COSC 220: Computer Science II

Module 3

Instructor:

Dr. Xiaohong (Sophie) Wang
(xswang@salisbury.edu)

Department of Mathematics & Computer Science

Salisbury University

Spring 2021

Content

1. Structured Data
   1.1 Abstract Data Types
   1.2 Array of Structures
   1.3 Structures as Function Arguments
   1.4 Pointers to Structures
   1.5 Enumerated Data Types
2. Classes

1.1 Abstract Data Types

- An abstract data type (ADT) is a new **data type** created by the programmer
  - Compared with primitive data types, such as `int`, `bool`, `char`, etc.

- An ADT specifies
  - the **primitive data types** it contains
  - **operations** that can be performed on these data types

What does "abstract" mean here?

Abstract Data Types

- Abstraction: a definition that captures general characteristics without details
- For example
  - A student has attributes such as `studentID`, `name`, `yearInSchool`, `gpa`, etc.
  - ADT enables us to define a new data type named `Student` that represents all the students
    - Each variable of this `Student` data type represents a student (an instance of the `Student` category)

How to define an abstract data type?

Combining Data into Structures

- Structure:
  - C++ allows you to group multiple **member variables** together into a single item known as structure
- General Format:

```
struct StructName
{
   dataType1 memberName1;
   dataType2 memberName2;
   . . .
};
```

  - Must have `;` after closing `}`
  - `StructName` commonly begin with uppercase letter
  - Multiple members of same type can be in comma-separated list:

```
string name, address;
```

Example

```
struct Student
{
   int studentID;
   string name;
   short yearInSchool;
   double gpa;
};
```

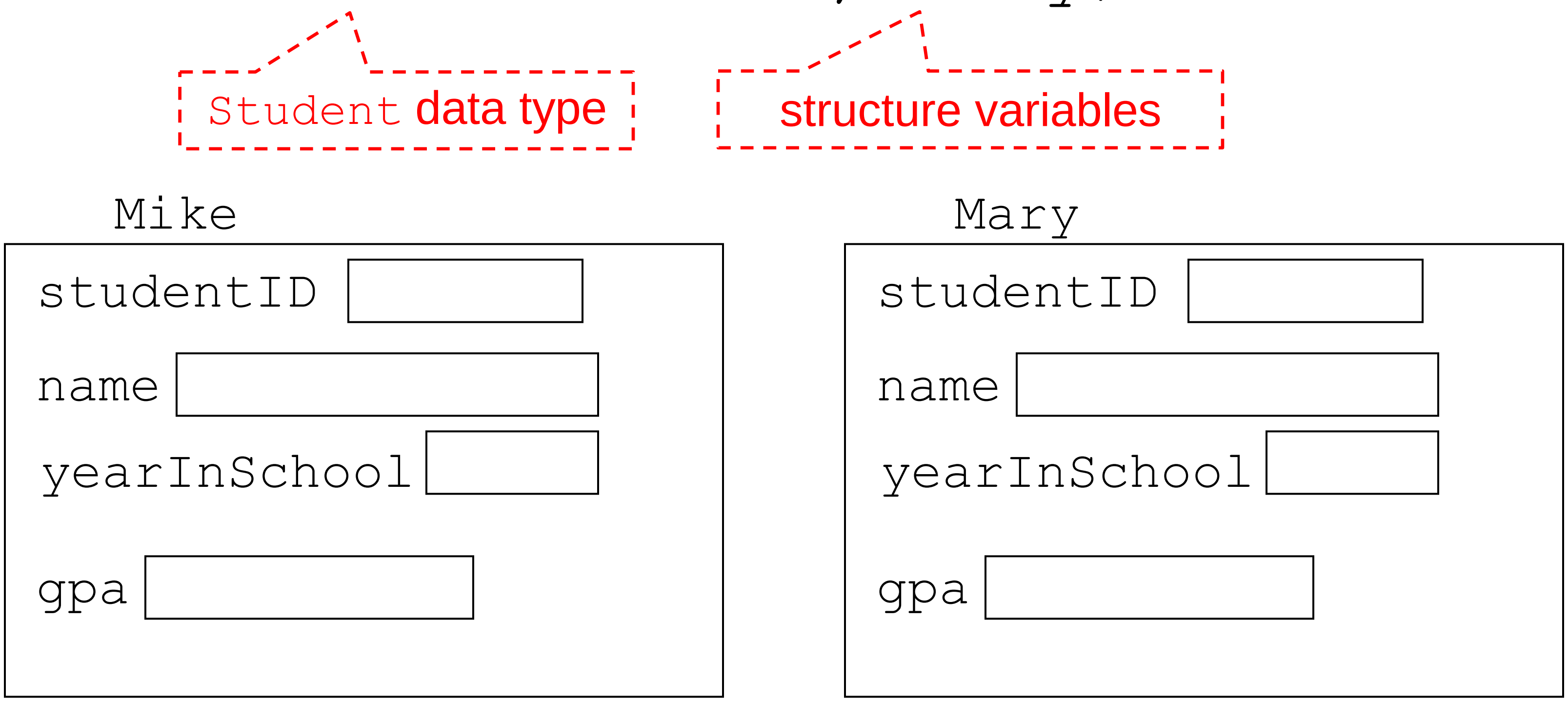
structure tag

structure members

Defining Variables

- To define structure variables, use *`StructName`* as <u>data type name</u>:

```
Student Mike, Mary;
```



Note: Each **structure variable** is an instance that contains all the **member variables**.

Accessing Structure Members

- Use the dot `(.)` operator to refer to member variables (or members) of `struct` variables:

```
Student stu1;
cin >> stu1.studentID;
getline(cin, stu1.name);
stu1.yearInSchool = 2;
stu1.gpa = 3.75;
```

- To display the contents of a `struct` variable, must display each member separately, using the dot operator

```
cout << stu1; // won't work
cout << stu1.studentID << endl;
cout << stu1.name << endl;
cout << stu1.yearInSchool;
cout << " " << stu1.gpa;
```

Note: With the dot operator, you can use member variables just like regular variable.

Example

```
#include <iostream>
#include <cmath>
using namespace std;

const double PI = 3.14159;

struct Circle{
    double radius, diameter, area;
};

int main(){
    Circle c;

    cout << "Enter the diameter of a circle: ";
    cin >> c.diameter;
    c.radius = c.diameter / 2;
    c.area = PI * pow(c.radius,2.0);

    cout << "The radius of the circle is: " << c.radius << endl;
    cout << "The area of the circle is: " << c.area << endl;
    return 0;
}
```

Output:
Enter the diameter of a circle: 10
The radius of the circle is: 5
The area of the circle is: 78.5397

Initializing a `struct` variable

- `struct` variable can be initialized when defined:

```
Student s = {11465, "Joan", 2, 3.75};
```

- Can also be initialized member-by-member after definition:

```
s.name = "Joan";
s.gpa = 3.75;
```

1.2 Array of Structures

- An array of structures is an array that contains multiple same-type structures

```
struct BookInfo{
    string title, author, publisher;
    double price;
}

BookInfo bookList[20];
```

- Individual structures are accessible using subscript notation
- Members within a structure are accessible using dot notation

```
bookList[5].title
```

1.3 Structures as Function Arguments

- May pass members of `struct` variables to functions

```
struct Rectangle{
    double length, width, area;
};

double multiply(double x, double y){
    return x * y;
}

Rectangle box = {3.0, 4.0};
box.area = multiply(box.length, box.width);
```

Structures as Function Arguments

- May pass entire `struct` variables to functions:

```
struct Rectangle{
    double length, width, area;
};

void showRect(Rectangle r){
    cout << r.length << endl;
    cout << r.width << endl;
    cout << r.area << endl;
}

Rectangle box = {3.0, 4.0, 12.0};
showRect(box);
```

Structures as Function Arguments

- Can use reference parameter if function needs to modify contents of structure variable

```
struct Rectangle{
    double length, width, area;
};

void rectArea(Rectangle &r){
    cout << "Enter the box length and width: ";
    cin >> r.length >> r.width;
    r.area = r.length * r.width;
}

int main(){
    Rectangle box;
    rectArea(box);
    cout << "The box length is: " << box.length << endl;
    cout << "The box width is: " << box.width << endl;
    cout << "The box area is: " << box.area << endl;
}
```

```
Enter the box length and width: 3.0 4.0
The box length is: 3
The box width is: 4
The box area is: 12
```

In-class practice

- Programming challenges 1 (Page 659)
  - Write a program that uses a structure named `MovieData` to store the following information about a movie:
    - Title
    - Director
    - Year Released
    - Running Time (in minutes)
  - The program should create two `MovieData` variables, store values in their members, and pass each one, in turn, to a function that displays the information about the movie in a clearly formatted manner.

Reference code: spc11-1.cpp

1.4 Pointers to Structures

- A structure variable has an address
- A pointer to structure is a variable that can hold the address of a structure:

```
Student *stuPtr;
```

- Can use **&** operator to assign address:

```
stuPtr = &stu1;
```

- Structure pointer can be a function parameter

Accessing Structure Members via Pointer

- Must use `()` to dereference pointer variable
  - As the dot operator "." has higher precedence than the indirection operator "*"

```
cout << (*stuPtr).studentID;
```

- Can use structure pointer operator "->" to eliminate `()` and use clearer notation

```
cout << stuPtr->studentID;
```

Example

```
#include <iostream>
#include <string>
using namespace std;

struct Student{
    string name;
    int idNum, creditHours;
    double gpa;
};

void getData(Student *);  //Function prototype

int main(){
    Student freshman;
    getData(&freshman);

    cout << "\nThe student's information: \n";
    cout << "Name: " << freshman.name << endl;
    cout << "ID Number: " << freshman.idNum << endl;
    cout << "Credit Hours: " << freshman.creditHours << en
dl;
    cout << "GPA: " << freshman.gpa << endl;
    return 0;
}
```

Example (continue)

```
void getData(Student *s){
    cout << "Input student name: ";
    getline(cin, s->name);
    cout << "Input student ID number: ";
    cin >> s->idNum;
    cout << "Input student credit hours: ";
    cin >> s->creditHours;
    cout << "Input student GPA: ";
    cin >> s->gpa;
}
```

Input student name: Frank Smith
Input student ID number: 4876
Input student credit hours: 12
Input student GPA: 3.9

The student's information:
Name: Frank Smith
ID Number: 4876
Credit Hours: 12
GPA: 3.9

Dynamically Allocating a Structure

- Can use a structure pointer and the **`new`** operator to dynamically allocate a structure

```
struct Circle {
    double radius, diameter, area;
};

Circle *cirPtr = nullptr;
cirPtr = new Circle;
cirPtr -> radius = 10;
cirPtr -> diameter = 20;
cirPtr -> area = 314.159;
```

1.5 Enumerated Data Types

- An enumerated data type is a programmer-defined data type. It consists of values known as ***enumerators***, which represent integer constants.
- Example:

```
enum Day { MONDAY, TUESDAY,
           WEDNESDAY, THURSDAY,
           FRIDAY };
```

  - The identifiers `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, and `FRIDAY` are *enumerators*. They represent the values that belong to the `Day` data type.

Note: The *enumerators* are not strings and aren't enclosed in quotes. They are identifiers.

Enumerated Data Types

- Once you have created an enumerated data type in your program, you can define variables of that type. Example:

```
Day workDay;
```

- We may assign any of the enumerators `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, or `FRIDAY` to a variable of the `Day` type. Example:

```
workDay = WEDNESDAY;
```

Enumerated Data Types

- An *enumerator* is an integer named constant
- Internally, the compiler assigns integer values to the enumerators, beginning at 0.

```
enum Day { MONDAY, TUESDAY,
           WEDNESDAY, THURSDAY,
           FRIDAY };
```

In memory...

```
MONDAY    = 0
TUESDAY   = 1
WEDNESDAY = 2
THURSDAY  = 3
FRIDAY    = 4
```

Example

- Using the `Day` declaration, the following code...

```
cout << MONDAY << " "
     << WEDNESDAY << " “
     << FRIDAY << endl;
```

...will produce this output:

```
0 2 4
```

Assigning an integer to an `enum` Variable

- You cannot directly assign an integer value to an `enum` variable. This will not work:

```
workDay = 3; // Error!
```

- Instead, you must cast the integer:

```
workDay = static_cast<Day>(3);
```

- However, you CAN assign an enumerator to an `int` variable.
  - This following code assigns 3 to `x`.

```
int x;
x = THURSDAY;
```

2. Classes

2.1 Procedural and Object-Oriented Programming

2.2 Introduction to Classes

2.3 Constructors

2.4 Destructors

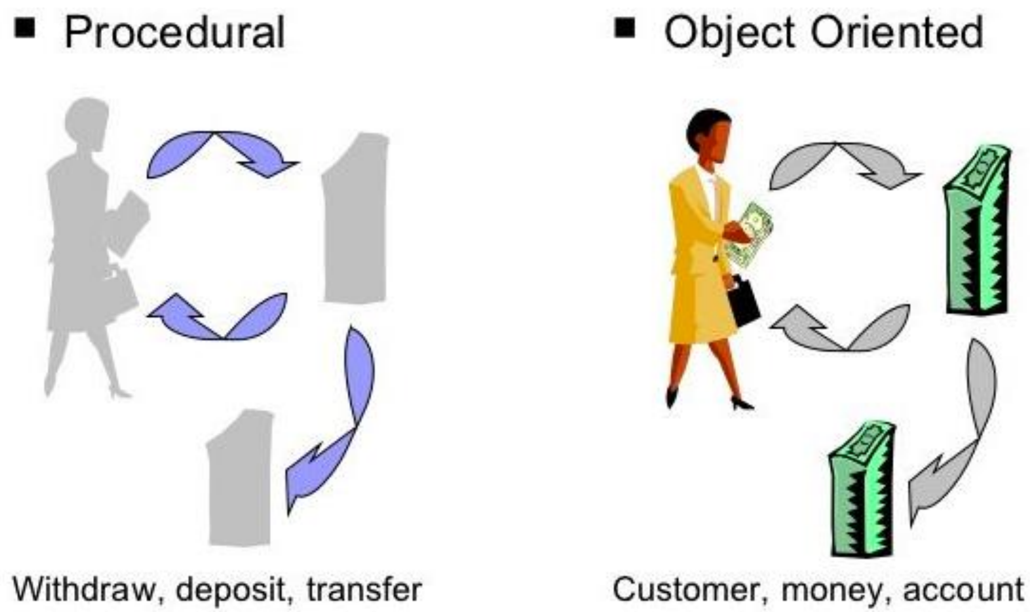
2.5 Overloading Constructors

2.6 Copy Constructors

2.7 Operator Overloading

2.1 Procedural and Object-Oriented Programming

- Procedural programming focuses on the **process/actions** that occur in a program
- Object-Oriented programming is based on the **data** and the **functions** that operate on it. Objects are instances of ADTs that represent the data and its functions



https://www.alphansotech.com/procedural-vs-object-oriented-programming

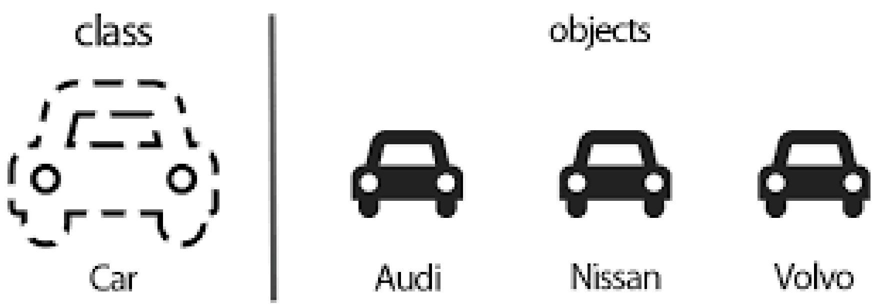
Procedural and Object-Oriented Programming

Procedural Programming	Object-Oriented Programming
• Program is divided into parts called **functions**	• Program is divided into parts called **objects**
• Top down design	• Object focused design
• Limited code reuse	• Code reuse
• Complex code	• Complex design
• Global data focused	• Protected data
• Less secure	• More secure

 https://www.slideshare.net/HarisBinZahid/procedural-vs-object-oriented-programming

Classes and Objects

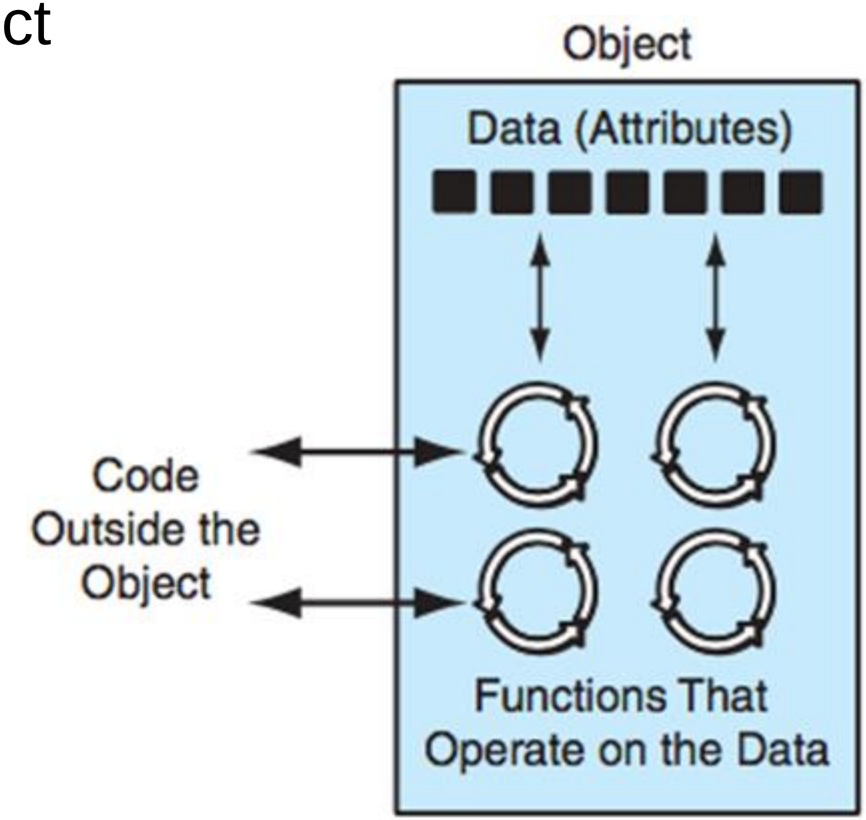
- class: A `class` is a code template for creating objects. It specifies the attributes (member variables) and behaviors (member functions) that a particular type of objects may have
- object: An object is an instance of a `class`. It has all the attributes and behaviors defined in the `class`



A Class is like a template and objects are built from the template

Encapsulation and Data Hiding

- Encapsulation: combine data and code into a single object
- Data hiding: hide data from code that is outside the object
- Public interface: data and functions of an object that are available outside of the object
- Imagine the "simple" interface to drive a vehicle: it "hides" very complex functionality from the user
  - The interfaces are **public** members (attributes & functions)
  - The information is hided in **private** members



http://faculty.salisbury.edu/~jtanderson/teaching/cosc220/sp20/index.html

2.2 Introduction to Classes

- Class declaration:

```
class ClassName
{
    declaration;
    // ... More declarations;
};
```

Example:

```
class Rectangle {
    double width;
    double length;
};
```

- The declaration statements are for the **variables (attributes)** and **functions (behaviors)**, which are members of that class
- The members of a class are `private` by default, i.e. these private members can't be accessed by code outside the class

How to define members that can be accessed from outside the class?

Access Specifiers

- Used to control access to members of the class
  - **`public`:** can be accessed by functions outside of the class
  - **`private`:** can only be called by or accessed by functions that are members of the class
  - Can be listed in any order and appear multiple times

```
class ClassName
{
    private:
        // Declarations of private members
    public:
        // Declarations of public members
};
```

Access specifiers are followed by a colon then followed by one or more member declarations

Example

```
class Rectangle
{
   private:
      double width;
      double length;
   public:
      void setWidth(double);
      void setLength(double);
      double getWidth() const;
      double getLength() const;
      double getArea() const;
};
```

Two `private` member variables (attributes), which can be accessed **ONLY** by the member functions in this class

1. Five `public` member functions (behaviors), which can be called from statements outside the class.
2. They are only declarations. The implementation of member functions will be introduced later.

Note: You may understand **encapsulation**, **data hiding**, **public interface** from this example.

Defining a Member Function

- When defining a member function:
  - Put prototype in class declaration
  - Define function **outside (after)** the class declaration, using class name and scope resolution operator (**::**)

`ReturnType ClassName::functionName(ParameterList)`

```
void Rectangle::setWidth(double w)
{
        width = w;
}

...
int Rectangle::getWidth() const
{
        return width;
}
```

Inline Member Functions

- Member functions can be defined
  - **in** class declaration (inline member functions)
  - **after** the class declaration (regular member functions)
- Inline appropriate for <u>short function bodies</u>:

```
int getWidth() const
{
        return width;
}
```

- Code for an inline function is copied into program in place of call – larger executable program, but no function call overhead, hence faster execution

Accessors and Mutators

- Mutator: a member function that stores a value in a private member variable, or changes its value in some way
- Accessor: function that retrieves a value from a private member variable. Accessors do not change an object's data, so they should be marked `const`. For example:

```
double getWidth() const;
double getLength() const;
double getArea() const;
```

Note: `const` appearing after the parentheses in a member function declaration specifies that the function will not change any data in the calling object

Defining an Instance of a Class

- An object is an instance of a class
- Object definition:

```
ClassName objectName;

Rectangle r;
```

- Access members using **dot operator**:

```
r.setWidth(5.2);
cout << r.getWidth();
```

- Compiler error if attempt to access `private` member using dot operator

Example

- Program 13-1
  - Refer to “Pr13-1.cpp”

Pointer to an Object

- Can define a pointer to an object. The pointer holds the address of this object.

```
Rectangle myRectangle;
Rectangle *rectPtr = nullptr;
rectPtr = &myRectangle;
```

- Pointer can access public members using "**->**" operator:

```
rectPtr->setLength(12.5);
cout << rectPtr->getLength() << endl;
```

Dynamically Allocating an Object

- We can also use a pointer to dynamically allocate an object.

```
// Define a Rectangle pointer.
Rectangle *rectPtr = nullptr;

// Dynamically allocate a Rectangle object.
rectPtr = new Rectangle;

// Store values in the object's width and length.
rectPtr->setWidth(10.0);
rectPtr->setLength(15.0);

// Delete the object from memory.
delete rectPtr;
rectPtr = nullptr;
```

In-class practice

- Define a `Car` class that contains
  - 3 `private` attributes (member variables) named `make`, `model`, and `year`
  - 3 `public` behaviors (member functions) named `setMake`, `setModel`, and `setYear` to set the values of above 3 attributes
  - 3 `public` behaviors named `getMake`, `getModel`, and `getYear` to return the values of above 3 attributes
- In the main program,
  - create a `Car` object named `myCar`
  - ask the user to input the make, model, and year of this car
  - call `setMake`, `setModel`, and `setYear` functions to store the input information
  - call `getMake`, `getModel`, and `getYear` to return these information and print it out
- Test your code

Reference code: Car.cpp

Separating Specification from Implementation

- Place class declaration in a header file that serves as the class specification file. Name the file ***`ClassName.h`***. For example, `Rectangle.h`
- Place member function definitions in class implementation file named ***`ClassName.cpp`***. For example, `Rectangle.cpp`. File should `#include` the class specification file
- Programs that use the class must `#include` the class specification file, and be compiled and linked with the class implementation file

Example: Rewrite Pr13-1 to Pr13-4

Rectangle.h

```
// Specification file for the Rectangle
class.
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle{
   private:
      double width;
      double length;
   public:
      void setWidth(double);
      void setLength(double);
      double getWidth() const;
      double getLength() const;
      double getArea() const;
};

#endif
```

Rectangle.cpp

```
// Implementation file for the Rectangle class.
#include "Rectangle.h"   // Enclosed in " ", not in < >
#include <iostream>      // Needed for cout
#include <cstdlib>       // For the exit function
using namespace std;

void Rectangle::setWidth(double w){
   if (w >= 0)
      width = w;
   else{
      cout << "Invalid width\n";
      exit(EXIT_FAILURE);
   }
}

void Rectangle::setLength(double len){
   … …
}
... ... // Other functions
```

Note: #ifndef checks whether the given token has been defined earlier in the file or in an included file; if not, it includes the code between the #define and #endif statements

 https://www.cprogramming.com/reference/preprocessor/ifndef.html

Example (Cont'd)

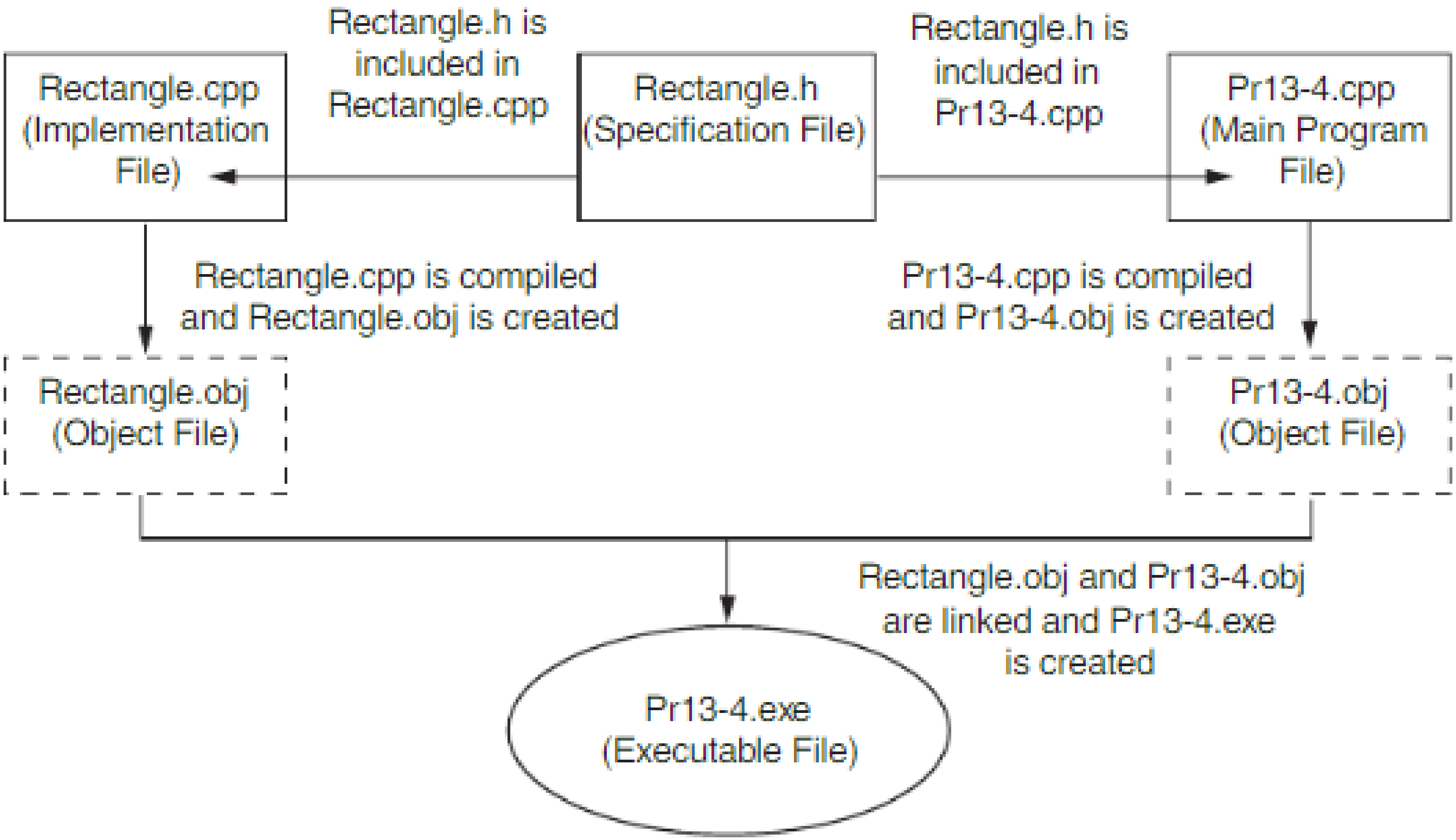
Main program

```
// This program uses the Rectangle class, which is declared in the Rectangle.h file.
// The Rectangle class's member functions are defined in the Rectangle.cpp file.
// This program should be compiled with those files in a project.
#include <iostream>
#include "Rectangle.h"  // Enclosed in " ", means the ".h" file is in current directory
using namespace std;

int main() {
   Rectangle box;     // Define an instance of the Rectangle class
   double rectWidth;  // Local variable for width
   double rectLength; // Local variable for length

   // Get the rectangle's width and length from the user.
   cout << "This program will calculate the area of a\n";
   cout << "rectangle. What is the width? ";
   cin >> rectWidth;
   cout << "What is the length? ";
   cin >> rectLength;
   … …
}
```

Note: Include class's header file in both implementation file and the main program file.

Example (Cont'd)

- Steps of creating an executable file



2.3 Constructors

- A constructor is a member function that is automatically called when an object is created
- Purpose is to initialize attributes of an object
- Constructor function name is **same** as the class name
- Has no return type

```
ClassName::ClassName(ParameterList)
{
    // Statements;
}
```

Example (Rectangle class)

Rectangle.h

```
// Specification file for Rectangle class
// This version has a constructor.
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle
{
  private:
    double width;
    double length;
  public:
    Rectangle();      // Constructor
    void setWidth(double);
    void setLength(double);
    double getWidth() const
      { return width; }
    double getLength() const
      { return length; }
    double getArea() const
      { return width * length; }
};
#endif
```

Rectangle.cpp

```
// Implementation file for the Rectangle class.
// This version has a constructor.
#include "Rectangle.h"
#include <iostream>      // Needed for cout
#include <cstdlib>       // Needed for the exit function
using namespace std;

Rectangle::Rectangle()
{
  width = 0.0;
  length = 0.0;
}

void Rectangle::setWidth(double w){
  if (w >= 0)
    width = w;
  else{
    cout << "Invalid width\n";
    exit(EXIT_FAILURE);
  }
}
… …
```

Default Constructors

- A default constructor is a constructor that **takes no arguments**.
- If you write a class with no constructor at all, C++ will write a default constructor for you, one that **does nothing**.
- A simple instantiation of a class (with no arguments) calls the default constructor:

```
Rectangle r;
```

Passing Arguments to Constructors

- To create a constructor that takes arguments:
  - Indicate parameters in the constructor declaration:

```
Rectangle(double, double);
```

  - Use parameters in the constructor implementation:

```
Rectangle::Rectangle(double w, double len)
{
   width = w;
   length = len;
}
```

  - Pass arguments to the constructor when you create an object

```
Rectangle r(10, 5);
```

Example

Rectangle.h

```
// Specification file for Rectangle class
// This version has a constructor.
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle {
  private:
    double width;
    double length;
  public:
    Rectangle(double, double); //Constructor
    void setWidth(double);
    void setLength(double);

    double getWidth() const
      { return width; }
    double getLength() const
      { return length; }
    double getArea() const
      { return width * length; }
};
#endif
```

Rectangle.cpp

```
// Implementation file for the Rectangle class.
// The constructor accepts arguments.
#include "Rectangle.h"
#include <iostream>
#include <cstdlib>
using namespace std;

Rectangle::Rectangle(double w, double len) {
  width = w;
  length = len;
}

void Rectangle::setWidth(double w) {
  if (w >= 0)
    width = w;
  else
  {
    cout << "Invalid width\n";
    exit(EXIT_FAILURE);
  }
}
… …
```

Example (Cont'd)

Main program

```
// This program calls the Rectangle class constructor.
#include <iostream>
#include <iomanip>
#include "Rectangle.h"
using namespace std;

int main() {
   double houseWidth,   // To hold the room width
               houseLength;  // To hold the room length

   // Get the width of the house.
   cout << "In feet, how wide is your house? ";
   cin >> houseWidth;

   // Get the length of the house.
   cout << "In feet, how long is your house? ";
   cin >> houseLength;

   // Create a Rectangle object.
   Rectangle house(houseWidth, houseLength );
       … …
}
```

Using Default Arguments with Constructors

- A constructor may have default arguments
- The default value is listed in the parameter list of the function's declaration or the function header

```
Rectangle::Rectangle(double w, double len = 12.0)
{
    width = w;
    length = len;
}


Rectangle house(houseWidth);
```

When only one argument is passed to the constructor function, the default `12.0` will be assigned to `len`

More About Default Constructors

- If a constructor has default arguments for **all** its parameters, it can be called with no explicit arguments. Then it becomes the default constructor. For example:

```
Rectangle::Rectangle(double w = 10.0, double len = 12.0)
{
   width = w;
   length = len;
}
```

- In this case, the constructor can be called with no argument:

```
Rectangle r;
```

In-class practice

- Programming challenges 3 (Page 808)
  - Write a class named `Car` that has the following member variables:
    - `yearModel` – an `int` that holds the car's year model
    - `make` – a `string` that holds the make of the car
    - `speed` – an `int` that holds the car's current speed
  - In addition, the class should have the following constructor and other member functions:
    - Constructor – Accept the car's year model and make arguments to initial `yearModel` and `make` member variables; assign 0 to `speed`
    - Accessor – return the values of `yearModel`, `make`, and `speed`
    - `accelerate` – add 5 to the `speed` each time it is called
    - `brake` – subtract 5 from the `speed` each time it is called
  - Demonstrate the class in a program that creates a `Car` object, then call the `accelerate` function 5 times. After each call to the `accelerate` function, get the current `speed` of the car and display it. Then, call the `brake` function 5 times. After each call to the `brake` function, get the current `speed` of the car and display it

Reference code: spc13-3.cpp

2.4 Destructors

- A destructor is a member function that is automatically called when an object is destroyed
- Destructors perform shutdown procedures when the object goes out of existence.
  - For example: to free memory that was dynamically allocated by the class object
- Destructor name is `~ClassName`, *e.g.*, `~Rectangle`
- Has no return type; takes no arguments
- Only one destructor per class, *i.e.*, it cannot be overloaded

Example (ContactInfo.h)

```
#ifndef CONTACTINFO_H
#define CONTACTINFO_H
#include <cstring>      // Needed for strlen and strcpy

class ContactInfo {
private:
       char *name;     // The contact's name
       char *phone;    // The contact's phone number
public:
       ContactInfo(char *n, char *p)    // Constructor
       { // Allocate enough memory for the name and phone number.
         name = new char[strlen(n) + 1];
         phone = new char[strlen(p) + 1];
         // Copy the name and phone number to the allocated memory.
         strcpy(name, n);
         strcpy(phone, p); }

       ~ContactInfo()          // Destructor
       { delete [] name;
         delete [] phone; }

       const char *getName() const
       { return name; }

       const char *getPhoneNumber() const
       { return phone; }
};
#endif
```

2.5 Overloading Constructors

- A class can have more than one constructor
- Overloaded constructors in a class **must** have different parameter lists:

```
Rectangle();
Rectangle(double);
Rectangle(double, double);
```

Example

```
class InventoryItem {
private:
   string description; // The item description
   double cost;        // The item cost
   int units;          // Number of units on hand
public:
   InventoryItem(){    // Constructor #1 (default constructor)
      description = "";
      cost = 0.0;
      units = 0; }

   InventoryItem(string desc){    // Constructor #2
      description = desc;
      cost = 0.0;
      units = 0; }

   InventoryItem(string desc, double c, int u){   // Constructor #3
      description = desc;
      cost = c;
      units = u; }
  … …
};
```

Member Function Overloading

- Non-constructor member functions can also be overloaded
- Must have unique parameter lists

```
void setCost(double c){ // cost stored in double
    cost = c;
}

void setCost(string c){ // cost stored in a string
    cost = stod(c);
}
```

`stod` function converts the string to a double

2.6 Operator Overloading

- **Operator overloading**: redefine how standard operators (**=**, **+**, **etc.**) work when used with class objects
  - The operands are objects

- An example of overloaded operators:
  - Floating-point division: `5.0 / 2 = 2.5`
  - Integer division: `5 / 2 = 2`

Operator Overloading

- The name of the function for the overloaded operator is `operator` followed by the operator symbol, *e.g.*,
  - **`operator+`** to overload the **`+`** operator, and
  - **`operator=`** to overload the **`=`** operator
- Prototype for the overloaded operator goes in the declaration of the class that is overloading it
- Overloaded operator function definition goes with other member functions

The `this` Pointer

- `this`: a built-in pointer that every class has
  - available to a class's member functions
  - always points to the instance (object) of the class whose function is being called
  - is passed as a hidden argument to all non-static member functions
- Assume `student1` and `student2` are two `StudentTestScores` objects (*page 835*)

```
cout << student1.getStudentName() << endl;
```

  - When run the above line, `this` pointer points to `student1`

```
cout << student2.getStudentName() << endl;
```

  - When run the above line, `this` pointer points to `student2`

Here `getStudentName` is a member function of `StudentTestScores` class

Overloading the = Operator

- Define a member function called **=** operator function

  - Prototype:

```
const SomeClass operator=(const SomeClass &right);
```

return type [1]

function name

parameter for object on right side of operator

[1] https://www.linuxtopia.org/online_books/programming_books/thinking_in_c++/Chapter08_014.html

Overloading the = Operator

- Define a member function called **=** operator function
  - **=** operator function implementation

```
// Overloaded = operator
const SomeClass SomeClass::operator=(const SomeClass &right){
    if (this != &right){  //left and right objects are not same
        value = new int;
        *value = *(right.value);
        }
     return *this;    // dereference the this pointer, giving
}                     // us the actual object that received the
                      // assignment
```

Overloading the = Operator

- Invoke the = operator function

```
SomeClass object1(5);
SomeClass object2;
object2 = object1;
object1.setVal(13);
cout << object1.getVal() << endl;
cout << object2.getVal() << endl;
```

Output:
13
5

- Operator can be invoked as a member function:

```
object2.operator=(object1);
```

Same as:

```
object2 = object1;
```


Example code: OperatorOverloading.cpp

Returning a Value

- Overloaded operator can return a value

```
class Point2d
{
private:
     int x, y;
...
public:
  double operator-(const point2d &right)
  { return sqrt(pow((x-right.x),2)
                + pow((y-right.y),2)); }
};

Point2d point1(2,2), point2(4,4);

// Compute and display distance between 2 points.
cout << point2 - point1 << endl; // displays 2.82843
```

Notes on Overloaded Operators

- Can change meaning of an operator
- Can **NOT** change the number of operands of the operator
- Only certain operators can be overloaded. Can **NOT** overload the following operators:

  `?:      .      .*      ::      sizeof`

  - Overloading prefix/postfix ++ operator (page 849)
  - Overloading relational operators (page 852)
  - Overloading << and >> operators (page 854)
  - Overloading `[]` operator (page 858)

Reading textbook

- Chapter 11, 13, 14

Reference

- The teaching materials of this course refer to:
  - Professor Xiaohong (Sophie) Wang. COSC 120 teaching materials
    - Salisbury University
  - Textbook:
    - Starting Out with C++: From Control Structures through Objects, by Tony Gaddis, Pearson (9th Edition)
    - Instructor materials of the above textbook (All rights reserved)