CSCI 123 Introduction to Programming Concepts in C++

Brad Rippe

Structures and ADTs
Overview

10.1 Structures

10.2 Classes

10.3 Abstract Data Types
10.1 Structures
What Is a Class?

- A class is a data type whose variables are objects
- Some pre-defined classes you have used are:
  - string
  - vector
  - ifstream
- You can define your own classes as well
Class Definitions

• A class definition includes
  – A description of the kinds of values the variable can hold
  – A description of the member functions

• We will start by defining structures as a first step toward defining classes
Structures

• A structure can be viewed as an object
  – Contains no member functions
    (The structures used here have no member functions)
  – Structures are the light version of classes

  – Contains multiple values of possibly different types
    • The multiple values are logically related as a single item
    • Example: A Student
      has the following values:
        a name
        an student id
The Student Definition

• The Student structure can be defined as
  ```c
  struct Student {
    char name[256];
    char sID[256];
  }
  ```
  Remember this semicolon!

• Keyword `struct` begins a structure definition
• Student is the structure tag or the structure’s type
• Member names are identifiers declared in the braces
Using the Structure

• Structure definition is generally placed outside any function definition
  – This makes the structure type available to all code that follows the structure definition
• To declare two variables of type Student:
  
  ```
  Student studentOne;
  Student studentTwo;
  ```

• `studentOne` and `studentTwo` contain distinct member variables `name`, and `sID`
The Structure Value

• The Structure Value
  – Consists of the values of the member variables

• The value of an object of type Student
  – Consists of the values of the member variables

  name
  sID
Specifying Member Variables

• Member variables are specific to the structure variable in which they are declared
  
  – Syntax to specify a member variable:
    `structureVariableName . memberVariableName`
  
  – Given the declaration:
    `Student studentOne;`
    `Student studentTwo;`
  
  • Use the dot operator to specify a member variable
    `studentOne.name`
    `studentOne.sID`
    `studentTwo.name`
    `studentTwo.sID`
Using Member Variables

• Member variables can be used just as any other variable of the same type

// Student One
cout << "Please type the name of the student.\n";
cin.getline(studentOne.name, 256);
cout << "Please type the student id of the student.\n";
cin.getline(studentOne.sID, 256);

// Student Two
cout << "Please type the name of the student.\n";
cin.getline(studentTwo.name, 256);
cout << "Please type the student id of the student.\n";
cin.getline(studentTwo.sID, 256);

Notice that studentOne and studentTwo are different variables!
Duplicate Names

• Member variable names duplicated between structure types are not a problem.

```
struct Student {
    char name[80];
    char sID[80];
};
Student aStudent;

struct Instructor {
    char name[80];
    double salary;
};
Instructor aInstructor;
```

• `aStudent.name` and `aInstructor.name` are different variables stored in different locations (memory addresses)
Structures as Arguments

• Structures can be arguments in function calls
  – The formal parameter can be call-by-value
  – The formal parameter can be call-by-reference

• Example:
  void getStudentInfo(Student& aStudent);
  – Uses the structure type Student we saw earlier as the type for a call-by-reference parameter

void getStudentInfo(Student aStudent);
  – Uses the structure type Student we saw earlier as the type for a call-by-value parameter
Structures as Return Types

- Structures can be the type of a value returned by a function

Example:

```c
Student getStudentInfo(char aName[], char aSID[]) {
    Student aStudent;
    strcpy(aStudent.name, aName);
    strcpy(aStudent.sID, aSID);
    return aStudent;
}
```
Using Function getStudentInfo

• getStudentInfo builds a complete structure value in aStudent, which is returned by the function

• We can use getStudentInfo to give a Student variable a value in this way:

```java
Student stuOne = getStudentInfo("John Doe", "22345");
```
Assignment and Structures

- The assignment operator can be used to assign values to structure types as you saw in the previous example.
- Using the `SimpleStruct` structure again:

```
SimpleStruct struct1;
SimpleStruct struct2;
struct1.simpleVar1 = 123;
struct1.simpleVar2 = 34;
struct2 = struct1;
```

- Assigns all member variables in `struct2` the corresponding values in `struct1`
Assignment and Structures

• The assignment operator can be used to assign values to structure types as you saw in the previous example.

Using the `SimpleStruct` structure again:

```cpp
Student studentOne;
Student studentTwo;

studentOne = getStudentInfo("John Doe", "22345");
studentTwo = studentOne;

outputStudentInfo(studentTwo);
```

- Assigns all member variables in `studentOne` the corresponding values in `studentTwo`
Hierarchical Structures

• Structures can contain member variables that are also structures

```c
struct Person {
    char name[256];
    char ssid[10];
};
```

```c
struct Student {
    Person aPerson;
    char sID[256];
};
```

• `struct Student` contains a `Person` structure
Using Student

• A variable of type Student is declared by
  Student student;

• To display the student’s name, first access the
  person member of student

```cpp
cout << student.aPerson...;
```

• But we want the name, so we now specify the
  name member of the Person member

```cpp
cout << student.aPerson.name;
```
Initializing Classes

• A structure can be initialized when declared
• Example:

```c
struct Person {
    char name[256];
    char ssID [10];
};
```

– Can be initialized in this way

```c
Person aPerson = { "Brad Rippe", "0000000000" }; 
```
10.2

Classes
Classes

- A class is a data type whose variables are objects
  - The definition of a class includes
    - Description of the kinds of values of the member variables
    - Description of the member functions
  - A class description is somewhat like a structure definition plus the member functions
  - We’ve used stream objects and string objects
Classes

- Have attributes (*member variables*) and behaviors (*member functions*)
- The objects (the instantiated class) can be used to simulate real world entities
- Promotes code reuse by allowing the programmer to specify a very general version of an object (Person) and make it more specific (Student or Instructor) by allowing the programmer to build on their code
Class Circle

• To create a new type named *Circle* as a class definition
  – Decide on the values to represent
  – This example’s value is the circle’s radius 10.0 or 25.0 using a double for the radius
    • Member variable radius
  – Decide on the member functions needed
    • setRadius
    • getRadius
    • getArea
class Circle {

public:
    double getRadius();
    void setRadius(double aRadius);
    double getArea();

private:
    double mRadius;
};
Defining a Member Function

- Member functions are declared in the class declaration
- Member function definitions identify the class in which the function is a member
- If the class is declared and defined in the same file as main, the class declaration should be above the main and the definitions (described here) should be below

```cpp
void Circle::setRadius(double aRadius) {
    mRadius = aRadius;
}

double Circle::getRadius() {
    return mRadius;
}

double Circle::getArea() {
    return mRadius*mRadius*PIE;
}
```
Member Function Definition

• Member function definition syntax:
  ReturnType ClassName::functionName(Parameters) {
    Function Body Statement(s);
  }

  – Example:
  double Circle::getRadius() {
    return mRadius;
  }
The ‘::’ Operator

• ‘::’ is the scope resolution operator

  – Tells the class a member function is a member of the class

  – double Circle::getRadius() indicates that function getRadius is a member of the Circle class

  – The class name that precedes ‘::’ is a type qualifier (what class the member function belongs to)
‘::’ and ‘.’

- ‘::’ used with classes to identify a member

```cpp
double Circle::getRadius() {
    return mRadius;
}
```

- ‘.’ used with variables to identify a member or call/invoke a member function

```cpp
Circle circle2;
circle2.setRadius(10);
```
Calling Member Functions

• Calling the Circle member function output is done in this way:

```cpp
cout << "Circle's radius is " << circle.getRadius() << endl;
cout << "Circle2's radius is " << circle2.getRadius() << endl;
```

— Note that `circle` and `circle2` have their own copies of the `radius` variable for use by the `getArea` function (Setting the value of circle’s radius has no affect on circle2’s radius)
Instantiations

- Each instance has its own radius
- In other words, each has its own member variable(s)
- Each instance has its own member functions (However, share the same implementation of member functions)
Encapsulation

- Encapsulation is
  - Combining a number of items, such as variables and functions, into a single package such as an object of a class

- UML Diagram
  - - denotes a private member
  - + denotes a public member

<table>
<thead>
<tr>
<th>Circle</th>
</tr>
</thead>
<tbody>
<tr>
<td>-radius : double</td>
</tr>
<tr>
<td>+setRadius() : void</td>
</tr>
<tr>
<td>+getRadius() : double</td>
</tr>
<tr>
<td>+getArea() : double</td>
</tr>
</tbody>
</table>
Problems With Class Definitions

- The *accessors* and *mutators* in the Circle class definition help to limit the problems with applications that use the class.
- We try to limit changes to the class interface that would require changes to the overall program.
- Modifications to the class in the Circle version 2 (*online examples*) don’t require any changes to the main() which uses the Circle class.
Ideal Class Definitions

• An ideal class definition of Circle could be changed without requiring changes to the program that uses Circle
Public Or Private?

• C++ helps us restrict the program from directly referencing member variables
  – private members of a class can only be referenced within the definitions of member functions
    • If the program tries to access a private member, the compiler gives an error message
  – Private members can be variables or functions
Private Variables

- Private variables cannot be accessed directly by the program
  - Changing their values requires the use of public member functions of the class

- To set the private radius variable in a new Circle class use a member function:

```cpp
void Circle::setRadius(double aRadius) {
    mRadius = aRadius;
}
```
Public or Private Members

• The keyword private identifies the members of a class that can be accessed only by member functions of the class
  – Members that follow the keyword private are private members of the class
• The keyword public identifies the members of a class that can be accessed from outside the class
  – Members that follow the keyword public are public members of the class
Using Private Variables

- It is normal to make all member variables private
- Private variables require member functions to perform all changing and retrieving of values
  - Accessor functions allow you to obtain the values of member variables
    - Example: `getRadius()` in class Circle
  - Mutator functions allow you to change the values of member variables
    - Example: `setRadius()` in class Circle
General Class Definitions

• The syntax for a class definition is

```cpp
class ClassName {
    public:
        memberSpecification1
        memberSpecification2
        ...
        memberSpecification3
    private:
        memberSpecificationN+1
        memberSpecificationN+2
        ...
};
```
A Bike

• Bike (version 1)
  – Uses all public member variables
  – Notice my convention is to capitalize the first letter of the type name, “B” in Bike.

```cpp
class Bike {
public:
    char mName[80];
    int mSize;
    double mWheelDiameter;
};
```
A Bike

• Bike (version 1)
  – Simple
  – Suppose we don’t find the cString flexible enough and want to change the name variable to a String
  – This causes a problem for the main function and any users that are utilizing the Bike class
  – The main must be changed
Fix the Bike

• Bike (version 2)
  – Use accessors and mutators to access members
  – Restrict member variables to private access

```cpp
class Bike {
public:
    string getName();
    void setName(string aName);
    int getSize();
    void setSize(int aSize);
    double getWheelDiameter();
    void setWheelDiameter(double aDiameter);

private:
    string mName;
    int mSize;
    double mWheelDiameter;
};```
Fix the Bike

• Bike (version 2)
  – Use accessors and mutators to access members
  – Restrict member variables to private access
  – Name accessor and mutator with a string name variable

```cpp
string Bike::getName() {
    return mName;
}

void Bike::setName(string aName) {
    mName = aName;
}
```
Fix the Bike

• Bike (version 2)
  – Use accessors and mutators to access members
  – Restrict member variables to private access
  – What if we change the internal type of the name variable
  – User code is NOT effected
  – Suppose mName was stored as a cString

```cpp
string Bike::getName() {
    // don’t do this.... Just for demonstration
    string tempName = mName;
    return tempName;
}

void Bike::setName(string aName) {
    strcpy(mName,aName.c_str());
}
```
Declaring an Object

• Once a class is defined, an object of the class is declared just as variables of any other type
  – Example: To create two objects of type Bicycle:
    
    ```
    class Bike {
        // class definition lines
    }
    
    Bike mtnBike;
    Bike roadBike;
    ```
The Assignment Operator

• Objects and structures can be assigned values with the assignment operator (=)
• Example:

```java
Bike mtnBike;
mtnBike.setName("Foes");
mtnBike.setSize(18);
mtnBike.setWheelDiameter(26.0);

Bike mtnBike2 = mtnBike;
```
Calling Public Members

• Recall that if calling a member function from the main function of a program, you must include the object name:

mtnBike.getName();
Calling Private Members

- When a member function calls a private member function, an object name is not used

```cpp
class Bike {
public:
    string getName();
    void setName(string aName);
    int getSize();
    void setSize(int aSize);
    double getWheelDiameter();
    void setWheelDiameter(double aDiameter);
private:
    bool validSize(int aSize);
    string mName;
    int mSize;
    double mWheelDiameter;
};
```
Calling Private Members

• When a member function calls a private member function, an object name is not used

```cpp
void Bike::setSize(int aSize) {
    if(validSize(aSize))
        mSize = aSize;
    else
        mSize = 10;
}

bool Bike::validSize(int aSize) {
    if(aSize <= 0)
        return false;
    return true;
}
```
Constructors

- A constructor can be used to initialize member variables when an object is declared
  - A constructor is a member function that is usually public
  - A constructor is automatically called when an object of the class is declared
  - A constructor’s name must be the name of the class
  - A constructor cannot return a value
    - No return type, not even void, is used in declaring or defining a constructor
A constructor for the Bike class could be declared as:

class Bike {
    public:
        Bike();
        Bike(string aName, int aSize, double aDiameter);
        // continued...

    private:
        // continued...
};
The constructor for the Bike class could be defined as

```cpp
Bike::Bike(string aName, int aSize, double aDiameter) {
    if(aName == "")
        mName = "Unknown Name";
    else
        mName = aName;
    if(aSize <= 0)
        mSize = 10;
    else
        mSize = aSize;
    if(aDiameter <= 4)
        mWheelDiameter = 5.0;
    else
        mWheelDiameter = aDiameter;
}
```

Note that the class name and function name are the same.
Calling A Constructor (1)

- A constructor is not called like a normal member function:

  Bike mtnBike;
  mtnBike.Bike("Foes", 18, 26.0);
Calling A Constructor (2)

• A constructor is called in the object declaration

Bike mtnBike2("Foes", 18, 26.0);

– Creates a Bike object and calls the constructor to initialize the member variables
Overloading Constructors

• Constructors can be overloaded by defining constructors with different parameter lists
  – Other possible constructors for the Bike class might be

    Bike (string aName);
    Bike (int aSize, double aDiameter);
The Default Constructor

• A default constructor uses no parameters
• A default constructor for the Bike class could be declared in this way

```cpp
class Bike {
public:
    Bike();
    // initializes name
    // initializes size
    // initializes wheelDiameter
    ... // The rest of the class definition
};
```
Default Constructor Definition

• The default constructor for the Bike class could be defined as
  Bike::Bike() {
      mName = "Unknown Name";
      mSize = 10;
      mWheelDiameter = 12.0;
  }

• It is a good idea to always include a default constructor even if you do not want to initialize variables
Calling the Default Constructor

• The default constructor is called during declaration of an object
  – An argument list is not used

    Bike mtnBike1;
    // uses the default Bike constructor

    Bike mtnBike1();
    // Is not legal
Initialization Sections

• An initialization section in a function definition provides an alternative way to initialize member variables

// Default Constructor
Bike::Bike() : mName("Unknown Name"), mSize(10), mWheelDiameter(12.0) {
}

• The values in parenthesis are the initial values for the member variables listed
Parameters and Initialization

- Member functions with parameters can use initialization sections

```cpp
Bike::Bike(string aName, int aSize, double aDiameter) :
    mName(aName), mSize(aSize), mWheelDiameter(aDiameter) {
    if(aName == "")
        mName = "Unknown Name";
    if(aSize <= 0)
        mSize = 10;
    if(aDiameter <= 4)
        mWheelDiameter = 5.0;
}
```

- Notice that the parameters can be arguments in the initialization
10.3

Abstract Data Types
Abstract Data Types

- A data type consists of a collection of values together with a set of basic operations defined on the values.
- A data type is an Abstract Data Type (ADT) if programmers using the type do not have access to the details of how the values and operations are implemented.
Classes To Produce ADTs

• To define a class so it is an ADT
  – Separate the specification of how the type is used by a programmer from the details of how the type is implemented
  – Make all member variables private members
  – Basic operations a programmer needs should be public member functions
  – Fully specify how to use each public function
  – Helper functions should be private members like validSize()
ADT Interface

• The ADT interface tells how to use the ADT in a program
  – The interface consists of
    • The public member functions
    • The comments that explain how to use the functions
  – The interface should be all that is needed to know how to use the ADT in a program
ADT Implementation

• The ADT implementation tells how the interface is realized in C++
  – The implementation consists of
    • The private members of the class
    • The definitions of public and private member functions
  – The implementation is needed to run a program
  – The implementation is not needed to write the main part of a program or any non-member functions
ADT Benefits

• Changing an ADT implementation does require changing a program that uses the ADT

• ADT’s make it easier to divide work among different programmers
  – One or more can write the ADT
  – One or more can write code that uses the ADT

• Writing and using ADTs breaks the larger programming task into smaller tasks
Interface Preservation

• To preserve the interface of an ADT so that programs using it do not need to be changed
  – Public member declarations cannot be changed
  – Public member definitions can be changed
  – Private member functions can be added, deleted, or changed
Information Hiding

• Information hiding was referred to earlier as writing functions so they can be used like black boxes

• ADT’s implement information hiding because
  – The interface is all that is needed to use the ADT
  – Implementation details of the ADT are not needed to know how to use the ADT
  – Implementation details of the data values are not needed to know how to use the ADT