Misr International University (MIU)

Faculty of Computer Science

Computer Science Program

CSC 105: Introduction to Programming & Problem Solving

Dr. Ayman Ezzat and Dr. Ashraf AbdelRaouf

Lecture 5

Pointers

Objectives

In this Lecture, you will:

- Learn about the pointer data type and pointer variables
- Explore how to declare and manipulate pointer variables
- Learn about the address of operator and the dereferencing operator
- Discover dynamic variables
- Explore how to use the `new` and `delete` operators to manipulate dynamic variables
- Learn about pointer arithmetic
- Discover dynamic arrays

Pointer Data Type and Pointer Variables

- Pointer variable: content is a memory address
- There is no name associated with the pointer data type in C++

Declaring Pointer Variables

- Syntax:

```
dataType *identifier;
```

- Examples:

```
int *p;
char *ch;
```

- These statements are equivalent:

```
int  *p;
int*  p;
int * p;
```

Declaring Pointer Variables (continued)

- In the statement:

```
int* p, q;
```

  only `p` is the pointer variable, not `q`; here `q` is an `int` variable

- To avoid confusion, attach the character `*` to the variable name:

```
int   *p, q;
int   *p, *q;
```

Address of Operator (&)

- The ampersand, &, is called the address of operator
- The address of operator is a unary operator that returns the address of its operand

Dereferencing Operator (*)

- When used as a unary operator, * is the dereferencing operator or indirection operator
  - Refers to object to which its operand points
- Example:

```
int x = 25;
int *p;
p = &x;    //store the address of x in p
```

  - To print the value of x, using p:

```
cout << *p << endl;
```

  - To store a value in x, using p:

```
*p = 55;
```

```
int *p;
int num;
```

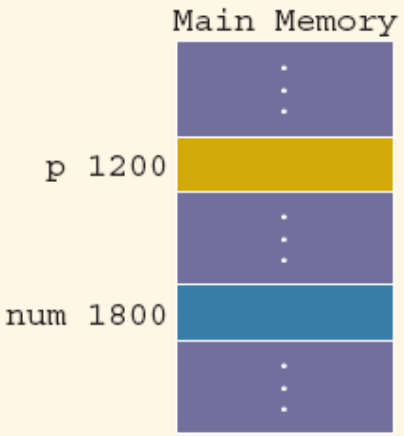


FIGURE 14-1 Main memory, p, and num

```
num = 78;
```

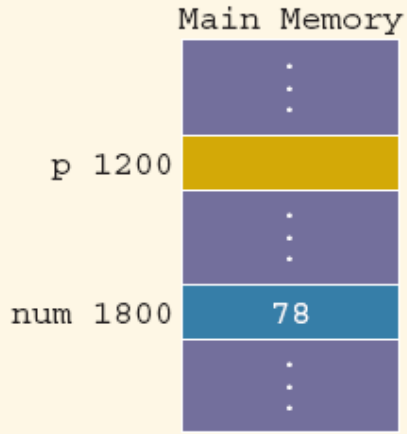


FIGURE 14-2 num after the statement num = 78; executes

```
p = &num;
```

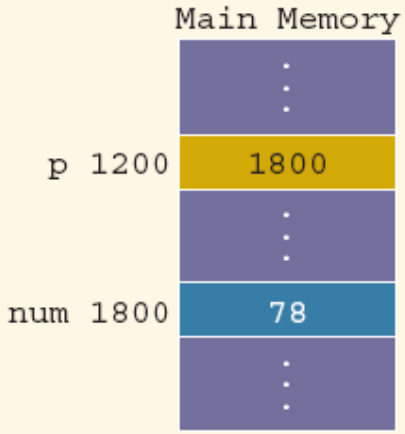


FIGURE 14-3 p after the statement p = # executes

```
*p = 24;
```

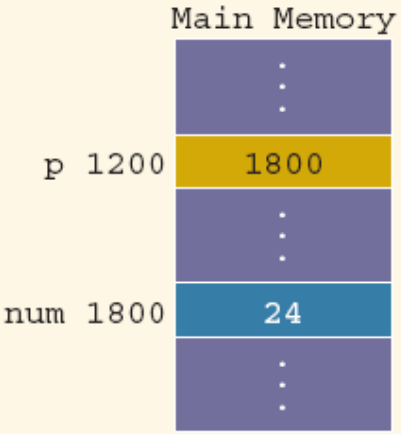


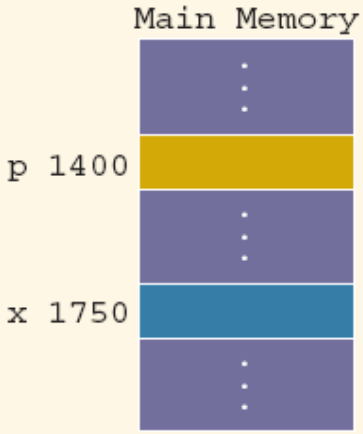
FIGURE 14-4 *p and num after the statement *p = 24; executes

EXAMPLE 14-1

Consider the following statements:

```
int *p;
int x;
```

Allocates memory for `p` only, not for `*p`



	Value
&p	1400
p	??? (unknown)
*p	Does not exist (undefined)
&x	1750
x	??? (unknown)

```
x = 50;
p = &x;
*p = 38;
```

	Value
&p	1400
p	??? (unknown)
*p	Does not exist (undefined)
&x	1750
x	50

	Value
&p	1400
p	1750
*p	50
&x	1750
x	50

	Value
&p	1400
p	1750
*p	38
&x	1750
x	38

```
int _tmain(int argc, _TCHAR* argv[])
{
    int N=30;
    int *NPtr;
    NPtr=&N;
    cout<<"N=\t"<<N<<endl;
    cout<<"&N=\t"<<&N<<endl;
    cout<<"*NPtr=\t"<<*NPtr<<endl;
    cout<<"NPtr=\t"<<NPtr<<endl;
    N=200;
    cout<<"N=\t"<<N<<endl;
    cout<<"&N=\t"<<&N<<endl;
    cout<<"*NPtr=\t"<<*NPtr<<endl;
    cout<<"NPtr=\t"<<NPtr<<endl;
    cout<<"&NPtr=\t"<<&NPtr<<endl;
    return 0;
}
```

```
N=      30
&N=     002AFC54
*NPtr=  30
NPtr=   002AFC54
N=      200
&N=     002AFC54
*NPtr=  200
NPtr=   002AFC54
&NPtr=  002AFC48
Press any key to continue . . .
```

Initializing Pointer Variables

- C++ does not automatically initialize variables
- Pointer variables must be initialized if you do not want them to point to anything
  - Initialized using the constant value 0
    - Called the **null pointer**
    - Example: `p = 0;`
  - Or, use `NULL` named constant: `p = NULL;`
  - The number 0 is the only number that can be directly assigned to a pointer variable

Dynamic Variables

- Dynamic variables: created during execution
- C++ creates dynamic variables using pointers
- Two operators, `new` and `delete`, to create and destroy dynamic variables
  - `new` and `delete` are reserved words

Operator new

- new has two forms:

```
new dataType;                //to allocate a single variable
new dataType[intExp];        //to allocate an array of variables
```

  - where intExp is any expression evaluating to a positive integer
- new allocates memory (a variable) of the designated type and returns a pointer to it
  - The address of the allocated memory
- The allocated memory is uninitialized

Operator new (continued)

```
int *p;
char *q;
int x;
```

- The statement: `p = &x;`
  - Stores address of `x` in `p`
    - However, no new memory is allocated
- The statement: `p = new int;`
  - Creates a variable during program execution somewhere in memory, and stores the address of the allocated memory in `p`
    - To access allocated memory: `*p`

```
int *p;                 //p is a pointer of type int
char *name;             //name is a pointer of type char
string *str;            //str is a pointer of type string

p = new int;            //allocates memory of type int
                        //and stores the address of the
                        //allocated memory in p
*p = 28;                //stores 28 in the allocated memory

name = new char[5];     //allocates memory for an array of
                        //five components of type char and
                        //stores the base address of the array
                        //in name
strcpy(name, "John");   //stores John in name

str = new string;       //allocates memory of type string
                        //and stores the address of the
                        //allocated memory in str
*str = "Sunny Day";     //stores the string "Sunny Day" in
                        //the memory pointed to by str
```

Operator new (continued)

- `new` allocates memory space of a specific type and returns the (starting) address of the allocated memory space
- If `new` is unable to allocate the required memory space, it throws `bad_alloc` exception
  - If this exception is not handled, it terminates the program with an error message

Operator delete

```
p = new int;        //Line 1
*p = 54;            //Line 2
p = new int;        //Line 3
*p = 73;            //Line 4
```

FIGURE 14-8 p after the execution of p = new int;

FIGURE 14-9 p and *p after the execution of *p = 54;

Operator delete (continued)

```
p = new int;          //Line 1
*p = 54;              //Line 2
p = new int;          //Line 3
*p = 73;              //Line 4
```

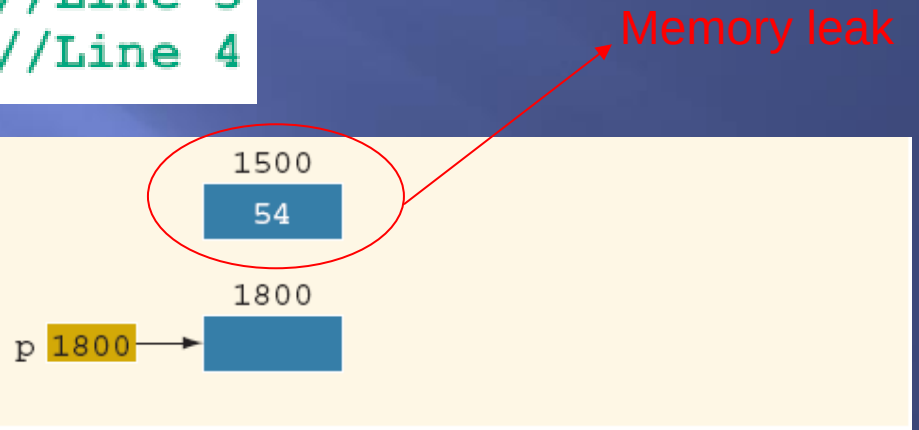


FIGURE 14-10 p after the execution of p = new int;

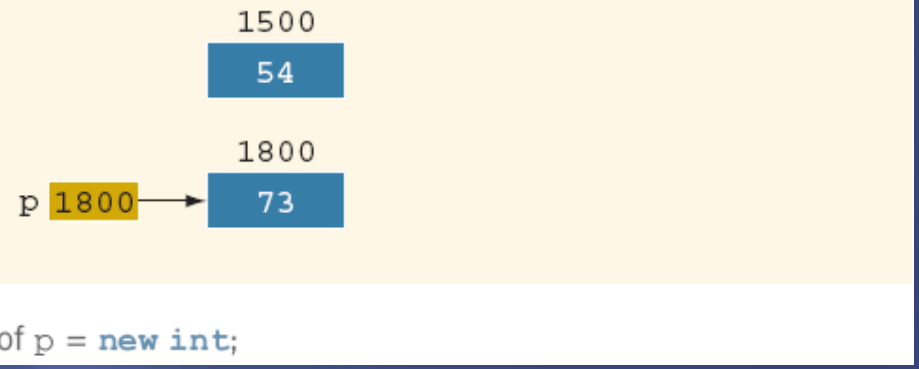


FIGURE 14-11 p after the execution of p = new int;

Operator delete (continued)

- To avoid memory leak, when a dynamic variable is no longer needed, destroy it
  - Deallocate its memory
- delete is used to destroy dynamic variables
- Syntax:

```
delete pointerVariable;     //to deallocate a single
                            //dynamic variable
delete [] pointerVariable; //to deallocate a dynamically
                            //created array
```

  - Tip: to avoid **dangling pointers**, set variable to NULL afterwards

Operations on Pointer Variables

- Assignment: value of one pointer variable can be assigned to another pointer of same type
- Relational operations: two pointer variables of same type can be compared for equality, etc.
- Some limited arithmetic operations:
  - Integer values can be added and subtracted from a pointer variable
  - Value of one pointer variable can be subtracted from another pointer variable

Operations on Pointer Variables (continued)

- Examples:

```
int *p, *q;
p = q;
```

  - In this case, `p == q` will evaluate to `true`, and `p != q` will evaluate to `false`

```
int *p
double *q;
```

  - In this case, `q++;` increments value of `q` by `8`, and `p = p + 2;` increments value of `p` by `8`

Operations on Pointer Variables (continued)

- Pointer arithmetic can be very dangerous
  - The program can accidentally access the memory locations of other variables and change their content without warning
    - Some systems might terminate the program with an appropriate error message
- Always exercise extra care when doing pointer arithmetic

Dynamic Arrays

- Dynamic array: array created during the execution of a program
- Example:

```
int *p;
p = new int[10];
```

- The statements:

```
p[0] = 25;
p[1] = 35;
```

store `25` and `35` into the first and second array components, respectively

```
int list[5];
```

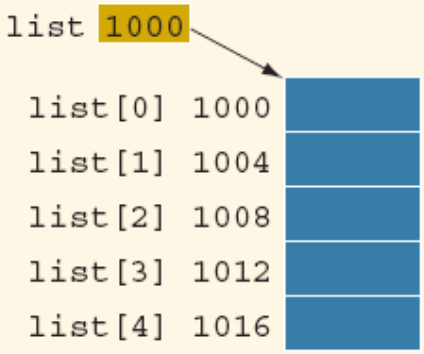


FIGURE 14-12 `list` and array `list`

```
list[0] = 25;
list[3] = 78;
```

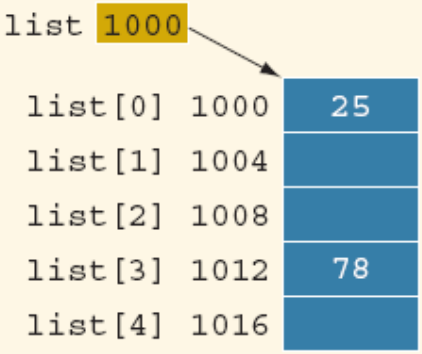


FIGURE 14-13 Array `list` after the execution of the statements `list[0] = 25;` and `list[3] = 78;`

Shallow versus Deep Copy of Pointers

```
int *first;
int *second;

first = new int[10];
```

- Assume some data is stored in the array:

FIGURE 14-17 Pointer `first` and its array

- If we execute: `second = first;` `//Line A`

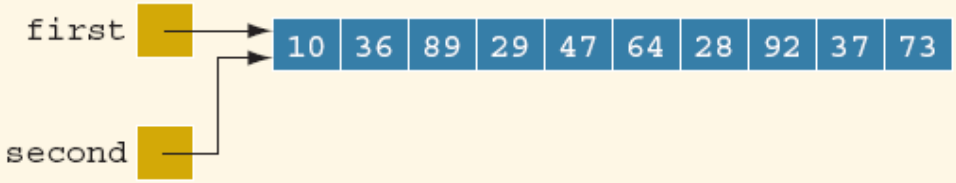


FIGURE 14-18 `first` and `second` after the statement `second = first;` executes

Shallow versus Deep Copy and Pointers (continued)

- Shallow copy: two or more pointers of the same type point to the same memory
  - They point to the same data

```
delete [] second;
```

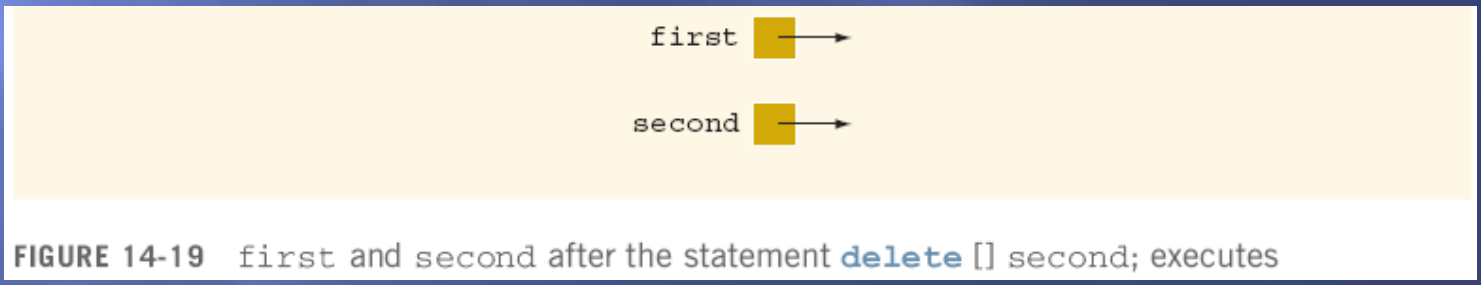


FIGURE 14-19 first and second after the statement delete [] second; executes

Shallow versus Deep Copy and Pointers (continued)

- Deep copy: two or more pointers have their own data

```
second = new int[10];

for (int j = 0; j < 10; j++)
    second[j] = first[j];
```

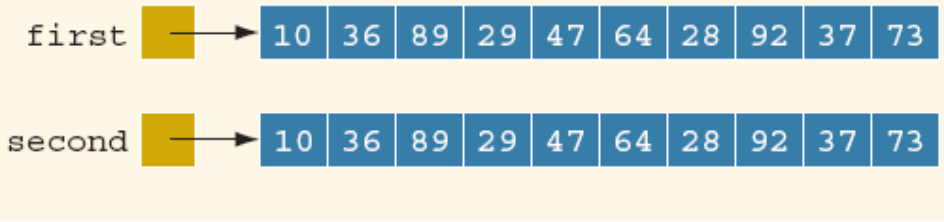


FIGURE 14-20 first and second both pointing to their own data