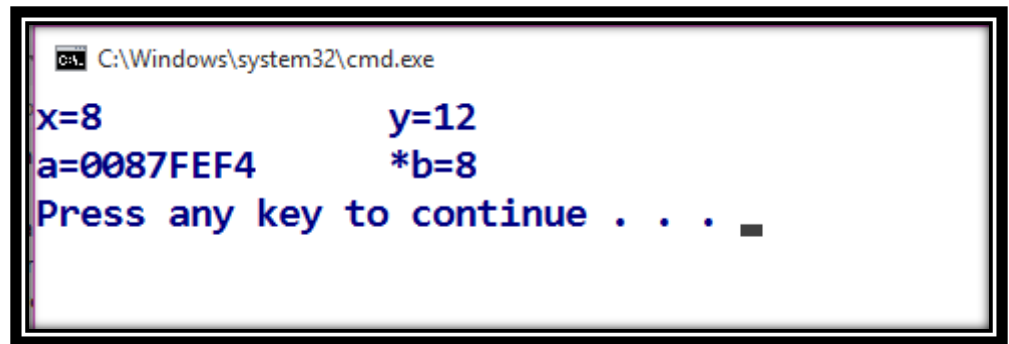
3. Examples

Trace Exercise 1

- What is the output of the following code segment:

```
int x  = 8;  
int y  = 9;  
int *a = &x;  
int *b = &y;  
a      = b;  
b      = &x;  
*a     = 12;
```

```
cout<<x<<endl<<y<<endl;  
cout<<a<<endl<<*b<<endl;
```



A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window displays the output of the code segment: "x=8", "y=12", "a=0087FEF4", and "*b=8". Below this output, it says "Press any key to continue . . .".

```
C:\Windows\system32\cmd.exe  
x=8                y=12  
a=0087FEF4        *b=8  
Press any key to continue . . .
```

3. Examples – (cont.)

Trace Exercise 2

- What is the output of the following code segment:

```
int i=3,*j;
```

```
j=&i;
```

```
cout<<i**j*i+*j<<endl;
```

30

3. Examples – (cont.)

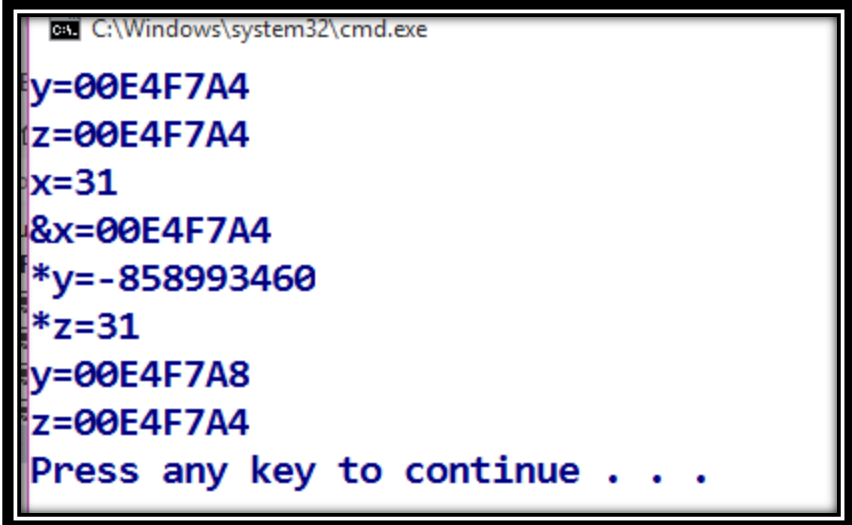
```
int x=30,*y,*z;  
y=&x; //y has 30  
z=y; //z has 30  
cout<<"y="<<y<<endl<<"z="<<z;  
*y++; //increment address pointed to by z  
x++;  
cout<<endl<<"x="<<x<<endl<<"&x="<<&x<<endl<<"*y="<<*y<<endl<<"*z="<<*z<<endl;  
cout<<"y="<<y<<endl<<"z="<<z<<endl;
```

3. Examples – (cont.)

Trace Exercise 3

- What is the output of the following code segment:

```
int x=30,*y,*z;  
y=&x; //y has 30  
z=y; //z has 30  
cout<<"y="<<y<<endl<<"z="<<z;  
*y++; //increment address pointed to by z  
x++;  
cout<<endl<<"x="<<x<<endl<<"&x="<<&x<<endl<<"*y="<<*y<<endl<<"*z="<<*z<<endl;  
cout<<"y="<<y<<endl<<"z="<<z<<endl;
```



A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The output of the C++ program is displayed in blue text. The output shows the memory addresses for y and z, the value of x, the address of x, the value of *y, and the value of *z. The output is as follows:

```
y=00E4F7A4  
z=00E4F7A4  
x=31  
&x=00E4F7A4  
*y=-858993460  
*z=31  
y=00E4F7A8  
z=00E4F7A4  
Press any key to continue . . .
```

4. Summary

POINTER VARIABLE DECLARATIONS

A variable that can hold pointers to other variables of type *Type_Name* is declared similar to the way you declare a variable of type *Type_Name*, except that you place an asterisk at the beginning of the variable name.

SYNTAX

```
Type_Name *Variable_Name1, *Variable_Name2, . . . ;
```

EXAMPLE

```
double *pointer1, *pointer2;
```

4. Summary – (cont.)

THE * AND & OPERATORS

The * operator in front of a pointer variable produces the variable to which it points. When used this way, the * operator is called the **dereferencing operator**.

The operator & in front of an ordinary variable produces the address of that variable; that is, it produces a pointer that points to the variable. The & operator is simply called the **address-of operator**.

For example, consider the declarations

```
double *p, v;
```

The following sets the value of p so that p points to the variable v:

```
p = &v;
```

*p produces the variable pointed to by p, so after the above assignment, *p and v refer to the same variable. For example, the following sets the value of v to 9.99, even though the name v is never explicitly used:

```
*p = 9.99;
```

4. Summary – (cont.)

Pointer Variables Used with =

If p1 and p2 are pointer variables, then the statement

```
p1 = p2;
```

changes the value of p1 so that it is the memory address (pointer) in p2. A common way to think of this is that the assignment will change p1 so that it points to the same thing to which p2 is currently pointing.

4. Summary – (cont.)

THE new OPERATOR

The new operator creates a new dynamic variable of a specified type and returns a pointer that points to this new variable. For example, the following creates a new dynamic variable of type MyType and leaves the pointer variable p pointing to this new variable:

```
MyType *p;  
p = new MyType;
```

If the type is a class type, the default constructor is called for the newly created dynamic variable. You can specify a different constructor by including arguments as follows:

```
MyType *mtPtr;  
mtPtr = new MyType(32.0, 17); // calls MyType(double, int);
```

A similar notation allows you to initialize dynamic variables of nonclass types, as illustrated below:

```
int *n;  
n = new int(17); // initializes *n to 17
```

With earlier C++ compilers, if there was insufficient available memory to create the new variable, then new returned a special pointer named NULL. The C++ standard provides that if there is insufficient available memory to create the new variable, then the new operator, by default, terminates the program.

4. Summary – (cont.)

NULL

NULL is a special constant pointer value that is used to give a value to a pointer variable that would not otherwise have a value. NULL can be assigned to a pointer variable of any type. The identifier NULL is defined in a number of libraries, including `<iostream>`. (The constant NULL is actually the integer 0.)

4. Summary – (cont.)

THE delete OPERATOR

The `delete` operator eliminates a dynamic variable and returns the memory that the dynamic variable occupied to the freestore. The memory can then be reused to create new dynamic variables. For example, the following eliminates the dynamic variable pointed to by the pointer variable `p`:

```
delete p;
```

After a call to `delete`, the value of the pointer variable, like `p` above, is undefined. (A slightly different version of `delete`, discussed later in this chapter, is used when the dynamically allocated variable is an array.)

4. Summary – (cont.)

Pitfall

DANGLING POINTERS

When you apply `delete` to a pointer variable, the dynamic variable to which it is pointing is destroyed. At that point, the value of the pointer variable is undefined, which means that you do not know where it is pointing. Moreover, if some other pointer variable was pointing to the dynamic variable that was destroyed, then this other pointer variable is also undefined. These undefined pointer variables are called **dangling pointers**. If `p` is a dangling pointer and your program applies the dereferencing operator `*` to `p` (to produce the expression `*p`), the result is unpredictable and usually disastrous. Before you apply the dereferencing operator `*` to a pointer variable, you should be certain that the pointer variable points to some variable.

C++ has no built-in test to check whether a pointer variable is a dangling pointer. One way to avoid dangling pointers is to set any dangling pointer variable equal to `NULL`. Then your program can test the pointer variable to see if it is equal to `NULL` before applying the dereferencing operator `*` to the pointer variable. When you use this technique, you follow a call to `delete` by code that sets all dangling pointers equal to `NULL`. Remember, other pointer variables may become dangling pointers besides the one pointer variable used in the call to `delete`, so be sure to set *all* dangling pointers to `NULL`. It is up to the programmer to keep track of dangling pointers and set them to `NULL` or otherwise ensure that they are not dereferenced.

4. Summary – (cont.)

Tip

DEFINE POINTER TYPES

You can define a pointer type name so that pointer variables can be declared like other variables without the need to place an asterisk in front of each pointer variable. For example, the following defines a type called `IntPtr`, which is the type for pointer variables that contain pointers to `int` variables:

```
typedef int* IntPtr;
```

Thus, the following two pointer variable declarations are equivalent:

```
IntPtr p;
```

and

```
int *p;
```

You can use `typedef` to define an alias for any type name or definition. For example, the following defines the type name `Kilometers` to mean the same thing as the type name `double`:

```
typedef double Kilometers;
```

Once you have given this type definition, you can define a variable of type `double` as follows:

```
Kilometers distance;
```

Renaming existing types this way can occasionally be useful. However, our main use of `typedef` will be to define types for pointer variables.

Keep in mind that a `typedef` does not produce a new type but is simply an alias for the type definition. For example, given the previous definition of `Kilometers`, a variable of type

Kilometers may be substituted for a parameter of type double. Kilometers and double are two names for the same type.

There are two advantages to using defined pointer type names, such as IntPtr defined previously. First, it avoids the mistake of omitting an asterisk. Remember, if you intend p1 and p2 to be pointers, then the following is a mistake:

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

Since the * was omitted from the p2, the variable p2 is just an ordinary int variable, not a pointer variable. If you get confused and place the * on the int, the problem is the same but is more difficult to notice. C++ allows you to place the * on the type name, such as int*, so that the following is legal:

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

Although the above is legal, it is misleading. It looks like both p1 and p2 are pointer variables, but in fact only p1 is a pointer variable; p2 is an ordinary int variable. As far as the C++ compiler is concerned, the * that is attached to the identifier int may as well be attached to the identifier p1. One correct way to declare both p1 and p2 to be pointer variables is

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

An easier and less error-prone way to declare both p1 and p2 to be pointer variables is to use the defined type name IntPtr as follows:

```
IntPtr p1, p2;
```

The second advantage of using a defined pointer type, such as IntPtr, is seen when you define a function with a call-by-reference parameter for a pointer variable. Without the defined pointer type name, you would need to include both an * and an & in the declaration for the function, and the details can get confusing. If you use a type name for the pointer type, then a call-by-reference parameter for a pointer type involves no complications. You define a call-by-reference parameter for a defined pointer type just like you define any other call-by-reference parameter. Here's an example:

```
void sampleFunction(IntPtr& pointerVariable);
```

4. Summary – (cont.)

TYPE DEFINITIONS

You can assign a name to a type definition and then use the type name to declare variables. This is done with the keyword `typedef`. These type definitions are normally placed outside the body of the `main` part of your program and outside the body of other functions, typically near the start of a file. That way the `typedef` is global and available to your entire program. We will use type definitions to define names for pointer types, as shown in the example below.

SYNTAX

```
typedef Known_Type_Definition New_Type_Name;
```

EXAMPLE

```
typedef int* IntPtr;
```

The type name `IntPtr` can then be used to declare pointers to dynamic variables of type `int`, as in the following example:

```
IntPtr pointer1, pointer2;
```

4. Summary – (cont.)



Common Programming Error 8.1

*Assuming that the * used to declare a pointer distributes to all names in a declaration's comma-separated list of variables can lead to errors. Each pointer must be declared with the * prefixed to the name (with or without spaces in between). Declaring only one variable per declaration helps avoid these types of errors and improves program readability.*



Good Programming Practice 8.1

Although it isn't a requirement, including the letters Ptr in a pointer variable name makes it clear that the variable is a pointer and that it must be handled accordingly.



Common Programming Error 8.2

Dereferencing an uninitialized pointer could cause a fatal execution-time error, or it could accidentally modify important data and allow the program to run to completion, possibly with incorrect results.



Common Programming Error 8.3

An attempt to dereference a variable that is not a pointer is a compilation error.



Common Programming Error 8.4

Dereferencing a null pointer is often a fatal execution-time error.



Error-Prevention Tip 8.1

Initialize pointers to prevent pointing to unknown or uninitialized areas of memory.

Thank
you.

A yellow rectangular sticky note is centered on a white background. The words "Thank" and "you." are written in a black, sans-serif font, one above the other. Two small red dots are positioned above the 'i' in "Thank" and the 'i' in "you.". A red, curved line is drawn below the word "you.", resembling a smiley face.