Lesson 7:
Pointers, Classes, and Lists

Pointer Data Type and Pointer Variables

- Pointer variable: content is a memory address
- 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 (cont'd.)

- In the statement:

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

  - Only `p` is a pointer variable
  - `q` is an `int` variable

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

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

Address of Operator (&)

- Address of operator (&):
  - A unary operator that returns the address of its operand
- Example:

```
int x;
int *p;
p = &x;
```

  - Assigns the address of x to p

Dereferencing Operator (*)

- Dereferencing operator (or indirection operator):
  - When used as a unary operator, * refers to object to which its operand points
- Example:

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

  - Prints the value stored in the memory location pointed to by `p`

Classes, `structs`, and Pointer Variables

- You can declare pointers to other data types:

```
struct studentType
{
    char name[26];
    double gpa;
    int sID;
    char grade;
} ;

studentType  student;
studentType *studentPtr;
```

  - `student` is an object of type `studentType`
  - `studentPtr` is a pointer variable of type `studentType`

Classes, `structs`, and Pointer Variables (cont'd.)

- To store address of `student` in `studentPtr`:

```
studentPtr = &student;
```

- To store `3.9` in component `gpa` of `student`:

```
(*studentPtr).gpa = 3.9;
```

  - ( ) used because dot operator has higher precedence than dereferencing operator
  - Alternative: use **member access operator arrow** (->)

Classes, `structs`, and Pointer Variables (cont'd.)

- Syntax to access a `class` (`struct`) member using the operator -> :

```
pointerVariableName->classMemberName
```

- Thus,

```
(*studentPtr).gpa = 3.9;
```

is equivalent to:

```
studentPtr->gpa = 3.9;
```

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 **null pointer**: the value 0
  - Or, use the `NULL` named constant
  - The number 0 is the only number that can be directly assigned to a pointer variable
- C++11 includes a `nullptr`

Dynamic Variables

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

Operator `new`

- `new` has two forms:

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

  - `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 allocated memory is uninitialized

Operator `new` (cont'd.)

- Example: `p = new int;`
  - Creates a variable during program execution somewhere in memory
  - Stores the address of the allocated memory in `p`
- To access allocated memory, use `*p`
- A dynamic variable cannot be accessed directly
  - Because it is unnamed

Operator `delete`

- **Memory leak**: previously allocated memory that cannot be reallocated
  - To avoid a memory leak, when a dynamic variable is no longer needed, destroy it to deallocate its memory
- `delete` operator: used to destroy dynamic variables
- Syntax:

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

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 (cont'd.)

- Pointer arithmetic can be very dangerous:
  - Program can accidentally access 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 program execution
- Example:

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

*p = 25;
p++; //to point to next array component
*p = 35;
```

← stores `25` into the first memory location

← stores `35` into the second memory location

Dynamic Arrays (cont'd.)

- Can use array notation to access these memory locations
- Example:

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

  - Stores `25` and `35` into the first and second array components, respectively
- An *array name* is a *constant pointer*

Functions and Pointers

- Pointer variable can be passed as a parameter either by value or by reference
- As a reference parameter in a function heading, use `&`:

```
void pointerParameters(int* &p, double *q)
{
    . . .
}
```

Pointers and Function Return Values

- A function can return a value of type pointer:

```
int* testExp(...)
{
    . . .
}
```

Dynamic Two-Dimensional Arrays

- You can create dynamic multidimensional arrays
- Examples:

```
int *board[4];
```

declares `board` to be an array of four pointers wherein each pointer is of type `int`

```
for (int row = 0; row < 4; row++)
    board[row] = new int[6];
```

creates the rows of `board`

```
int **board;
```

declares `board` to be a pointer to a pointer

Shallow Versus Deep Copy and Pointers

- Shallow copy: when two or more pointers of the same types point to the same memory
  - They point to the same data
  - Danger: deleting one deletes the data pointed to by all of them
- Deep copy: when the contents of the memory pointed to by a pointer are copied to the memory location of another pointer
  - Two copies of the data

Array-Based Lists

- List: collection of elements of the same type
- Typical list operations
  - Create the list
  - Determine if the list is empty
  - Determine if the list is full
  - Find the size of the list
  - Destroy or clear the list
  - Determine whether an item is the same as a given element

Array-Based Lists (cont'd)

- Typical list operations (cont'd.)
  - Insert an item into the list at the specified location
  - Remove an item from the list at the specified location
  - Replace an item at the specified location with another item
  - Retrieve an item from the list at the specified location
  - Search the list for a given item

Unordered Lists

- Can derive class unorderedArrayListType from the abstract class arrayListType
- Unordered list operations
  - insertAt: Insert item at location
  - insertEnd: Insert item at list end
  - replaceAt: Replace item at location with new item
  - seqSearch: find location of item
  - remove: remove item from list

Ordered Lists

- Can derive class orderedArrayListType from the abstract class arrayListType
- Unordered list operations
  - insertAt: Insert item at location
  - insertEnd: Insert item at list end
  - replaceAt: Replace item at location with new item
  - seqSearch: find location of item
  - remove: remove item from list
  - Insert: insert item into its ordered location

Address of Operator and Classes

- `&` operator can create aliases to an object
- Example:

```
int x;
int &y = x;
```

`x` and `y` refer to the same memory location

`y` is like a constant pointer variable

`y = 25;` sets the value of `y` (and of `x`) to `25`

`x = 2 * x + 30;` updates value of `x` and `y`

Address of Operator and Classes (cont'd.)

- Address of operator can also be used to return the address of a private member variable of a class
  - However, if you are not careful, this operation can result in serious errors in the program