OOP244

Created	@May 9, 2023 7:55 AM
Class	OOP244
Materials	
Class Timings	8.00am Tue & Thu
Status	In progress

Welcome

- C++ is a Strongly typed Object Oriented language (every entered character needs to have its own data type).

Namespaces

- In C++, namespaces are used to group related identifiers (such as classes, functions, and variables) together and to avoid naming conflicts between different parts of a program.
- When you declare a namespace, all the identifiers declared within that namespace become part of that namespace. You can then access these identifiers using the scope resolution operator (::) followed by the name of the namespace and the name of the identifier. This helps to organize and clarify your code, especially when you are working with multiple libraries or modules that might define the same identifier names.
- For example, suppose you have two libraries that both define a function called "calculate". If you do not use namespaces, you will have a naming conflict when you try to use both libraries in the same program. However, if you put each library's

"calculate" function in a separate namespace (e.g., "lib1::calculate" and "lib2::calculate"), you can use both functions in the same program without any naming conflicts.

- Here's an example of how namespaces can be used in C++:

```cpp
#include <iostream>

namespace lib1 {
    void calculate() {
        std::cout << "Calculating from lib1..." << std::endl;
    }
}

namespace lib2 {
    void calculate() {
        std::cout << "Calculating from lib2..." << std::endl;
    }
}

int main() {
    lib1::calculate();
    lib2::calculate();
    return 0;
}
```

The output for the above code will be:

```
Calculating from lib1...
Calculating from lib2...
```

- The `using` directive is used to bring identifiers from a namespace into the current scope, which allows you to use those identifiers without having to qualify them with the namespace name.
- For example, let's say you have a namespace `my_namespace` that contains a function called `my_function`. Normally, you would need to use the scope resolution operator to call `my_function`:

```
my_namespace::my_function();
```

- However, if you add a `using` directive at the beginning of your code, like this:

```
using namespace my_namespace;
```

- You can then call `my_function` without having to qualify it with the namespace name:

```
my_function();
```

- It's important to note that using `using` directives can lead to naming conflicts if two namespaces contain identifiers with the same name. In general, it's recommended to avoid using `using` directives at the global scope, and instead use them only within functions or smaller scopes where the namespace is well-defined and naming conflicts are unlikely to occur.
- Here's an example of how `using` directives can be used in C++:

```cpp
#include <iostream>

namespace my_namespace {
    void my_function() {
        std::cout << "Hello, world!" << std::endl;
    }
}

int main() {
    using namespace my_namespace;
    my_function();
    return 0;
}
```

The output for the above code will be:

```
Hello, world!
```

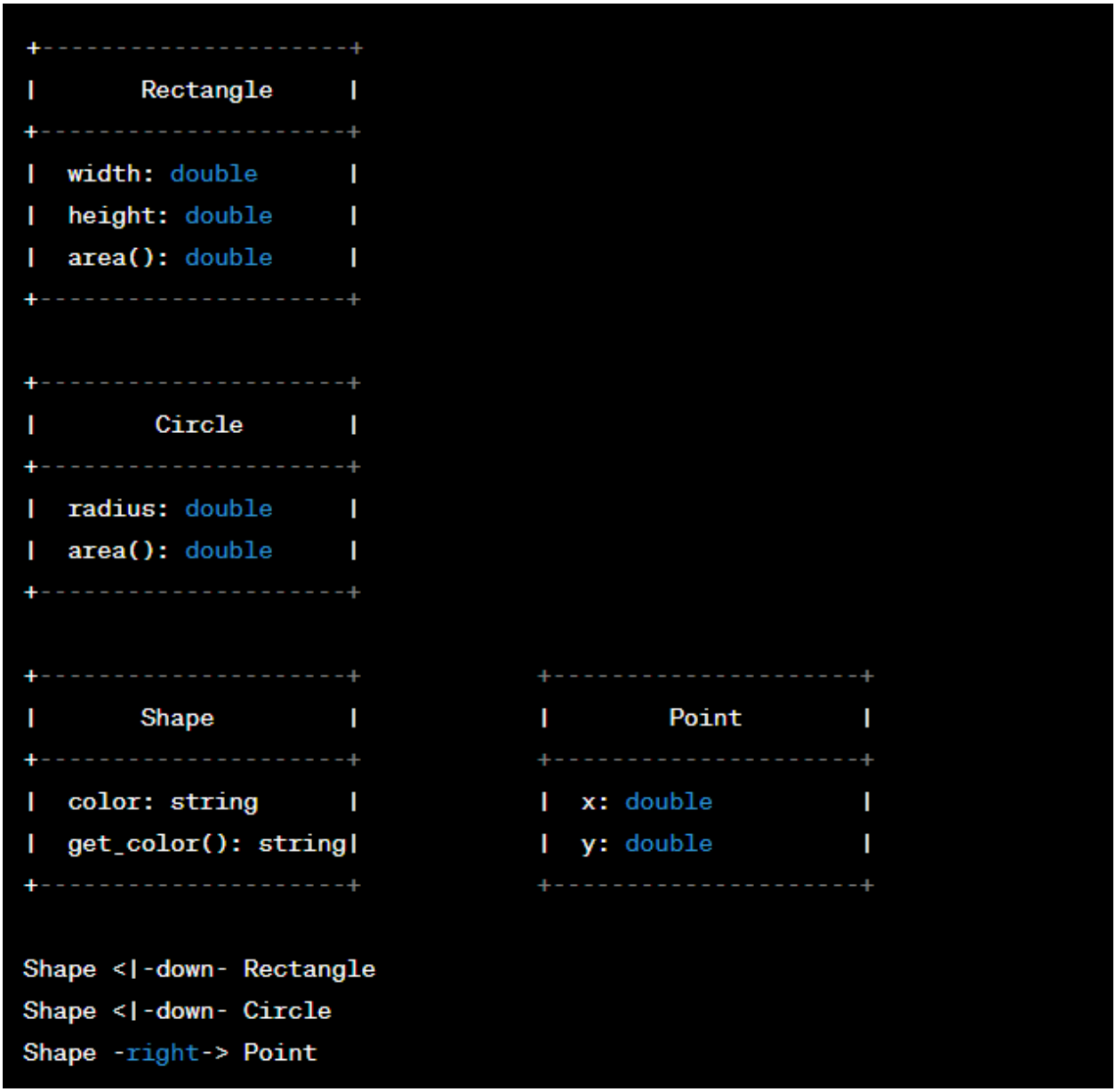
- Note that you can even use the printf() function in C++ by including the <cstdio> library in your code.
- For the normal C++ coding, you need to use the <iostream> library.

UML (Unified Modelling Language)

- UML stands for Unified Modeling Language, and it is a standardized visual modeling language used in software engineering to describe, design, and analyze software systems.
- UML provides a set of graphical notations for modeling different aspects of software systems, including its Structure, behaviour and interaction.

UML : Class Diagram

- A UML class diagram is a type of UML diagram that shows the static structure of a software system by modeling its classes, attributes, methods, and relationships.
- In a class diagram, classes are represented as boxes that contain the name of the class, its attributes (i.e., data members), and its methods (i.e., member functions). The relationships between classes are represented as lines that connect the classes and show how they are related to each other.



- In this example, we have four classes: Rectangle, Circle, Shape, and Point. Rectangle and Circle are subclasses of Shape, and Point is associated with Shape. Each class has its own attributes and methods, and the relationships between the classes are shown with arrows and labels.
- Note that the UML turns out to be not useful in your corporate world.

Inheritance

- Inheritance is a fundamental concept in object-oriented programming that allows you to create new classes based on existing classes. Inheritance enables a new class, called a subclass, to inherit the properties and behavior of an existing class, called a superclass. This means that the subclass automatically has all the variables, methods, and other characteristics of the superclass, in addition to any new variables or methods that the subclass defines.
- Here's a simple example of inheritance in C++:

```
class Animal {
public:
    void eat() {
        cout << "Animal is eating." << endl;
    }
    void sleep() {
        cout << "Animal is sleeping." << endl;
    }
};

class Dog : public Animal {
public:
    void bark() {
        cout << "Dog is barking." << endl;
    }
};

int main() {
    Dog myDog;
    myDog.eat();    // Output: Animal is eating.
    myDog.sleep();  // Output: Animal is sleeping.
    myDog.bark();   // Output: Dog is barking.
    return 0;
}
```

- In this example, we have two classes: Animal and Dog. Dog is a subclass of Animal, which means that it inherits all the properties and behavior of the Animal class. The Dog class adds a new method called bark().
- In the main() function, we create an object of the Dog class and call its methods. Because Dog inherits from Animal, the Dog object can also call the Animal methods eat() and sleep(). When we run the program, it outputs:

```
Animal is eating.
Animal is sleeping.
Dog is barking.
```

- This example illustrates how inheritance allows you to create new classes that reuse code and behavior from existing classes, while also adding new functionality. Inheritance is a powerful tool for creating complex software systems that are easy to maintain and extend over time.

C++ Hybrid Code

The object-oriented C++ source code for displaying our welcome phrase is

```
// A Language for Complex Applications
// welcome.cpp
//
// To compile on linux:  g++ welcome.cpp
// To run compiled code: a.out
//
// To compile on windows:   cl welcome.cpp
//  To run compiled code: welcome
//

#include <iostream>
using namespace std;

int main ( ) {
    cout << "Welcome to Object-Oriented" << endl;
}
```

The object-oriented syntax consists of:

1. The directive inserts the **<iostream>** header file into the source code.
   The **<iostream>** library provides access to the standard input and output objects.

   ```
   #include <iostream>
   ```

   ---

2. The object represents the standard output device.

```
cout
```

3. The operator inserts whatever is on its right side into whatever is on its left side.

```
<<
```

4. The manipulator represents an end of line character along with a flushing of the output buffer.

```
endl
```

Note the absence of a formatting string. The **cout** object handles the output formatting itself.

That is, the complete statement

```
cout << "Welcome to Object-Oriented" << endl;
```

inserts into the standard output stream the string **"Welcome to Object-Oriented"** followed by a newline character and a flushing of the output buffer.

SUMMARY

- object-oriented languages are designed for solving large, complex problems
- object-oriented programming focuses on the objects in a problem domain
- C++ is a hybrid language that can focus on activities as well as objects
- C++ provides improved type safety relative to C
- **cout** is the library object that represents the standard output device
- **cin** is the library object that represents the standard input device
- **<<** is the operator that inserts data into the object on its left-side operand

- **>>** is the operator that extracts data from the object on its left-side operand

Polymorphism

- Polymorphism is a core concept in object-oriented programming (OOP) that allows objects of different classes to be treated as objects of a common base class. It enables the same code to be used with objects of different types, providing flexibility and extensibility in software design.
- There are two main types of polymorphism:
    1. Compile-time Polymorphism (Static Polymorphism):
    - Achieved through function overloading and operator overloading.
    - The appropriate function or operator implementation is determined at compile-time based on the arguments' types.
    - Examples: Overloading functions with different parameter types, using operators like "+" for different data types.
    2. Runtime Polymorphism (Dynamic Polymorphism):
    - Achieved through inheritance and virtual functions.
    - The appropriate function implementation is determined at runtime based on the actual object's type.
    - Examples: Base class pointers or references pointing to derived class objects, virtual functions.
- Key concepts related to runtime polymorphism are:
    - Base Class: The common parent class that defines a set of methods or behaviors. It may have one or more derived classes.
    - Derived Class: A class that inherits properties and methods from the base class and can extend or override them.
    - Virtual Function: A function declared in the base class that can be overridden by a derived class. The function to be called is determined at runtime based on the actual object type.

    - Function Overriding: The process of providing a different implementation of a virtual function in a derived class.
    - Late Binding: The process of determining the appropriate function implementation at runtime based on the object's actual type.
    - Pure Virtual Function: A virtual function declared in the base class that has no implementation and must be overridden by any derived class before it can be instantiated. The base class becomes an abstract class.
    - Abstract Class: A class that contains one or more pure virtual functions and cannot be instantiated. It serves as a base class for other classes to derive from.
- Polymorphism allows for code reusability, modularity, and flexibility in object-oriented systems. By treating objects of different classes as objects of a common base class, it enables the creation of more generic and flexible code that can work with various object types without the need for extensive conditional logic or type checking.
- Example:

```cpp
#include <iostream>

// Base class
class Shape {
public:
    virtual void draw() {
        std::cout << "Drawing a shape." << std::endl;
    }
};

// Derived class 1
class Circle : public Shape {
public:
    void draw() {
        std::cout << "Drawing a circle." << std::endl;
    }
};

// Derived class 2
class Rectangle : public Shape {
public:
    void draw() {
        std::cout << "Drawing a rectangle." << std::endl;
    }
};

int main() {
    Shape* shape1 = new Circle();
    Shape* shape2 = new Rectangle();

    shape1->draw();  // Calls the draw() function of Circle
    shape2->draw();  // Calls the draw() function of Rectangle

    delete shape1;
    delete shape2;

    return 0;
}
```

In this example, we have a base class called `Shape` that has a virtual function called `draw()`. Two derived classes, `Circle` and `Rectangle`, inherit from the base class and

override the `draw()` function with their specific implementations.

In the `main()` function, we create pointers of type `Shape` pointing to objects of `Circle` and `Rectangle` classes. When we call the `draw()` function using these pointers, the appropriate version of the function is invoked based on the actual object type at runtime. This is achieved through dynamic dispatch and late binding.

- Output:

```
Drawing a circle.
Drawing a rectangle.
```

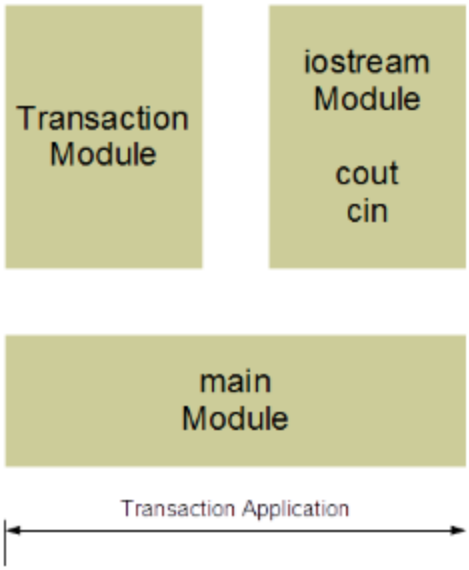
As you can see, even though we are calling the `draw()` function using pointers of the base class type, the correct `draw()` function implementation is called based on the actual object type. This is the essence of runtime polymorphism, allowing different objects to exhibit different behaviors through a common interface.

SUMMARY

- An object is a chunk of information with a crisp conceptual boundary and a well-defined structure.
- Objects are abstractions of the most important chunks of information from a problem domain. They distinguish the different feature sets in the problem domain.
- A class describes the structure common to a set of similar objects. Each object in the set is a single instance of its class.
- Encapsulation hides the implementation details within a class - the internal data and internal logic are invisible to client applications that use objects of that class.
- We can upgrade the structure of a well-encapsulated class without altering any client code.
- The cornerstones of object-oriented programming are encapsulation, inheritance and polymorphism.

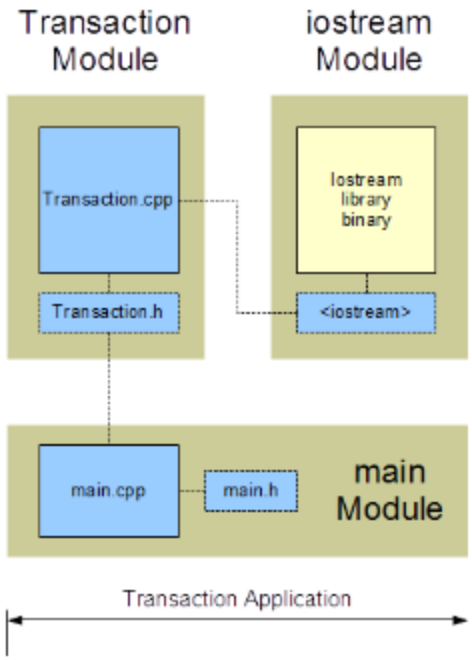
Modules

- A well-designed module is a highly cohesive unit that couples loosely to other modules. The module addresses one aspect of the programming solution and hides as much detail as practically possible. A compiler translates the module's source code independently of the source code for other modules into its own unit of binary code.
- Consider the schematic of the Transaction application shown below.
  The **main** module accesses the **Transaction** module.
  The **Transaction** module accesses the **iostream** module.
  The **Transaction** module defines the transaction functions used by the application. The **iostream** module defines the **cout** and **cin** objects used by the application.



- To translate the source code of any module the compiler only needs certain external information. This information includes the names used within the module but defined outside the module. To enable this in C++, we store the source code for each module in two separate files:
  - a header file - defines the class and declares the function prototypes

    - an implementation file - defines the functions and contains all of the logic
  - The file extension **.h** (or **.hpp**) identifies the header file. The file extension **.cpp** identifies the implementation file.
  - Note, however, that the names of the header files for the standard C++ libraries do not include a file extension (consider for example, the **<iostream>** header file for the **cout** and **cin** objects).
- An implementation file can include several header files but DOES NOT include any other implementation file. Note the absence of any direct connections between the implementation files.

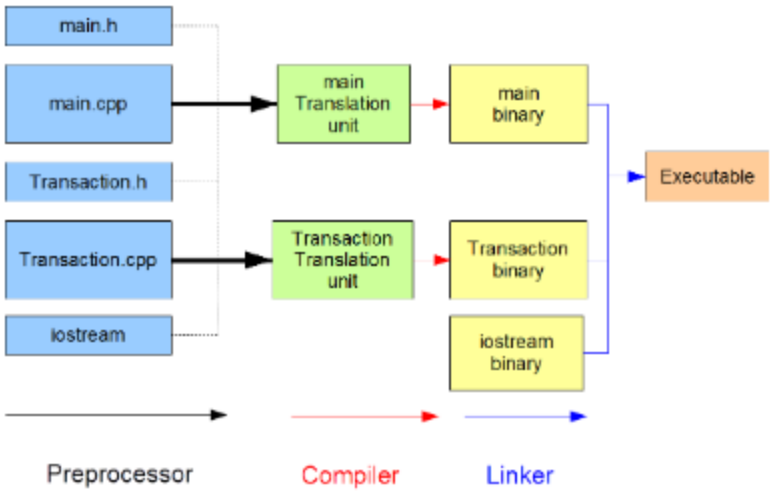


- We compile each implementation (***.cpp**) file separately and only once. We do not compile header (***.h**) files.

Stages Of Compilation

- Comprehensive compilation consists of three independent but sequential stages (as shown in the figure below):

1. Preprocessor - interprets all directives creating a single translation unit for the compiler - (inserts the contents of all **#include** header files), (substitutes all**#define** macros)
2. Compiler - compiles each translation unit separately and creates a corresponding binary version
3. Linker - assembles the various binary units along with the system binaries to create one complete executable binary



- To compile an application that includes a C++11 feature on a legacy Linux installation, we may need to specify the standard option. For example, to access C++11 features on the GCC 4.6 installation on our matrix cluster, we write

```
g++ -o accounting -std=c++0x main.cpp Transaction.cpp
```

Programming Errors

- Programming errors that require debugging skills are of two kinds:
  - syntactic

    - semantic
- **Syntactic Errors**
    - Syntactic errors are errors that break the rules of the programming language.
    - DEBUGGING TECHNIQUES:
        - IDE intellisense
        - compiler error messages (compiler output)
        - comparing error messages from different compilers - some are more cryptic than others
        - reading code statements (walkthroughs)
- **Semantic Errors**
    - Semantic errors are errors that fail to implement the intent and meaning of the program designer.
    - DEBUGGING TECHNIQUES:
        - vocalization - use your sense of hearing to identify the error (compound conditions)
        - intermediate output - **cout** statements at critical stages
        - walkthrough tables
        - interactive debugging using
            - Visual Studio IDE - integrated debugger for Windows OSs
            - Eclipse IDE - integrated debugger for Linux OSs
            - **gdb** - GNU debugger for **gcc**

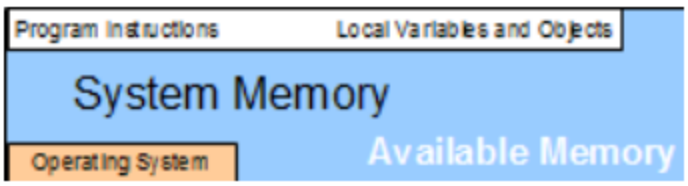
SUMMARY

- a module consists of a header file and an implementation file
- a module's header file declares the names that are exposed to client modules

- a module's implementation file defines the module's logic
- a module's implementation file needs the header files of those modules that define classes or functions used in the implementation file
- the three stages of creating an executable are preprocessing, compiling, and linking
- it is good practice to write the suite of unit tests for each module of an application before coding the module's implementation

Static and Dynamic Memory

- The memory accessible by a C++ program throughout its execution consists of static and dynamic components. After the user starts an application, the operating system loads its executable into RAM and transfers control to that executable's entry point (the **main()** function). The loaded executable only includes the memory allocated at compile time. During execution, the application may request more memory from the operating system. The system satisfies such requests by allocating more memory in RAM. After the application terminates and returns control to the operating system, the system recovers all of the memory that the application has used.
- **Static Memory:**
  - The memory that the operating system allocates for the application at load time is called *static memory*. Static memory includes the memory allocated for program instructions and program data. The compiler determines the amount of static memory that each translation unit requires. The linker determines the amount of static memory that the entire application requires.



- The application's variables and objects share static memory amongst themselves. When a variable or object goes out of scope its memory becomes available for newly defined variables or objects. The lifetime of each local variable and object concludes at the closing brace of the code block within which it has been defined:

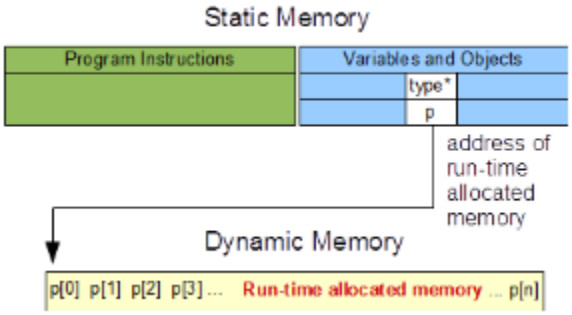
```
// lifetime of a local variable or object

for (int i = 0; i < 10; i++) {
    double x = 0;       // lifetime of x starts here
    // ...
}                       // lifetime of x ends here

for (int i = 0; i < 10; i++) {
    double y = 4;       // lifetime of y starts here
    // ...
}                       // lifetime of y ends here
```

- Since the variable **x** goes out of scope before the variable **y** is declared, the two variables may occupy the same memory location. This system of sharing memory amongst local variables and objects ensures that each application minimizes its use of RAM.
- Static memory requirements are determined at compile-link time and do not change during execution. This memory is fast, fixed in its amount and allocated at load time.

- **Dynamic Memory**
  - The memory that an application obtains from the operating system during execution is called *dynamic memory*.
  - Dynamic memory is distinct from the static memory. While the operating system allocates static memory for an application at load time, the system reserves dynamic memory, allocates it and deallocates it at run-time.
  - To keep track of an application's dynamic memory, we store the address of each allocated region in a pointer variable. We allocate memory for this pointer itself in static memory. This pointer variable must remain in scope as

long as we need access to the data in the allocated region of dynamic memory.

- Consider allocating dynamic memory for an array of n elements. We store the array's address in a pointer, **p**, in static memory as illustrated below. We allocate memory for the elements of the array dynamically and store the data in those elements starting at address **p**.



- **Lifetime**
  - The lifetime of any dynamically allocated memory ends when the pointer holding its address goes out of scope. The application must explicitly deallocate the allocated region of dynamic memory within this scope. If the application neglects to deallocate the allocated region, that memory becomes inaccessible and irrecoverable until the application returns control to the operating system.
  - Unlike variables and objects that have been allocated in static memory, those in dynamic memory do not automatically go of out scope at the closing brace of the code block within which they were defined. We must manage their deallocation explicitly ourselves.
- **Dynamic Memory Allocation**
  - The keyword **new** followed by **[n]** allocates contiguous memory dynamically for an array of **n** elements and returns the address of the array's first element.
  - Dynamic allocation of arrays takes the form

```
pointer = newType[size];
```

where ***Type*** is the type of the array's elements.

- For example, to allocate dynamic memory for an array of **n Student**s, we write

```
int n;                          // the number of students
Student* student = nullptr; // the address of the dynamic array

cout << "How many students in this section? ";
cin >> n;

student = new Student[n];   // allocates dynamic memory
```

- The **nullptr** keyword identifies the address pointed to as the null address. This keyword is an implementation constant. Initialization to **nullptr** ensures that **student** is not pointing to any valid dereferencable address. The size of the array is a run-time variable and not an integer constant or constant expression. Note that the size of an array allocated in static memory must be an integer constant or constant expression.

- **Dynamic Memory Deallocation**
  - The keyword **delete** followed by **[]** and the address of a dynamically allocated region of memory deallocates the memory that the corresponding **new[]** operator had allocated.
  - Dynamic deallocation of arrays takes the form

```
delete []pointer;
```

where ***pointer*** holds the address of the dynamically allocated array.

- For example, to deallocate the memory allocated for the array of **n Student**s above, we write

```
delete [] student;
student = nullptr;  // optional
```

        - The **nullptr** assignment ensures that **student** now holds the null address. This optional assignment eliminates the possibility of deleting the original address a second time, which is a serious run-time error. Deleting the **nullptr** address has no effect.
        - Note that omitting the brackets in a deallocation expression deallocates the first element of the array, leaving the other elements inaccessible.
        - NOTE: Deallocation does not return dynamic memory to the operating system. The deallocated memory remains available for subsequent dynamic allocations. The operating system only reclaims all of the dynamically allocated memory once the application has returned control to the system.

- **MEMORY ISSUES**
    - Issues regarding dynamic memory allocation and deallocation include:
        1. memory leaks
        2. insufficient memory
    - **Memory Leak**
        - Memory leaks are one of the most important bugs in object-oriented programming. A memory leak occurs if an application loses the address of dynamically allocated memory before that memory has been deallocated. This may occur if
            - the pointer to dynamic memory goes out of scope before the application deallocates that memory
            - the pointer to dynamic memory changes its value before the application deallocates the memory starting at the address stored in that pointer
        - Memory leaks are difficult to find because they often do not halt execution immediately. We might only become aware of their existence indirectly through subsequently incorrect results or progressively slower execution.

- **Insufficient Memory**
  - On small platforms where memory is severely limited, a realistic possibility exists that the operating system might not be able to provide the amount of dynamic memory requested. If the operating system cannot dynamically allocate the requested memory, the application may throw an exception and stop executing. The topic of exception handling is beyond the scope of these notes. One method of trapping a failure to allocate memory is described in the chapter entitled The ISO/IEC Standard.

SUMMARY

- the memory available to an application at run-time consists of static memory and dynamic memory
- static memory lasts the lifetime of the application
- the linker determines the amount of static memory used by the application
- the operating system provides dynamic memory to an application at run-time upon request
- the keyword **new []** allocates a contiguous region of dynamic memory and returns its starting address
- we store the address of dynamically allocated memory in static memory
- **delete []** deallocates contiguous memory starting at the specified address
- allocated memory must be deallocated within the scope of the pointer that holds its address

Types, Overloading and References

1. **Types:**
   - The built-in types of the C++ language are called its *fundamental type*s. The C++ language, like C, admits struct types constructed from these fundamental types and possibly other struct types. The C++ language standard refers to

struct types as *compound type*s. (The C language refers to struct types as derived types.)

- **Fundamental Types**
  - The fundamental types of C++ include:
    - Integral Types (store data exactly in equivalent binary form and can be signed or unsigned)
      - **bool** - not available in C
      - **char**
      - **int - short, long, long long**
    - Floating Point Types (store data to a specified precision - can store very small and very large values)
      - **float**
      - **double - long double**
- **bool**
  - The **bool** type stores a logical value: **true** or **false**.
  - The **!** operator reverses that value: **!true** is **false** and **!false** is **true**.
  - **!** is self-inverting on **bool** types, but not self-inverting on other types.
- **bool to int**
  - Conversions from **bool** type to any integral type and vice versa require care. **true** promotes to an **int** of value 1, while **false** promotes to an **int** of value 0. Applying the **!** operator to an **int** value other than 0 produces a value of 0, while applying the **!** operator to an **int** value of 0 produces a value of 1. Note that the following code snippet displays 1 (not 4)

`int x = 4; cout << !!x;`	`1`

  - Both C and C++ treat the integer value **0** as false and any other value as true.
- **Compound Types**

- A *compound type* is a type composed of other types. A struct is a compound type. An object-oriented class is also a compound type. To identify a compound type we use the keywords **struct** or **class**. We cover the syntax for classes in the following chapter.
- For example,

```
// Modular Example
// Transaction.h

struct Transaction {
    int acct;      // account number
    char type;     // credit 'c' debit 'd'
    double amount; // transaction amount
};
```

- The C++ language requires the keyword identifying a compound type only in the declaration of that type.

- **auto Keyword**
  - The **auto** keyword was introduced in the C++11 standard. This keyword deduces the object's type directly from its initializer's type. We must provide the initializer in any **auto** declaration.
  - For example,

```
auto x = 4;   // x is an int that is initialized to 4
auto y = 3.5; // y is a double that is initialized to 3.5
```

- **One Definition Rule**
  - In the C++ language, a definition may only appear once within its scope. This is called the *one-definition rule*.
  - For example, we cannot define **Transaction** or **display()** more than once within the same code block or translation unit.
  - NOTE: Header files consist of declarations. When we include several header files in a single implementation file, multiple declarations may

occur. If some of the declarations are also definitions, this may result in multiple definitions within the same translation unit. Any translation unit must not break the one-definition rule. We need to design our header files to respect this rule.

- **Proper Header File Inclusion**
  - To avoid contaminating system header files, we include header files in the following order:
    - **#include < ... >** - system header files
    - **#include " ... "** - other system header files
    - **#include " ... "** - your own header files

    We insert namespace declarations and directives after all header file inclusions.
- **Shadowing**
  - An identifier declared with an inner scope can shadow an identifier declared with a broader scope, making the latter temporarily inaccessible. For example, in the following program the second declaration shadows the first declaration of **i**:

`// scope.cpp #include <iostream> using namespace std; int main() { int i = 6; cout << i << endl; for (int j = 0; j < 3; j++) { int i = j * j; cout << i << endl; } cout << i << endl; }`	`6 0 1 4 6`

- **Function Signature**
  - A function's *signature* identifies an overloaded function uniquely. Its signature consists of
    - the function identifier
    - the parameter types (ignoring **const** qualifiers or address of operators as described in references below)
    - the order of the parameter types

```
typeidentifier (typeidentifier [, ... ,typeidentifier] )
```

- The square brackets enclose optional information. The return type and the parameter identifiers are not part of a function's signature.
- C++ compilers preserve identifier uniqueness by renaming each overloaded function using a combination of its identifier, its parameter types and the order of its parameter types. We refer to this renaming as name *mangling*.

- Consider the following example of an overloaded function. To display data on the standard output device, we can define a **display()** function with different meanings:

```
// Overloaded Functions
// overload.cpp
#include <iostream>
using namespace std;

// prototypes
void display(int x);
void display(const int* x, int n);

int main() {
    auto x = 20;
    int a[] = {10, 20, 30, 40};
    display(x);
    display(a, 4);
}

// function definitions
//
void display(int x) {
    cout << x << endl;
}

void display(const int* x, int n) {
    for (int i = 0; i < n; i++)
        cout << x[i] << ' ';
    cout << endl;
}
```

- C++ compilers generate two one definition of **display()** for each set of parameters. The linker binds each function call to the appropriate definition based on the argument types in the function call.

- **Default Parameter Values**
  - We may include default values for some or all of a function's parameters in the first declaration of that function. The parameters with default values must be the rightmost parameters in the function signature.
  - Declarations with default parameter values take the following form:

```
type identifier(type[, ...],type = value);
```

  - The assignment operator followed by a value identifies the default value for each parameter.
  - Specifying default values for function parameters reduces the need for multiple function definitions if the function logic is identical in every respect except for the values received by the parameters.
  - For example,

```
// Default Parameter Values // default.cpp #include <iostream> using namespace std; void display(int, int = 5, int = 0); int main() { display(6, 7, 8); display(6); display(3, 4); } void display(int a, int b, int c) { cout << a << ", " << b << ", " << c << endl;  }
```

```
6, 7, 8 6, 5, 0 3, 4, 0
```

  - Each call to **display()** must include enough arguments to initialize the parameters that don't have default values. In this example, each call must include at least one argument. An argument passed to a parameter that has a default value overrides the default value.
- **References**
  - A *reference* is an alias for a variable or object. Object-oriented languages rely on referencing. The declaration of a function parameter that is received as a reference to the corresponding argument in the function call takes the form

```
type identifier ( type & identifier , ... )
```

  - The **&** identifies the parameter as an alias for, rather than a copy of, the corresponding argument. The identifier is the alias for the argument within the function definition. Any change to the value of a parameter received by reference changes the value of the corresponding argument in the function call.

- **ARRAY OF POINTERS**
  - Arrays of pointers are data structures like arrays of values. Arrays of pointers contain addresses rather than values. We refer to the object stored at a particular address by dereferencing that address. Arrays of pointers play an important role in implementing polymorphism in the C++ language.
  - An array of pointers provides an efficient mechanism for processing the set. With the objects' addresses collected in a contiguous array, we can refer to each object indirectly through the pointers in the array and process the data by iterating on its elements.
  - In preparation for a detailed study of polymorphic objects later in this course, consider the following preliminary example:

```
// Array of Pointers int main() { const int NO_STUDENTS = 3; Student john = {1234, 67.8, "john"}; Student jane = {1235, 89.5, "jane"}; Student dave = {1236, 78.4, "dave"}; Student* pStudent[NO_STUDENTS]; // array of pointers  pStudent[0] = &john; pStudent[1] = &jane; pStudent[2] = &dave; for (int i = 0; i < NO_STUDENTS; i++) {  cout << pStudent[i]->no << endl; cout << pStudent[i]->grade << endl; cout << pStudent[i]->name << endl; cout << endl; } }
```

```
1234 67.8 john
1235 89.5 jane
1236 78.4 dave
```

  - Here, while the objects are of the same type, the processing of their data is done indirectly through an array of pointers to that data.
- **Comparison Examples**
  - Consider a function that swaps the values stored in two different memory locations. The programs listed below compare pass-by-address and pass-by-reference solutions. The program on the left passes by address using pointers. The program on the right passes by reference:

```
// Swapping values by address // swap1.cpp #include <iostream> using namespace std; void swap ( char *a, char *b ); int main ( ) { char left; char right; cout << "left is "; cin >> left; cout << "right is "; cin >> right; swap(&left, &right); cout << "After swap:" "\nleft is " << left << "\nright is " << right << endl; } void swap ( char *a, char *b ) { char c; c = *a; *a = *b; *b = c; }
```

```
// Swapping values by reference // swap2.cpp #include <iostream> using namespace std; void swap ( char &a, char &b ); int main ( ) { char left; char right; cout << "left is "; cin >> left; cout << "right is "; cin >> right; swap(left, right); cout << "After swap:" "\nleft is " << left << "\nright is " << right << endl; } void swap ( char &a, char &b ) { char c; c = a; a = b; b = c; }
```

  - Clearly, reference syntax is simpler. To pass an object by reference, we attach the address of operator to the parameter type. This operator

instructs the compiler to pass by reference. The corresponding arguments in the function call and the object names within the function definition are not prefixed by the dereferencing operator required in passing by address.

- Technically, the compiler converts each reference to a pointer with an unmodifiable address.

SUMMARY

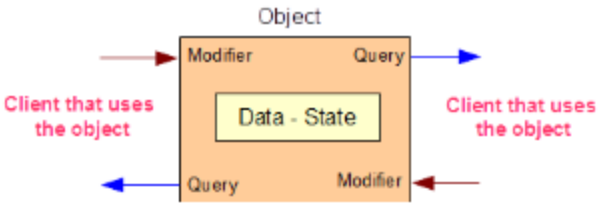
- a **bool** type can only hold a **true** value or a **false** value
- C++ requires the **struct** or **class** keyword only in the definition of the class itself
- a declaration associates an identifier with a type
- a definition attaches meaning to an identifier and is an executable statement
- a definition is a declaration, but a declaration is not necessarily a definition
- the scope of a declaration is that part of the program throughout which the declaration is visible
- we overload a function by changing its signature
- a function's signature consists of its identifier, its parameter types, and the order of its parameter types
- a C++ function prototype must include all of the parameter types and the return type
- the **&** operator on a parameter type instructs the compiler to pass by reference
- pass by reference syntax simplifies the pass by address syntax in most cases
- an array of pointers is a data structure that provides an efficient way for iterating through a set of objects based on their current type

Member Functions and Privacy

- The primary concept of object-oriented programming is class encapsulation. Encapsulation incorporates within a class the structure of data that its objects store and the logic that operates on that data. In other words, encapsulation creates a clean interface between the class and its clients while hiding the implementation details from its clients. The C++ language describes this logic in the form of

functions that are members of the class. The data members of a class hold the information about the state of its objects, while the member functions define the operations that query, modify and manage that state.

- **MEMBER FUNCTIONS**
  - The member functions of a class provide the communication links between client code and objects of the class. Client code calls the member functions to access an object's data and possibly to change that data.
  - We classify member functions into three mutually exclusive categories:
    - *queries* - also called accessor methods - report the state of the object
    - *modifiers* - also called mutator methods - change the state of the object
    - *special* - also called manager methods - create, assign and destroy an object



    - Every member function has direct access to the members of its class. Each member function receives information from the client code through its parameters and passes information to the client code through its return value and possibly its parameters.
  - **Adding a Member Function**
    - Consider a **Student** type with the following definition

```
const int NG = 20;

struct Student {
    int no;          // student number
    float grade[NG]; // grades
    int ng;          // number of grades filled
};
```

- **Function Declaration**
  - To declare a member function to a class, we insert its prototype into the class definition.
  - For example, to add **display()** as a member to our **Student** type, we write:

```
struct Student {
    int no;
    float grade[NG];
    int ng;

    void display() const; // member function
};
```

  - The **const** qualifier identifies the member function as a query. A query does not change the state of its object. That is, this query cannot change the value of **no** or any **grade**.
  - As a member function, **display()** has direct access to the data members (**no** and **grade**). There is no need to pass their values as arguments to the function.
- **Function Definition**
  - We define **display()** in the implementation file as follows:

```
void Student::display() const {

    cout << no << ": \n";
    for (int i = 0; i < ng; i++)
        cout << grade[i] << endl;

}
```

  - The definition consists of four elements:
    - the **Student::** prefix on the function name identifies it as a member of our **Student** type
    - the empty parameter list - this function does not receive any values from the client code or return any values through the parameter list to

the client code

- the **const** qualifier identifies this function as a query - this function cannot change any of the values of the object's data members
- the data members - the function accesses **no** and **grade** are defined outside the function's scope but within the class' scope, which encompasses the function's scope

- **Calling a Member Function**
  - Client code calls a member function in the same way that an instance of a **struct** refers to one of its data members. The call consists of the object's identifier, followed by the **.** operator and then followed by the member function's identifier.
  - For example, if **harry** is a **Student** object, we display its data by calling **display()** on **harry**:

```
Student harry = {975, 78.9f, 69.4f};

harry.display();  // <== client call to the member function

cout << endl;
```

  - The object part of the function call (the part before the member selection operator) identifies the data that the function accesses.
- **Scope of a Member Function**
  - The scope of a member function lies within the scope of its class. That is, a member function can access any other member within its class' scope.
- **Accessing Global Functions**
  - A member function can also access a function outside its class' scope. Consider the following global function definition:

```
void displayNo() {
    cout << "Number...\n";
}
```

- Note that this definition does not include any scope resolution identifier. This global function shares the same identifier with one of the member functions, but does not introduce any conflict, since the client code calls each function using different syntax.

```
displayNo();       // calls the global display function
harry.displayNo(); // calls the member function on harry
```

- To access the global function from within the member function we apply the scope resolution operator:

```
void Student::display() const {
    ::displayNo(); // calls the global function
    displayNo();   // calls the member function
    for (int i = 0; i < ng; i++)
        cout << grade[i] << endl;
}
```

- **Privacy**
  - Data privacy is central to encapsulation. Data members defined using the **struct** keyword are exposed to client code. Any client code can change the value of a data member. To limit accessibility to any member, the C++ language lets us hide that member within the class by identifying it as private.
  - Well-designed object-oriented solutions expose to client code only those members that are the class's communication links. In a good design, the client code should not require direct access to any data that describes an object's state or any member function that performs internally directed operations.
  - **Accessibility Levels**
    - **private** identifies all subsequent members listed in the class definition as inaccessible.
    - **public** identifies all subsequent members listed in the class definition as accessible.
    - For example, in order to

- hide the data members of each **Student** object
- expose the member function(s) of the **Student** type

we insert the accessibility keywords as follows

```
struct Student {
  private:
    int no;
    float grade[NG];
    int ng;
  public:
    void display() const;
};
```

- Note that the keyword **struct** identifies a class that is *public by default*.

- The keyword **class** identifies a class that is *private by default*.
  - We use the keyword **class** to simplify the definition of a **Student** type:

```
class Student {
    int no;
    float grade[NG];
    int ng;
  public:
    void display() const;
};
```

  - The **class** keyword is the common keyword in object-oriented programming, much more common than the **struct** keyword. (The C language does not support privacy and a derived type in C can only be a **struct**).
  - Any attempt by the client code to access a private member generates a complier error:

```
void foo(const Student& harry) {

    cout << harry.no;  // ERROR - this member is private!

}
```

- The function **foo()** can only access the data stored in **harry** indirectly through **public** member function **display()**.

```
void foo(const Student& harry) {

    harry.display(); // OK

}
```

- **Modifying Private Data**
  - If the data members of a class are private, client code cannot initialize their values directly. We use a separate member function for this specific task.
  - For example, to store data in **Student** objects, let us introduce a **public** modifier named **set()**:

```
const int NG = 20;

class Student {
    int no;
    float grade[NG];
    int ng;
  public:
    void set(int, const float*, int);
    void display() const;
};
```

  - **set()** receives a student number, the address of an unmodifiable array of grades and the number of grades in that array from the client code and stores this information in the data members of the **Student** object:

```
void Student::set(int sn, const float* g, int ng_) {
    ng = ng_ < NG ? ng_ : NG;
    no = sn; // store the Student number as received
    // store the grades as received within the available space
    for (int i = 0; i < ng; i++)
        grade[i] = g[i];
}
```

- INPUT and OUTPUT examples
  - **cin**
    - The general expression for extracting characters from the standard input stream takes the form

```
cin >>identifier
```

      where **>>** is the extraction operator and ***identifier*** is the name of the destination variable.

      - The **cin** object skips leading whitespace with numeric, string and character types (in the same way
        that **scanf("%d"...)**, **scanf("%lf"...)**, **scanf("%s"...)** and **scanf("%c"...)** skip whitespace in C).
  - The **istream** type supports the following member functions:
    - **ignore(...)** - ignores/discards character(s) from the input buffer
    - **get(...)** - extracts a character or a string from the input buffer
    - **getline(...)** - extracts a line of characters from the input buffer
  - **ignore**
    - The **ignore()** member function extracts characters from the input buffer and discards them. **ignore()** does not skip leading whitespace. Two versions of **ignore()** are available:

```
cin.ignore();
cin.ignore(2000, '\n');
```

The no-argument version discards a single character. The two-argument version removes and discards up to the specified number of characters or up to the

specified delimiting character, whichever occurs first and discards the delimiting character. The default delimiter is end-of-file (not end-of-line).

- **cout**
  - The general expression for inserting data into the standard output stream takes the form

    `cout << identifier`

    where **<<** is the insertion operator and ***identifier*** is the name of the variable or object that holds the data.

    endl inserts a newline character into the stream and flushes the stream's buffer.
  - The **ostream** type supports the following public member functions for formatting conversions:
    - **width(int)** - sets the field width to the integer received
    - **fill(char)** - sets the padding character to the character received
    - **setf(...)** - sets a formatting flag to the flag received
    - **unsetf(...)** - unsets a formatting flag for the flag received
    - **precision(int)** - sets the decimal precision to the integer received
- **width**
  - The **width(int)** member function specifies the minimum width of the next output field:

```
// Field Width
// width.cpp

#include <iostream>
using namespace std;

int main() {
    int attendance = 27;
    cout << "1234567890" << endl;
    cout.width(10);
    cout << attendance << endl;
```

```
    cout << attendance << endl;
}
```

- The output for the above code will be:

```
1234567890
        27
27
```

- NOTE: **width(int)** applies only to the next field. Note how the field width for the first display of **attendance** is 10, while the field width for the second display of **attendance** is just the minimum number of characters needed to display the value (2).

- **fill**
  - The **fill(char)** member function defines the padding character. The output object inserts this character into the stream wherever text occupies less space than the specified field width. The default fill character is **' '** (space). To pad a field with **'*'**'s, we add:

```
// Padding
// fill.cpp

#include <iostream>
using namespace std;

int main() {
    int attendance = 27;
    cout << "1234567890" << endl;
    cout.fill('*');
    cout.width(10);
    cout << attendance << endl;
}
```

  - The output for the above code will be

```
1234567890
********27
```

- The padding character remains unchanged, until we reset it.

- **setf, unsetf**
  - The **setf()** and **unsetf()** member functions control formatting and alignment. Their control flags include:

Control Flag	Result
ios::fixed	ddd.ddd
ios::scientific	d.ddddddEdd
ios::left	align left
ios::right	align right

  - The scope resolution (**ios::**) on these flags identifies them as part of the **ios** class.

- **setf, unsetf - Formatting**
  - The default format in C++ is *general format*, which outputs data in the simplest, most succinct way possible (1.34, 1.345E10, 1.345E-20). To output a fixed number of decimal places, we select *fixed format*. To specify fixed format, we pass the **ios::fixed** flag to **setf()**:

```
// Fixed Format
// fixed.cpp

#include <iostream>
using namespace std;

int main() {
    double pi = 3.141592653;
    cout << "1234567890" << endl;
    cout.width(10);
    cout.setf(ios::fixed);
    cout << pi << endl;
}
```

  - The output for the above code will be:

```
1234567890
  3.141593
```

- Format settings persist until we change them. To unset fixed format, we pass the **ios::fixed** flag to the **unsetf()** member function:

```
// Unset Fixed Format
// unsetf.cpp

#include <iostream>
using namespace std;

int main() {
    double pi = 3.141592653;
    cout << "1234567890" << endl;
    cout.width(10);
    cout.setf(ios::fixed);
    cout << pi << endl;
    cout.unsetf(ios::fixed);
    cout << pi << endl;
}
```

- Output:

```
1234567890
  3.141593
3.14159
```

- To specify scientific format, we pass the **ios::scientific** flag to the **setf()** member function:

```
// Scientific Format
// scientific.cpp

#include <iostream>
using namespace std;

int main() {
    double pi = 3.141592653;
    cout << "12345678901234" << endl;
    cout.width(14);
    cout.setf(ios::scientific);
    cout << pi << endl;
}
```

- Output:

```
12345678901234
  3.141593e+00
```

- To turn off scientific format, we pass the **ios::scientific** flag to the **unsetf()** member function.

- **setf, unsetf - Alignment**
  - The default alignment is right-justified.
  - To specify left-justification, we pass the **ios::left** flag to the **setf()** member function:

```
// Left Justified // left.cpp #include <iostream> using namespace std;
int main() { double pi = 3.141592653; cout << "1234567890" << endl;
cout.width(10); cout.fill('?'); cout.setf(ios::left); cout << pi << endl;
}
```

```
//O UTPUT
1234567890
3.14159???
```

  - To turn off left-justification, we pass the **ios::left** flag to the **unsetf()** member function:

```
cout.unsetf(ios::left);
```

- **precision**
  - The **precision()** member function sets the precision of subsequent floating-point fields. The default precision is **6** units. General, fixed, and scientific formats implement precision differently. General format counts the number of significant digits. Scientific and fixed formats count the number of digits following the decimal point.
  - For a precision of **2** under general format, we write

```
// Precision // precison.cpp #include <iostream> using namespace
std; int main() { double pi = 3.141592653; cout << "1234567890" <<
endl; cout.setf(ios::fixed); cout.width(10); cout.precision(2); cout
<< pi << endl; }
```

```
//OUTPUT
1234567890 3.14
```

  - The precision setting applies to the output of all subsequent floating-point values until we change it.

SUMMARY

- object-oriented classes may contain both data members and member functions
- the keyword **private** identifies subsequent members as inaccessible to any client
- the keyword **public** identifies subsequent members as accessible to any client
- data members hold the information about an object's state
- member functions describe the logic that an object performs on its data members
- a query reports the state of an object without changing its state
- a modifier changes the state of an object
- an empty state is the set of data values that identifies the absence of valid data in an object
- a field width setting only holds for the next field
- all settings other than a field width setting persist until changed
- precision has different meanings under general, scientific, and fixed formats