Tutorial Letter 102/0/2024

Introduction to Programming II

COS1512

School of Computing

Study Guide for COS1512

BARCODE

Define tomorrow.

UNISA university of south africa

Study Guide for COS1512

Contents

Introduction

Welcome to COS1512, an introduction to objects and the object-oriented programming environment. This study guide accompanies the prescribed book for COS1512:

Walter Savitch. Problem Solving with C++*, 10th edition*. Pearson Education, Inc, 2018.

We will refer to the prescribed book as Savitch.

Note that this study guide is applicable to the 7th, 8th, 9th and 10th edition of Savitch throughout.

Tutorial Letter 101 specifies the sections in Savitch that are covered in COS1512. We repeat them here for your convenience. Some of the work in the prescribed book is also covered by COS1511, and therefore only certain sections and chapters are covered in COS1512. The path that COS1512 follows through the prescribed book can be outlined broadly as follows:

Section 1.2 in chapter 1 provides a general overview over programming and problem-solving with a brief introduction to object-oriented programming. Though most of chapters 4 and 5 have been covered in COS1511, section 4.6 (overloading functions) and section 5.5 (the `assert` macro) are included to ensure that students have the necessary background knowledge to understand and implement object-oriented programming. Chapter 6 uses file I/O streams as an introduction to objects and classes, and teaches students how to use pre-defined classes. Chapter 8 builds on using pre-defined classes by introducing the standard class `string`. Chapter 8 also covers C strings and provides a preview of the Standard Template Library (STL) with the `vector` class. Chapter 9 further adds to the background knowledge necessary to understand and implement classes and objects by discussing pointers and dynamic arrays. In Chapter 10 students learn how to define their own classes in order to create an abstract data type (ADT). Chapter 10 also covers inheritance briefly in order to make students aware of this concept. Chapter 11 continues to teach more techniques for defining functions and operators for classes. Chapter 12 covers separate compilation, to allow placing the interface and implementation of an ADT in files separate from each other and separate from the programs that use the ADT. In Chapter 14 recursion is introduced. In Chapter 15 single inheritance, i.e. deriving one class from another is covered. In Chapter 17 function and class templates are covered, which will allow students to understand and use the STL.

The specific learning outcomes and assessment criteria for COS1512 are provided in section 2 of Tutorial Letter 101. These specific learning outcomes translate to *learning objectives* for the study material which should be achieved in order to realise the learning outcomes. This study guide specifies the learning objectives for each section in the prescribed book.

The discussion of each chapter in Savitch given in this tutorial letter typically includes these subsections:

- Overview
- Learning Objectives.

The **Overview** indicates the main issues addressed in each chapter. The **Learning Objectives** of each chapter are the skills you should acquire when mastering the content of the chapter. You will be tested on these skills in the examination. These objectives can be achieved by doing the self-test exercises and the assignments, implementing the examples given in the textbook and doing some additional exercises.

Supplementary information is given where we feel that the explanation given in the prescribed book is inadequate. Please read these sections carefully, as you may not be able to complete the assignments without the additional information.

Guide to the prescribed book

Chapter 1: Introduction to Computers and C++ Programming

Savitch: **Section 1.1** Compilers

Section 1.2 Programming and Problem-Solving

1.1 Overview

Much of what is covered in this chapter you probably have encountered somewhere in your programming studies up to now. We want to bring two specific aspects in this chapter to your attention: the process of compiling a C++ program, and the concept of object-oriented programming. Though you will not be examined on this chapter as such, it serves as a brief orientation to the concepts covered in this course.

1.2 Learning Objectives

After having read sections 1.1 and 1.2, you should

- be aware of what the process of compiling and linking a C++ program entails;
- have a broad understanding of the concept of object-oriented programming.

1.3 Executing a C++ Program

As is indicated in Savitch, the process to translate and execute a C++ program is more complicated than the brief version given in the textbook. C++ programs typically go through six phases to be executed: edit, preprocess, compile, link, load and execute.

A *text editor* is used to type the program and make corrections if necessary. The program is then stored in a file (the `.cpp` file) on a secondary storage device such as the hard disk. This program is called the *source code* or source program. The Code::Blocks integrated development environment (IDE) provides the text editor we use in this course, and forms the interface to the compiler, the linker and the loader.

In the next step, the programmer gives the command to *compile* the program. The *compiler* checks for syntax errors and translates the C++ source program into machine language code (also referred to as *object code* and stored in a file with extension .o). In a C++ system, the compiler invokes a *preprocessor* program automatically before the compiler's translation phase begins. The C++ preprocessor executes special commands called preprocessor directives. Preprocessor directives typically call for the inclusion of other text files in the file that has to be compiled and therefore perform various text replacements. We use the MinGW compiler. Errors indicated by the compiler are corrected using the text editor.

C++ programs usually contain references to functions defined elsewhere, such as in private (user-defined) libraries or the standard libraries. The code for the functions residing in the libraries have to be combined with the object code produced by the C++ compiler. This is done by a *linker*. A linker *links* the object code produced from your source code with the code for the missing functions to create the *executable code* (saved in a file with the extension `.exe`). Next, the executable code is *loaded* into the main memory for execution. This is done by the *loader*, which transfers the executable program from disk to memory. Finally the computer *executes* the program.

1.4 Programming Methodologies

The two most popular programming paradigms are structured procedural programming and object-oriented programming.

Structured Procedural Programming

Procedural programming focuses on the *processes* that data undergoes from input until meaningful output is produced. During design the processing steps are described in an algorithm. These steps are typically sets of operations executed in sequence after each other. Structured programming uses a technique called top-down design or step-wise refinement. This refines or divides a problem into subproblems until eventually each subproblem is straightforward enough to be solved easily. In the structured procedural approach each subproblem is normally written as a separate function (or procedure) that is called by a main controlling function or module. A structured procedural program defines variables and then calls a series of procedures to input values for the variables, manipulate them and output the results.

Object-Oriented Programming

In object–oriented programming applications are organised around *objects* rather than processes. In the object-oriented paradigm, a system is seen as a collection of interacting objects that models the interaction between objects necessary to achieve the desired effect. In object-oriented programming, program components are envisioned as objects that belong to classes and are similar to real-world objects; the programmer then manipulate the objects and have them interrelate to achieve a desired result. With object-oriented analysis, design and programming, the focus is on determining the objects you want to manipulate rather than the processes or logic required to manipulate the data. Object-oriented programming involves three fundamental concepts: *encapsulation*, *inheritance* and *polymorphism*.

Writing object-oriented programs involves creating classes, creating objects from those classes, and creating applications, i.e. stand-alone executable programs, that use those objects. If they are designed properly, classes can be reused over and over again to develop new programs.

An object generally represents a concrete entity in the real world such as a person, place or thing. Such an object has certain characteristics or *attributes* and certain *behaviours* or actions. An object’s characteristics represent data or facts that we know about the object. This is sometimes called the state of the object. The object’s behaviours are actions that the object can take. These actions are expressed in functions or methods that execute algorithms. Each object plays a specific role in the program design. The object interacts with other objects through messages or function calls. Attributes and behaviours, i.e. an object’s data and algorithms, can be *encapsulated*, or combined together to form a class.

A class is like a template or blueprint that can be used to construct or *instantiate* many similar objects. A class can also be seen as a sort of user-defined data type, which we call an abstract data type (ADT). An object is an instance of a particular class. A class hides information by restricting access to it. In effect the data and methods within an object are hidden from users to prevent inadvertent changes. Users of the class simply need to understand how to use the interface or interaction between the object and its methods.

Through *inheritance* it is possible to define subclasses that share some or all of the parent class’s attributes and methods, but with more specific features. This enables reuse of code.

Polymorphism describes the feature in languages that allows us to interpret the same word differently in different situations based on the context in which it is used. In object-oriented programs this means that similar objects can respond to the same function call or message in different ways.

Chapter 4: Procedural Abstraction and Functions That Return a Value

Savitch: Section 4.6 Overloading Function Names

4.1 Overview

In this chapter both predefined (library) functions and programmer-defined functions are discussed. This includes the use of value parameters, function invocation, procedural abstraction, and local and global scope. These concepts are covered in COS1511, but in section 4.6, overloading function names, which is not covered in COS1511, is explained.

4.2 Learning Objectives

After having worked through section 4.6, you should be able to:

- demonstrate that you understand the concept of overloading functions;
- write and use overloaded functions.

4.3 Notes

1. Study section 4.6.
2. Do the self-test exercises for section 4.6.
3. If necessary, work through the rest of the chapter to refresh your memory.

4.4 Different Notation and Conventions

Some of the notation and conventions used in Savitch may differ from those which were used in the COS1511 study guide. Both versions are valid. For example:

- Function definitions may be given either before or after the `main( )` function. However, if a function definition is given after the `main( )` function, the function prototype must be shown before the `main( )` function.
- Savitch uses the term *arguments* for the parameters listed in parentheses after the function name in a function call, while in COS1511 they were called *actual parameters*. See the explanation box Parameters and Arguments' in section 5.2 in chapter 5 in Savitch for an explanation of the different terms that have to do with parameters and arguments.

Chapter 5: Functions for All Subtasks

Savitch: Section 5.5 General Debugging Techniques

5.1 Overview

This chapter completes the description of C++ functions by discussing `void` functions and reference parameters. All of these concepts are covered in COS1511. It also discusses testing and debugging functions. In this regard we cover the `assert` macro in section 5.5.

5.2 Learning Objectives

After having worked through section 5.5, you should be able to:

- Use the `assert` macro to ensure that an expected condition is true at the location of the macro.

5.3 Notes

1. Study section 5.5.
2. Do the self-test exercises for section 5.5.
3. If necessary, work through the rest of the chapter to refresh your memory.

Chapter 6: I/O Streams as an Introduction to Objects and Classes

Savitch: Sections 6.1 to 6.3

6.1 Overview

This chapter explains how to accept input from a file and send output to a file by means of streams. In the process of explaining streams, the basic ideas of what objects are and how to use them in a program, is introduced.

The iostream header file is vitally important because it provides a consistent interface to all input/output in C++: screen, keyboard and file. Furthermore, all input/output devices can be accessed in C++ through streams (using operator overloading.) This input/output header file also provides access to a set of input/output routines that can be used to output all the primitive (i.e. built-in) C++ types, and can be overloaded by the user to output any user defined type. (Operator overloading is covered in Chapter 11.)

6.2 Learning Objectives

After having worked through sections 6.1 to 6.3, you should be able to:

- write programs that takes input from files (reading from files);
- write programs that send output to files (writing to files);
- format output with stream functions;
- manage character I/O correctly using the `get()` and `put()` member functions;
- use streams as arguments to functions;
- demonstrate that you understand the concepts of a *class*, an *object*, a *member function* among stream classes;
- declare an object of a class;
- use the dot operator to call a member function of an object.

6.3 Notes

1. Study sections 6.1 to 6.3.
2. Do the self-test exercises for sections 6.1 to 6.3.

6.4 File input and output

For programs that read their input from a file instead of the keyboard, we need to create the input file to hold our data before running the program. Any text editor (**not a word processor like MS Word, WordPad or WordPerfect!**) may be used. For example, we may use Notepad or the Code::Blocks editor to create the input file for Display 6.1 in Savitch as follows:

1. Open the editor.
2. Type in the input data in the order that the program will read it.
3. Save the file, giving it a meaningful name, e.g. **infile.dat**.

In Display 6.1 the input-file stream variable in_stream is declared with the statement

```
ifstream in_stream;
```

The statement:

```
in_stream.open("infile.dat");
```

connects the file **infile.dat** (created as described above) to the input-file stream variable by means of the `open` member function. If the input file and the program file have not been saved in the same directory, the full path name for the input file should be supplied. In this example **infile.dat** is the external file name, and `in_stream` the stream name used as the name of the file in the program.

Similarly, in Display 6.1 the output is written to the file **outfile.dat**. Output files are created by the program. The output file **outfile.dat** will therefore be created by the program. We can read the content of the file using a text editor, such as the DevC++ editor.

6.5 Classes and Objects

A class is a data type whose variables are objects, and an object is a variable that has member functions as well as the ability to hold data values in its data members (sometimes called member variables). Objects are the basic units of programming in object-oriented programming languages like C++. The attributes of an object are described through the values stored in its data members, while the member functions are used to execute the operations that can be performed on the object (its behaviour).

6.6 File Names as Input

Giving a file name as input is discussed in section 6.1 in Savitch. Note that the declaration for a variable to hold the file name

```
char file_name[16];
```

in the subsection 'File names as input' in section 6.1 declares a C-string variable, and not a variable of the standard `string` class. In COS1511 the standard `string` class was covered, and both types of string variables are covered in chapter 8 in Savitch.

One can also use a variable of the standard `string` class to hold the name of a file given as input, as shown below. Note that in this case when the name of the file is supplied as argument to the member function `open`, the `string` variable must first be converted to a C-string with the `c_str()`member function of the `string` class. The `string` member function `c_str()` converts a standard string to a character array.

```
string file_name;

ifstream in_stream;

cout << "Enter the name of the file to be processed: ";

cin >> file_name;

in_stream.open(file_name.c_str())
```

6.7 Header files

For keyboard/screen input/output you need to use `#include <iostream>`. Similarly, for file input/output you need to use `#include <fstream>`. The `fstream` class inherits from the `iostream` class, so most header files are arranged so that to `#include<fstream>` is sufficient to get the `iostream` header file included as well. Note, however, that this may not be the case for all compilers.

6.8 Processing data files versus processing text files

Files from which data should be read as `string`s, `int`s, `float`s or a mixture of data types, e.g. `int`s and `char`s, are called *data files*. Files which should be read and processed character by character, are called *text files*. The way in which we read data files differ from the way in which we read text files.

The typical way to read a data file, i.e. a file containing values that should be processed as `int`, `string` or `float`, is demonstrated below. Assume we have to process a data file called `infile` containing one `int` followed by a `string` on each line:

```
ifstream infile;
infile1.open("InputFile.dat");
if (!infile)
{
    cout << "Cannot open file "
         << "InputFile.dat" << " Aborting!" << endl;
    exit(1);
}

int value;
string name;
while (infile>> value >> name) //this checks for the eof marker of
                    // infile before reading the data from the file
{
     //process value and name
}
infile.close();
```

Processing the input file as a text file, i.e. a file consisting of characters which should be processed one by one, is demonstrated below:

```
ifstream infile;
infile1.open("InputFile.txt");
if (!infile)
{
    cout << "Cannot open file "
         << "InputFile.dat" << " Aborting!" << endl;
    exit(1);
}
```

```
     char ch;
     infile.get(ch);
     while (!infile.eof()) //explicit testing for eof of infile
     {
          //process ch
          infile.get(ch);
     }
     infile.close();
```

We can extract an object from a file of objects in the same way that we process a data file, e.g.:

Assume the following class definition:

```
class Student

{
     public:
          Student();
          string getName() const;
          int getMark() const;
     private:
          string name;
          string mark;
}
```

Then the following code fragment will process a file of objects of class `Student` `(InputFile.dat)`:

```
ifstream infile;
infile1.open("InputFile.dat");
if (!infile)
{
    cout << "Cannot open file "
         << "InputFile.dat" << " Aborting!" << endl;
    exit(1);
}

Student aStdt;
while (infile>> aStdt) //this checks for the eof marker
          // of infile before extracting aStdt (one object)
          //from the file
{
     //process object aStdt
}
infile.close();
```

Chapter 8: Strings and Vectors

Savitch: Sections 8.1 and 8.3

8.1 Overview

This chapter covers two topics that use arrays or are related to arrays: strings and vectors. Section 8.1 introduces C-string variables, which are inherited from C. C-strings represent a string as an array with base type `char`. Arrays have been covered in COS1511. Chapter 7 also covers arrays, and you may refer back to it if you need to refresh your memory. Section 8.2 discusses the standard `string` class, which was dealt with in COS1511. You will not be examined on section 8.2, but do refer to it if necessary. Section 8.3 introduces the container class `vector` in the Standard Template Library. We do not cover the use of iterators to access vector elements sequentially, though this is covered in Chapter 18 of Savitch.

8.2 Learning Objectives

After having worked through sections 8.1 and 8.3, as well as the subsection "Converting between `string` objects and C-strings", you should be able to:

- define and use C strings;
- define and manipulate a vector using the random access methods.

8.3 Notes

1. Study sections 8.1 and 8.3.
2. Do the self-test exercises for sections 8.1 and 8.3.

8.4 Converting between string objects and C strings

This section explains how to convert a string object to a C string by using the string member function `c_str()`. In section 6.6 of this Study Guide, we use the `c_str()` function to convert an external file name in `string` format to a C string, which is the format required by the stream member function `open()`.

8.5 The class `vector`

The class `vector` is defined in the Standard Template Library (STL) for C++. The STL vector container is a generalization of array. A `vector` is a container that is able to grow (and shrink) during program execution. A container is an object that can contain other objects. The STL vector container is implemented using a template class (templates are covered in Chapter 17).

A vector named `v` that can contain `int` values is declared with

```
      vector<int> v;
```

In order to use it, we need to include the correct header file that contains the declarations of the vector class and to make the names from the standard namespace available with:

```
#include <vector>
using namespace std;
```

A vector is a fixed-length group of elements of uniform type, indexed by integer keys. As mentioned before, vectors can be considered to be a generalization of the built in C++ array type. However, they have several other notable features not associated with the basic array. The main advantages of using vectors is that the size of a vector need not be fixed and can be changed during program execution, which is not possible with an array. Vectors are also important when the ability to rapidly access arbitrary elements is necessary, because, as with an array, individual elements in a vector can be directly indexed. The elements in a vector are assumed to be unsequenced, but the operation sort in the STL can be used to place the elements of a vector in order.

8.6 The Standard Template Library (STL)

The Standard Template Library (STL) standardises commonly used components through a set of containers (lists, vectors, sets, maps, etc.) that hold objects, built-in objects and class objects. The containers keep the objects secure and define a standard interface through which they can be manipulated. The STL also contains standard algorithms that can be applied to the objects while they are in the container, for instance, to sort them, compare them, merge objects into another container, find a specific object and many more operations. The STL algorithms use iterators, which are similar to pointers, to operate on the objects in the containers. The algorithms manipulate the objects in a standard way and the iterators ensure that the algorithms handle all and only the elements in the containers, and no more. You do not need to be able to use the iterator access methods for vectors, though more information on using iterators can be found in Chapter 18.

Chapter 9: Pointers and Dynamic Arrays

Savitch: Sections 9.1 and 9.2

9.1 Overview

This chapter introduces the concept of pointers and dynamic arrays.

9.2 Learning Objectives

After having worked through sections 9.1 and 9.2 you should be able to:

- define and manipulate basic pointer variables and dynamic variables;
- demonstrate that you understand the difference between automatic (ordinary) variables and dynamic variables;
- demonstrate that you understand the difference between static arrays and dynamic arrays.

9.3 Notes

1. Study sections 9.1 and 9.2 excluding the optional sections on Pointer Arithmetic and Multidimensional Dynamic Arrays.
2. Do the self-test exercises for sections 9.1 and 9.2.
3. We do not cover the subsections on Pointer Arithmetic and Multidimensional Dynamic Arrays.

Chapter 10: Defining Classes

Savitch: Sections 10.1 to 10.4

10.1 Overview

This chapter teaches you how to define your own classes. A class is a data type whose variables are objects. You have already used some pre-defined data types such as `int`, `char`, `string` and `ifstream`. The chapter first introduces structures as a first step toward defining classes. A structure (`struct`) can be viewed as kind of a simplified class without member functions.

10.2 Learning Objectives

After having worked through sections 10.1 to 10.4 you should be able to:

- define and use your own structures (`struct`s);
- define and use your own classes;
- understand all the terminology applicable to ADTs;
- create your own ADTs;
- write programs that use ADTs;
- demonstrate that you understand the concept of *inheritance*.

10.3 Notes

1. The terms *structure* and *struct* are often used as synonyms. The COS1511 study guide uses the term *struct,* while Savitch uses *structure* to refer to the same concept.
2. Savitch uses the term *class definition* for the declaration of a class. This is also known as an *interface* for the class or the *class specification*. Remember that a class is a data type, and the class definition specifies what attributes the data type have, i.e. which values it can hold, as well as its behaviour, i.e. what it can do. When we declare variables of a specific class, we say we instantiate objects of the class.
3. Study sections 10.1 to 10.4.
4. Do the self-test exercises on for sections 10.1 to 10.4.

10.4 Accessor functions

An accessor function is a member function that returns the value of a data member of the class. To ensure that accessor functions do not change data members by mistake, we mark them with the keyword `const.`

A `const` member function promises not to change any class data that it uses. A use of `const` in a class object declaration promises that the function won't call any other function with the potential to change that object's data members. The compiler tries to enforce these promises.

Use of the `const` keyword requires consistency. As shown in the `BankAccount` example below, if you declare a member function `const` in a class definition, you *must* use the `const` keyword in your definition of that member function. If you use a `const` object for an argument, and do not declare the function with a `const` parameter, you should get an error or a warning from your compiler.

To implement this in the class definition for the `BankAccount` class in Display 10.5:

(a) Change the function prototypes for the accessor functions `get_balance( )` in line 17 and

`get_rate( )` in line 19 to:

```
double get_balance( ) const;
double get_rate( ) const;
```

(b) Change the headings of the member function definitions for `get_balance( )` in line 85 and `get_rate( )` in line 90 to:

```
double BankAccount::get_balance( ) const
double BankAccount::get_rate( ) const
```

In this way the compiler will give an error message if any statement in the body of either `get_balance( )` or `get_rate( )` attempts to change a data member. The section on the *`const`* parameter modifier in section 11.1 in chapter 11 further explains this concept.

Chapter 11: Friends, Overloaded Operators, and Arrays in Classes

Savitch: Sections 11.1 to 11.4

Savitch: Appendix 7 The `this` pointer

Savitch: Appendix 8 Overloading Operators as Member Operators

11.1 Overview

This chapter continues teaching you how to define your own classes. It introduces friend functions, and teaches you how to overload common operators such as `+`, `–`, `*` and `/` so that they can be used in the same manner as with predefined types such as `int` and `double`. It also explains how to use classes of arrays, and arrays as data members of a class. Section 11.4 covers classes and dynamic arrays. This includes how to write destructors, copy constructors and how to overload the assignment operator.

11.2 Learning Objectives

After having worked through sections 11.1 to 11.4 you should be able to:

- define and use friend functions;
- define and use functions to overload both binary and unary operators;
- declare and manipulate arrays of classes as well as classes that contains arrays as data members;
- write destructors;
- write and use copy constructors;
- overload the assignment operator.

11.3 Notes

1. Study sections 11.1 to 11.4, Appendix 7 and Appendix 8.
2. Do the self-test exercises on for sections 11.1 to 11.4
3. Note the use of the `const` modifier as explained in section 11.1. We use this trick to prevent accessor functions from modifying data members.
4. Class definitions typically include a destructor.

11.4 Overloading operators as member functions

Appendix 8 explains how to overload operators as member functions. In Appendix 8, Savitch explains "When `+` is overloaded as a member operator, then in the expression `cost + tax`, the variable `cost` is the calling object and `tax` is the argument to `+`." This means that the compiler translates the expression `cost + tax` into the expression `cost.operator+(tax)`. This clearly shows that the function `operator+` has only one parameter, as is shown in Appendix 8 in Savitch.

The general form of functions to overload the binary operators as member functions of a class follows:

Function prototype (to be included in the definition of the class):

```
returnType operator#(const className&) const;
```

where `#` stands for the binary operator, arithmetic (e.g. +, -, *, /) or relational (e.g. ==, !=, >, >=, <, <=, &&, ||) to be overloaded; `returnType` is the type of value returned by the function; and `className` is the name of the class for which the operator is being overloaded.

Function definition:

```
returnType className::operator#(const className& otherObject) const
{
    //Algorithm to perform the overloading operation
    return value;
}
```

Compare this with the general syntax to overload binary operators as non-member (`friend`) functions:

Function prototype (to be included in the definition of the class):

```
friend returnType operator#(const className&, const className&) const;
```

where `#` stands for the binary operator, arithmetic or relational, to be overloaded; `returnType` is the type of value returned by the function; and `className` is the name of the class for which the operator is being overloaded.

Function definition:

```
returnType operator#(const className& firstObject,
                     const className& secondObject) const
{
    //Algorithm to perform the overloading operation
    return value;
}
```

11.5 The `this` pointer and overloading unary operators.

Appendix 7 covers the `this` pointer. As an additional example we show how the `this` pointer can be used in overloading the unary operators ++ and – in prefix position as member functions. (Note that overloading ++ and – in postfix position differs from what we show here!)

The process of overloading a unary operator is similar to that of overloading binary operators. The only difference is that in the case of binary operators, the operator has two operands. In the case of unary operators, the operator has only one operand. So, to overload a unary operator for a class, note the following:

1. If the operator function is a member of the class, it has no parameters.
2. If the operator function is a non-member of the class (i.e. a `friend` function), it has one parameter.

We use the class Money from chapter 11, and overload the pre-increment operator ++. We first show how to overload the pre-increment operator ++ as a `friend` function and then as a member function.

The general syntax to overload pre-increment operator ++ as a *non-member function* in prefix position can be described as follows:

Function prototype (to be included in the definition of the class):

```
friend className operator++(className& incObject);
```

Function definition:

```
className operator++(className& incObject)
{
     //increment object by 1
     return incObject;
}
```

Following this general syntax, we can overload the increment operator++ in prefix position as a non-member function as follows:

Function prototype (to be included in the definition of the class):

```
  friend Money operator++(Money & M);
```

Function definition:

```
    Money operator++(Money & M)
    {

    ++M.all_cents;
    return M;
    }
```

The general syntax to overload the pre-increment operator ++ as a *member function* in prefix position can be described as follows:

Function prototype (to be included in the definition of the class):

```
className operator++();
```

Function definition:

```
className className:: operator++()
{
     //increment object by 1
     return *this;
}
```

Following this general syntax, we can overload the increment operator++ in prefix position as a member function as follows:

Function prototype (to be included in the definition of the class):

```
Money operator++();
```

Function definition:

```
    Money Money::operator++()
    {
          ++all_cents;
          return*this;
    }
```

11.6 The Big Three: Destructor, Copy Constructor and `operator` =

In C++, three special functions are provided automatically for every class: destructor, copy constructor and `operator` =. In many cases the default behaviour provided by the compiler can be accepted. Sometimes, however, this cannot be done. This is typically the situation for classes with pointer data members. For these classes, three things are normally done:

1. Include the destructor in the class.
2. Overload the assignment operator (`operator` =) for the class.
3. Include the copy constructor.

Initially, we will show the destructor, assignment operator and copy constructor for a simple class, `IntCell` that simulates a memory cell containing an integer value. The class declaration and implementation for class `IntCell` is shown below. (Note that we use separate compilation in this example. If you have not studied separate compilation yet, just ignore all the pre-processor directives starting with `#`.)

Specification (IntCell.h):

```
#ifndef INTCELL_H
#define INTCELL_H

class IntCell
{
    public:
    IntCell (int Value = 0); //constructor
    int get_Value( ) const; //accessor
    void set_Value(int x);  //mutator

    private:
    int Value;
};

#endif
```

Implementation (IntCell.cpp):

```
#include "IntCell.h"

IntCell::IntCell(int InitialValue): Value (InitialValue)
{
}

int IntCell::get_Value( )const
{
    return Value;
}

void IntCell::set_Value(int x)
{
    Value = x;
}
```

11.6.1 Destructor

A destructor is a member function that is called *automatically* when an object of the class goes out of scope at the end of a function, or is deleted explicitly by a `delete` statement. Destructors can do any “housekeeping” necessary, such as to delete all dynamic variables created by the object. Just as the compiler creates a default constructor automatically if no constructor has been specified for the class, the compiler creates a destructor automatically if one is not included in the class definition. A class has only one destructor with no arguments. The name of the destructor is distinguished from the no-parameter constructor by the tilde symbol ~.

To include a destructor for the class `IntCell`, we add `~IntCell` as a member function to the class definition (specification). The implementation for the destructor function `~IntCell`, is as follows:

```
~IntCell ( )
{    //nothing to be done since IntCell contains only an int data
     // member.If IntCell contained any class objects as member
     // variables their destructors would be called
};
```

11.6.2 Copy Constructor

The copy constructor is a special constructor that is required to construct a new object, initialised to a copy of the same type of object. A copy constructor therefore has one (usually `const`) call by reference parameter that is of the same type as the class.

For any object, such as `IntCell`, a copy constructor is called

- When an object is declared and initialised by using the value of another object, such as

  ```
  IntCell B = C;
  ```

  ```
  IntCell B(C);
  ```

but not

```
    B = C; //Assignment operator
```

- When, as a parameter, an object is passed using call by value (instead of by `&` or `const &`)
- When the return value of a function is an object

In the first case the constructed objects are explicitly requested. In the second and third cases temporary objects are constructed. Even so, a construction is a construction and in both cases we are copying an object into a newly created object.

By default the copy constructor is implemented by applying copy constructors to each data member in turn. For data members that are primitive types (e.g. `int`, `double` or pointers), simple assignment is done. This is also the case for the data member `Value` in class `IntCell.` For data members that are themselves class objects, the copy constructor for each data member's class is applied to that data member.

The prototype for the default copy constructor for `IntCell` is

```
     IntCell(const IntCell & rhs); //copy constructor
```

with the following implementation:

```
IntCell::IntCell(const IntCell & rhs)
{
     Value = rhs.Value;
}
```

The `operator` =

The copy assignment operator, `operator` =, is used to copy objects. It is called when = is applied to two objects after both have been previously constructed. The expression `lhs = rhs` is intended to copy the state (i.e. the values of the data members) of `rhs` into `lhs`. By default the `operator` = is implemented by applying `operator` = to each data member in turn.

The prototype for the default assignment operator for `IntCell` is

```
     const IntCell & operator=(const IntCell & rhs);
```

with the following implementation:

```
const IntCell & operator=(const IntCell & rhs)
{
      if (this != &rhs) //standard alias test to avoid self-assignment
          Value = rhs.Value;
      return *this;
}
```

As mentioned in the last paragraph of chapter 11 in Savitch, to make sure that the object is not copied to itself, an alias test is done with

```
      if (this != &rhs) //standard alias test to avoid self-assignment
```

An additional keyword in C++, the pointer `this` points at the current object. We can think of the pointer `this` as a homing device that, at any instant in time, tells us where we are. (Also see

Appendix 7 in Savitch for more on `this`). The pointer `this` thus refers to the address of the current object, while the unary address operator `&` in combination with `rhs`, as in `&rhs`, refers to the address in memory where `rhs` is stored. If these two addresses are the same, i.e (`this == &rhs)`, the same object appears on both sides of the assignment operator and the operation should not be executed.

If the object is not copying to itself, the `operator=` is applied to each data member. `IntCell` has only one data member `Value`, so only one statement is needed:

```
    Value = rhs.Value;
```

`*this` is the current object, and returning `*this` will return a reference to the current object. This allows assignments to be chained, as in `a = b = c`.

11.6.3 Problems with the defaults

If a class consists of data members that are exclusively primitive types (such as `int`, `double`, `vector<int>`, `string`, etc.) or objects for which the defaults make sense, the defaults can be accepted. However, when a class contains a data member that is a pointer, the defaults are no longer sufficient.

Suppose that the class contains a single data member that is a pointer. This pointer points at a dynamically allocated object. If we adapt the specification for class `IntCell` so that its single member is a pointer to an integer, the specification and implementation changes to what is shown below. Note that the default destructor, copy constructor and assignment operator will be used since we do not define our own:

Specification (IntCell.h):

```
#ifndef INTCELL_H
#define INTCELL_H

class IntCell
{
    public:
    IntCell (int Value = 0); //constructor
    int get_Value( ) const; //accessor
    void set_Value(int x);  //mutator

    private:
    int *Value;
};

#endif
```

Implementation (IntCell.cpp):

```
#include "IntCell.h"

IntCell::IntCell(int InitialValue)
{
   Value = new int(InitialValue);
}

int IntCell::get_Value( )const
{
    return *Value;
}

void IntCell::set_Value(int x)
{
    *Value = x;
}
```

The output from the application below in which class `IntCell` is used, exposes problems with accepting the default destructor, copy constructor and assignment operator:

```
#include <iostream>
#include "IntCell.h"
using namespace std;

int main() {
   IntCell a(2);
   IntCell b = a;
   IntCell c;

   c = b;
   a.set_Value(4);
   cout << a.get_Value() << endl << b.get_Value() << endl
        << c.get_Value()
        << endl;
   return 0;
}
```

Output:

```
4
4
4
Press any key to continue . . .
```

The output produces three `4`s, even though logically only `a` should be `4`. The problem is that the default copy constructor and the default `operator=` both copy not the objects being pointed at, but simply the value of the pointer `Value`. Thus `a.Value`, `b.Value` and `c.Value` all point at the same `int` value. This is called shallow copying: i.e. the *pointer* rather than the *contents of the address to which the pointer is pointing*, is copied. Typically, we would expect a deep copy, in which a clone of the entire object is made.

A second less obvious problem is a memory leak: the `int` initially allocated by `a`'s constructor remains allocated and needs to be reclaimed. The `int` allocated by `c`'s constructor is no longer referenced by any pointer variable. It also needs to be reclaimed, but we no longer have a pointer to it. This is because the default destructor does nothing. We have to call `delete` ourselves. To fix these problems, we implement the Big Three, as shown below:

Specification (IntCell.h):

```
#ifndef INTCELL_H
#define INTCELL_H

class IntCell
{
    public:
    IntCell (int Value = 0); //constructor

    IntCell (const IntCell & rhs); //copy constructor
    ~IntCell( );             //destructor
    const  IntCell  &  operator=(const  IntCell  &  rhs);  //overloaded
                                                         //assignment

    int get_Value( ) const; //accessor
    void set_Value(int x);  //mutator

    private:
    int *Value;
};

#endif
```

Implementation (IntCell.cpp):

```
#include "IntCell.h"

IntCell::IntCell(int InitialValue) //constructor
{
    Value = new int(InitialValue);
}

IntCell::IntCell (const IntCell & rhs) //copy constructor
{
    Value = new int (*rhs.Value);
}

IntCell::~IntCell( )            //destructor
{
    delete Value;
}


const IntCell & IntCell::operator=(const IntCell & rhs) //overloaded
                                                   //assignment
{
    if (this != &rhs)   //standard alias test to avoid self-assignment
        *Value = *rhs.Value;
    return *this;
}

int IntCell::get_Value( )const    //accessor
{
    return *Value;
}

void IntCell::set_Value(int x)  //mutator
{
    *Value = x;
}
```

Running the same application as before, now gives the correct output, as shown below:

Output:

```
4
2
2
Press any key to continue . . .
```

To summarize, when a class has pointer members and a constructor allocates memory from the free store you need a copy constructor to guarantee correct initialization of an object of that class type from another object of that class type. And you also need a destructor to guarantee that free store memory pointed to by members are deallocated when the class object goes out of scope. If you assign objects of such a class to one another, then you need to overload the operator = to guarantee that the data pointed to by objects' pointers are copied, and not just the pointers.

Note that defining `operator =` has no effect on the behavior of the class objects during copy construction. Similarly, defining a copy constructor does nothing for the assignment operator. `Operator =` is invoked when assignment is made to a `class` object. The destructor is called when an automatic object goes out of scope, and when the destructor is called explicitly.

Chapter 12: Separate Compilation and Namespaces

Savitch: Sections 12.1 and the first part of section 12.2 ‘Creating a namespace’

12.1 Overview

This chapter teaches you how to divide an ADT into separate specification and implementation files. It also discusses how to implement separate compilation as well as conditional compilation using pre-processor directives. This chapter also discusses namespaces. Creating your own namespaces is beyond the scope of this module, but you do have to understand the concept of a namespace.

12.2 Learning Objectives

After having worked through section 12.1 and the first part of section 12.2 you should be able to:

- divide an ADT into separate specification and implementation files;
- understand the concept of a namespace.

12.3 Notes

1. Study section 12.1 and the first part of section 12.2.
2. Study the sections on *Include Directives and Namespaces* in section 2.2 in chapter 2, and *Namespaces Revisited* in section 4.5 in chapter 4. In this course we will only use the standard namespace `std`.
3. Do the self-test exercises following section 12.1.

12.4 Separate Compilation

C++ allows you to divide a program into parts. Each part can be stored in a separate file and can be compiled separately and then linked together when (or just before) the program is run. Any software project which is slightly more than trivial is generally divided into separate files.

Programs that use user-defined classes usually also use multiple source files. Typically the definition (specification) of the class will be placed in a **.h** file, called the interface or specification file, while the implementation will be placed in a **.cpp** file. This offers, among others, benefits such as compiling the files separately and software reuse. For example, the interface may be required (with `#include`) on multiple occasions within one or more projects’ code, but the implementation need only be compiled once.

To use separate compilation, you create a project to which the files to be included can be added. See Appendix A in this tutorial letter for instructions on how to do this in Code::Blocks

12.5 Preprocessing and conditional compilation

Preproccessing occurs before a program is compiled. Possible actions include:

- inclusion of other files in the file being compiled (by means of file inclusion directives),
- definition of symbolic constants and macros,
- conditional compilation of program code and
- conditional execution of preprocessor directives.

All preprocessor directives begin with `#`, and only whitespace characters may appear before a preprocessor directive on a line. Preprocessor directives are not C++ statements, so they do not end in a semicolon (`;`).

To prevent multiple declarations of a class, we use the pre-processor directives `#ifndef`, `#define` and `#endif` to control the inclusion of the header file so that the same definitions are not included repeatedly. This is called ‘conditional compilation’.

The `#include` preprocessor directive causes a copy of a specified file to be included in place of the directive. The two forms of the file inclusion directive (`#include <filename>` and `#include "filename"`) indicate where (which directory) the preprocessor should find the file to be included.

To include a predefined header file use `#include <filename>`, e.g.

```
   #include <iostream>
```

The `<` and `>` characters tells the compiler to look where the system stores predefined header files.

To include a header file you wrote, use `#include "filename"`, e.g.

```
   #include "dtime.h".
```

The `"` and `"` characters usually cause the compiler to look in the current directory for the header file. If the filename is given in terms of an absolute file system pathname, then the file is taken from that absolute location. For example, the following directive indicates that the file `sample.h` can be found in the directory `\example\source` on the `C:` drive.

```
   #include "C:\example\source\sample.h"
```

Once the preprocessing has been done, the compiler translates the source code into object code, stored in a `.o` file. The linker then links the object code with the code from libraries for functions that are included (with `#include`) to create the executable code, stored in a `.exe` file.

Chapter 14: Recursion

Savitch: Sections 14.1 to 14.2

14.1 Overview

This chapter introduces recursion. Recursion is a difficult topic to grasp. However, it's very easy to apply once you understand it. Recursion means allowing a function or procedure to *call itself*. It keeps calling itself until some limit is reached.

14.2 Learning Objectives

After having worked through sections 14.1 to 14.2 you should be able to:

- understand the concept of recursion;
- write recursive functions according to a recursive definition for a problem.

14.3 Notes

1. Study sections 14.1 and 14.2.
2. Do the self-test exercises following sections 14.1 and 14.2.

14.4 Recursion

The summation function, designated by an uppercase ∑ (Sigma) in mathematics, is a popular example of recursion:

$$\sum_{i=1}^{n} i = \sum_{i=1}^{n-1} i + n$$

or, put in a different way:

$$\begin{aligned} \text{sum}(n) &= 1 && \text{if } n = 1 \\ &= \text{sum}(n-1) + n && \text{if } n > 1 \end{aligned}$$

Written as a recursive function, this gives:

```
int calc_sum (int num)
{
 int sum;
 if (num == 1)       //base case
          sum = 1;
     else
          sum = calc_sum(num-1) + num;  //recursive call
     return sum
};
```

Suppose you call `calc_sum` for 3:

```
int answer = calc_sum(3);
```

`calc_sum (3)` becomes `calc_sum (2) + 3`.

`calc_sum (2)` becomes `calc_sum (1) + 2`.

At `1`, the recursion stops and becomes `1`.

`calc_sum (2)` becomes `1 + 2 = 3`.

`calc_sum (3)` becomes `3 + 3 = 6`.

`answer` becomes `6`.

Recursion works backward until a given point is reached at which an answer is defined (the base case), and then works forward with that definition, solving the other definitions which rely upon that one.

All recursive procedures/functions should have some sort of test to stop the recursion. Under one condition, called the base condition or the base case, the recursion should stop. Under all other conditions, the recursion should go deeper. In the example above, the base case is `if (num == 1)`. If you don't build in a base condition, the recursion will either not take place at all, or become infinite.

Chapter 15: Inheritance

Savitch: Section 15.1

15.1 Overview

This chapter introduces the concept of inheritance, a key feature of object-oriented programming. Inheritance is the ability to define new classes from existing classes. The new classes that we create from the existing classes are called derived classes; the existing classes derived are called the base classes. The derived classes inherit the properties of the base classes. So rather than create completely new classes from scratch, we take advantage of inheritance and reduce software complexity. We only cover section 15.1: *Inheritance Basics*. Polymorphism is not covered in this module.

15.2 Learning Objectives

After having worked through section 15.1 you should be able to:

- distinguish between 'is-a' relationships and 'has-a' relationships;
- understand the role of public inheritance in class design;
- create new classes by *inheriting* from existing classes;
- understand how inheritance promotes *software reusability*;
- understand the notions of *base classes* and *derived classes*;
- understand the effect of the access-specifier labels (i.e private, public and protected) in terms of public inheritance.
- define constructors for a derived class;
- redefine member functions;
- demonstrate that you understand the difference between redefining a member function in a class and overloading a member function.

15.3 Notes

1. Study section 15.1.
2. You can safely ignore all references to namespaces other than `std`. In Display 15.1, for example, lines 9 and 28 can be omitted, and the same applies to the other displays in this chapter.
3. Do the self-test exercises for section 15.1.

15.3 Inheritance

15.3.1 The Purpose of Inheritance

The primary purpose of inheritance is to reuse code by exploiting the 'is-a' relationship between objects. Software reusability saves time in program development. It encourages the reuse of proven and debugged high-quality software, thus reducing problems after a system becomes functional. Significant overlap between two classes indicates a potential case for inheritance.

15.3.2 The ‘is-a’ relationship

In the real world, objects exist in relation to one another. In solving problems we need to be as close to the problem as we can, within abstraction that allows us to ignore details in order to solve our problem. In our problem, we may describe one object as being like another, even saying one object ‘is-a’ type of some other object.

In an example of ‘is-a’ relationships, we might think of a class of (general) vehicles, and then think of a kind of vehicle for carrying passenger (automobiles), another kind that is small and carries cargo in the open (pickup truck), still another that carries cargo in an enclosure (van), others yet that carry cargo enclosed and are articulated - tractor-trailer). The Truck, Van, and Car classes inherit their common features from the class Vehicle.

In these cases, a car is a vehicle, so are pickups, vans, and tractor-trailers. A car ‘is a’ vehicle; a pickup truck ‘is a’ vehicle, and so on. The class abstraction of vehicle is the base class and the vehicles that bear the ‘is a’ relation to the base class are derived classes. In this example, the base class contains the features common to all types of vehicles. The derived class inherits all those common features and adds its own, extra, additional features to the base class to form its distinctiveness. In this way, the common features (commonalities) are encapsulated in the base class, and the distinctions are encapsulated in the derived class. The commonality is passed from the base class to the derived class with inheritance. A derived class is therefore more specific than its base class. This relationship is known as the ‘is-a’ relationship. The ‘is-a’ relationship specifies that one abstraction is a specialization of another. If we write this example in C++, we have

```
class Vehicle {/* . . . */};
class Car : public Vehicle { /* . . . */};
class Truck : public Vehicle { /* . . . */};
class TractorTrailer: public Truck {/* . . . */};
class StationWagon : public Car {/* . . . */ };
```

We see that inheritance creates a new class, called the derived class, that is an extension, specialization or modification of one or more existing classes, called the base classes. In the situation where there is only one base class, we have single inheritance. Where there is more than one base class, multiple inheritance occurs. In this, example, we could illustrate multiple inheritance by defining

```
class Emergency {/* . . . */};
```

and have had a class `PoliceCar` inherit from both class `Car` and class `Emergency`. Or we could have a class `Ambulance` inherit from both class `Truck` and class `Emergency`. The `PoliceCar` and Ambulance would inherit features from Car and the derived class adds its distinguishing features. Since multiple inheritance is beyond the scope of this module, we will not discuss it further.

Notice that the ‘is-a’ relationship is transitive (that is, a `TractorTrailer` is a `Truck`, a `Truck` is a `Vehicle`, and therefore a `TractorTrailer` is a `Vehicle`), but it is not reflexive (that is, not all `Vehicle`s are `TractorTrailer`s). Another way of expressing the ‘is-a’ relationship is to say that a `TractorTrailer` is a *kind* of `Truck` and that an `Truck` is a kind of `Vehicle`. See also the text box in section 15.1 in Savitch on ‘An Object of a Derived Class Has More Than One Type’.

In addition to the ‘is-a’ relationship that represents inheritance, two other relationships between abstractions are commonly used on object-oriented design, namely the ‘has-a’ and ‘uses-a’ relationships. The ‘has-a’ relationship says that some object is part of another. The ‘has-a’ relationship implies containment. For example, a car has an engine. The ‘uses-a’ relationship says that one object uses another object in some way. This usually realized by one object communicating with another via member functions. For example, suppose the operating system has a clock object that maintains the current date and time. The clock object has member functions that return the current date and time. Other objects that need the date or time, use the clock object by calling the appropriate member functions to fetch the current date or time.

15.4 Public Inheritance

Public inheritance should be used when a new class (the derived class) describes some set of objects that is a subset of the objects being described by the base class. This relationship is known as the ‘is-a’ relationship.

So what does *public inheritance* mean? Consider the classes `Base` and `Derived`:

```
class Base
{
  public:
    int x;
  protected:
    int y;
  private:
    int z;
};

class Derived : public Base
{
  public:
    int total( ) const
    {
        int sum = x;
        sum += y;
        //sum += z; //cannot access z directly
        sum += q;
        return sum;
    }
  private:
    int q;
};
```

Public members of the `Base` class are public members of the `Derived` class too. The `Derived` class inherits all data members and all member functions from the `Base` class, but it can access only those that are not private. That is, a derived class can access public and protected members of its base class. In the example, class `Derived` can access `x` and `y` directly but not `z`, because `z` is private in class `Base`. If we want to access `z` which is a private data member in the `Base` class, then we can access `z` in one of two ways:

(1) by creating a public / protected accessor in class Base, such as

```
int get_z( ) const
```

```
        {
            return z;
        }
```

(2) or we could make `z` protected (more about this in section 15.1 in Savitch).

Now consider the following client program that instantiates objects of type `Base` and `Derived`:

```
int main( )
{
    Base b;
    Derived d;
    cout << b.x;
    cout << d.x;
    //cout << b.y;
    //cout << d.y;
    //cout << b.z;
    //cout << d.z;
}
```

Since `x` is public in class `Base`, it is also public in class `Derived` through `public` inheritance. Therefore we can access `x` in the client program. But we cannot access `y` because it is a protected member of both `Base` and `Derived`. And of course we cannot access `z` because it is private.

Note, that the access specifier for class inheritance defaults to `private`. If class `Derived` has base class `Base`:

```
class Base {/* . . .*/};
class Derived :  access-specifier Base
{ /* . . . */};
//where access-specifier is one of public, protected, private
```

then the access that is granted to member functions of class `Derived` depends on the access-specifier selected. If the access-specifier is *omitted*, access to all members of the base class automatically *defaults to private*. This would mean that `Derived` has *no access* to the member functions and data members of `Base.`

Chapter 17: Templates

Savitch: Sections 17.1 to 17.2

17.1 Overview

This chapter discusses C++ templates. C++ templates provide a way to reuse code by defining functions and classes that have parameters for type names. Templates are very useful when implementing generic constructs like vectors, stacks, lists and queues, which can be used with any arbitrary type.

17.2 Learning Objectives

After having worked through sections 17.1 to 17.2 you should be able to:

- implement function templates;
- implement class templates;
- compare and contrast function overloading with the use of templates.**Notes**

1. Study sections 17.1 and 17.2.
2. Once again, you can safely ignore all references to namespaces other than `std`, such as in lines 12 and 49 of Display 17.4.
3. The MinGW compiler used in Code::Blocks does not allow separate compilation of templates. Therefore you need to include the template definition (i.e. its declaration and the implementation thereof) in the same (header) file. Note that you can place the template definition in a separate file and use a `#include` directive to include the template definition when you need to use it.
4. Note that the use of the keyword `class` in a template prefix does *not* mean that you can write only a class name there. You can use any type name already defined in the language such as, `int`, `float`, `double`, or any identifier that is a user defined type (such as a class, structure or enum).
5. Do the self-test exercises for sections 17.1 and 17.2.