Chapter 12 - Pointers, Classes, Virtual Functions, and Abstract Classes

Spring 2022

Objectives (1 of 3)

- In this chapter, you will:
  - Learn about the pointer data type and pointer variables
  - Explore how to declare and manipulate pointer variables
  - Learn about the address of the operator and the dereferencing operator
  - Learn how pointers work with classes and structs
  - Discover dynamic variables

Objectives (2 of 3)

- Explore how to use the new and delete operators to manipulate dynamic variables
- Learn about pointer arithmetic
- Learn how to work with dynamic arrays
- Become familiar with the limitations of range-based for loops with dynamic arrays
- Explore how pointers work with functions as parameters and functions as return values

Objectives (3 of 3)

- Become familiar with the shallow and deep copies of data
- Discover the peculiarities of classes with pointer member variables
- Learn about virtual functions
- Become aware of abstract classes
- Examine the relationship between the address of operator and classes

Pointer Data Type and Pointer Variables

- A **pointer variable** is a variable whose content is a memory address
- No name is associated with the pointer data type in C++

Declaring Pointer Variables (1 of 2)

- The general syntax to declare a pointer variable is:

```
dataType* identifier;
```

- The statements below each declare a pointer:
  - `int* p;`
  - `char* ch;`

- These statements are equivalent:
  - `int* p;`
  - `int *p;`
  - `int * p;`

Declaring Pointer Variables (2 of 2)

- In the statement:

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

- Only `p` is a pointer variable
- `q` is an `int` variable
- To avoid confusion,
  - declare each variable separately

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

  - otherwise, 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 (1 of 3)

- You can declare pointers to other data types, such as a struct:

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

studentType  student;
studentType* studentPtr;
```

- Read declaration from right to left:
  - `student` is an object of type `studentType`
  - `studentPtr` is a pointer variable of type `studentType`

Classes, structs, and Pointer Variables (2 of 3)

- To store address of student in studentPtr:

```
studentPtr = &student;
```

- To store 3.9 in component gpa of student:

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

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

Classes, structs, and Pointer Variables (3 of 3)

- 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 `nullptr`, the pointer literal included in modern C++

```
int* p = nullptr;
```

- ~~use the `NULL` named constant~~

  - Deprecated. Do not use `NULL` in modern C++.
- ~~The number 0 is the only number that can be directly assigned to a pointer variable~~
  - Deprecated. Do not assign the integer `0` to a pointer variable in modern C++

Dynamic Variables

- **Dynamic variables** are 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 (1 of 2)

- 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 (2 of 2)

- 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
```

- After memory has been deleted, pointer variables should be assigned `nullptr`

```
pointerVariable = nullptr;
```

Operations on Pointer Variables (1 of 2)

- 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 (2 of 2)

- 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 (1 of 2)

- **Dynamic array:** array created during program execution
- Example:

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

*p = 25;  // stores 25 in the first memory location
++p;      // to point to next array component
*p = 35;  // stores 35 into the second memory location
...
delete [] list;
list = p = nullptr;
```

Dynamic Arrays (2 of 2)

- 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 *pointer constant*.

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) {
    ...
}
```

  - p is a reference to a pointer to an int
  - q is a copy of a pointer to a double

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:

```
// board is an array of four pointers to int
int* board[4];
for (int row = 0; row < 4; ++row) {
    board[row] = new int[6];  // create board rows
}

// board is a pointer to a pointer to an int;
int** board = nullptr;
```

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

Classes and Pointers: Some Peculiarities (1 of 2)

- Example class:

```
class ptrMemberVarType {
public:
    ...
private:
    int  x;
    int  lenP;
```

```
    int* p = nullptr;
};
```

- Example program statements:

```
ptrMemberVarType objectOne;
ptrMemberVarType objectTwo;
```

Classes and Pointers: Some Peculiarities (2 of 2)

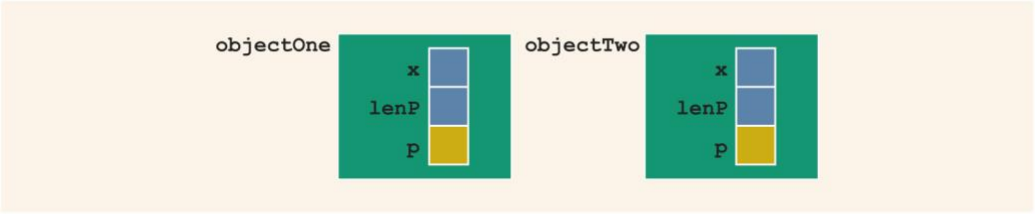

FIGURE 12-13 Objects `objectOne` and `objectTwo`

Destructor

- If `objectOne` goes out of scope, its member variables are destroyed
  - Memory space of a dynamic array stays marked as allocated, even though it cannot be accessed
- Solution: in destructor, ensure that when `objectOne` goes out of scope, its array memory is deallocated:

```
ptrMemberVarType::~ptrMemberVarType() {
    delete [] p;
}
```

Assignment Operator

- After a shallow copy: if objectTwo.p deallocates memory space to which it points, objectOne.p becomes invalid

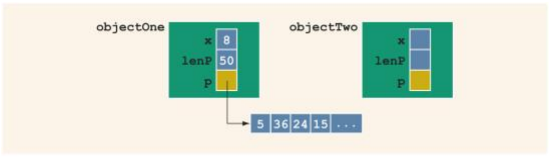

FIGURE 12-15 Objects `objectOne` and `objectTwo`

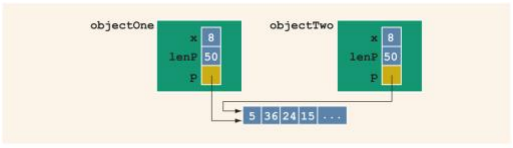

FIGURE 12-16 Objects `objectOne` and `objectTwo` after the statement `objectTwo = objectOne;` executes

- Solution: extend definition of the assignment operator to avoid shallow copying of data

Copy Constructor (1 of 3)

- Default member-wise initialization:
  - Initializing a class object by using the value of an existing object of the same type
- Example:

```
ptrMemberVarType objectThree(objectOne);
```

- **Copy constructor**: provided by the compiler
  - Performs this initialization
  - Leads to a shallow copying of the data if class has pointer member variables

Copy Constructor (2 of 3)

- Similar problem occurs when passing objects by value
- Copy constructor automatically executes in three situations:
  - When an object is declared and initialized by using the value of another object
  - When an object is passed by value as a parameter
  - When the return value of a function is an object

Copy Constructor (3 of 3)

- Solution: override the copy constructor

```
className(const className& otherObject);
```

- **Rule of Three**: For classes with pointer member variables, three things are normally done:
  - Include the destructor in the class
  - Overload the assignment operator for the class
  - Include the copy constructor

Inheritance, Pointers, and Virtual Functions (1 of 3)

- Can pass an object of a derived class to a formal parameter of the base class type
- **Compile-time binding**: the necessary code to call specific function is generated by compiler
  - Also known as **static binding** or **early binding**
- **Virtual function**: binding occurs at program execution time, not at compile time
  - Declared with reserved word `virtual`

Inheritance, Pointers, and Virtual Functions (2 of 3)

- **Run-time binding**:
  - Compiler does not generate code to call a specific function: it generates information to enable run-time system to generate specific code for the function call
  - Also known as **late binding** or **dynamic binding**
- Note: cannot pass an object of base class type to a formal parameter of the derived class type

Inheritance, Pointers, and Virtual Functions (3 of 3)

- Values of a derived class object can be copied into a base class object
- **Slicing problem**: if derived class has more data members than base class, some data could be lost
- Solution: use pointers for both base and derived class objects

Classes and Virtual Destructors (1 of 2)

- Classes with pointer member variables should have the destructor
  - Destructor should deallocate storage for dynamic objects
- If a derived class object is passed to a formal parameter of the base class type, destructor of the base class executes
  - Regardless of whether object is passed by reference or by value
- Solution: use a **virtual destructor** (base class)

```
virtual pointerMemberVarType::~pointerMemberVarType();
```

Classes and Virtual Destructors (2 of 2)

- **Virtual destructor** of a base class automatically makes the destructor of a derived class virtual
  - After executing the destructor of the derived class, the destructor of the base class executes
- If a base class contains virtual functions, make the destructor of the base class virtual

Abstract Classes and Pure Virtual Functions (1 of 2)

- New classes can be derived through inheritance without designing them from scratch
- Derived classes:
  - Inherit existing members of base class
  - Can add their own members
  - Can redefine or override public and protected member functions
- Base class can contain functions that you would want each derived class to implement

- However, base class may contain functions that may not have meaningful definitions in the base class

Abstract Classes and Pure Virtual Functions (2 of 2)

- **Pure virtual functions** do not have definitions (bodies have no code)
- Example:

```
virtual void draw() = 0;
```

- An **abstract class** is a class with one or more virtual functions
  - It can contain instance variables, constructors, and functions that are not pure virtual
  - It must provide the definitions of the constructor and functions that are not pure virtual

Address of Operator and Classes (1 of 2)

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

```
// x and y refer to the same memory location
// (y is like a constant pointer variable)
int x;
int& y = x;

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 (2 of 2)

- 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

Quick Review (1 of 2)

- Pointer variables contain the addresses of other variables as their values
  - Declare a pointer variable with an asterisk, *, between the data type and the variable
  - Address of operator (&) returns the address of its operand
  - Unary operator * is the dereferencing operator
  - Member access operator (->) accesses the object component pointed to by a pointer

Quick Review (2 of 2)

- Dynamic variable: created during execution
  - Created using `new`

  - Deallocated using `delete`
- Shallow copy: two or more pointers of the same type point to the same memory
- Deep copy: two or more pointers of the same type have their own copies of the data
- Binding of virtual functions occurs at execution time (dynamic or run-time binding)

Questions?