Yubing Hou

General Computer Science using C♯

with hundreds of examples and solutions

Contents

Preface

Compared with other mature science subjects, such as physics and chemistry, computer science is still in its childhood stage. Young sciences always brings issues in teaching since not many people have taught these subjects before. Since the first bachelor's degree in computer sciece was awarded, computer science have witnessed huge leap in human technology. However, while the technology is rapidly changing, the teaching of computer science, espeically introductory computer science, is still in a mist. What contents are taught, and how they are taught, vary by schools dramatically. A student who took CS 101 in a univeristy in Midwest may not even understand a word of the first few lectures of CS 102 in univeristies of West Coast. The outcome is, all students will receive computer science degree in the end, but their knowledge are not necessarily equivalent, even fragmented, since school A might put effort on cultivating student's programming skills thoroughly, while school B emphasizes more on the mathematical part of computer science.

That is the main motivation of writing this book: to offer a relatively comprehensive and flexible textbook for CS 101 courses. Generally speaking, most CS 101 courses in the United States cover these topics: imperative programming and object-oriented programming. Other than these, schools usually customize their own curriculums. Some of them may introduces more about object-oriented programming and software engineering, while others may directly jump into some elementary data structure and algorithms. All these curriculum designs have their rationale behind. Therefore, it can be concluded that a comprehensive but elementary textbook for CS 101 should be able to satisify most schools' needs, and all students have the same opportunity to learn the most of it.

This book is designed with the aim to solve issues mentioned ahead. This book starts with basics concepts of type system and elementary discrete mathematics. After a brief introduction to the C# programming language, this book jumps into the basics of imperative programming. Several elementary data structures, including array, list, queue, and stack are introduced before methods (functions) are covered. The last part of this book introduces the object-oriented programming paradigm and some fundamental I/O. The structure of this book was fundamentally following the CS 302 curriculum of the University of Wisconsin-Madison, where I took my introductory computer science.

Besides of the curriculum design, this book is published online through GitBook. This idea is from the inspiration of a pair of professors of Madison as well: Professor Andrea and Remzi Arpaci-Dusseau, who wrote a great undergraduate operating system textbook: Operating Systems: Three Easy Pieces and released it online as free book for everyone. Though I have never took a course from either of them, I strongly agree with their philosophy about free textbook, especially when the computer science is changing rapidly and textbook has became a burden for most students.

Finally, I would like to thank to people who have supported me writing this book and making this book possible. This book will be constantly updated as technology outdates rapidly. I wish every reader could enjoy the fun that computer science bring you as I did.

To Instructors This book is desinged with flexibility. A one-semester course should be able to finish the basic idea of file I/O. The last chapter, software testing, is optional for teaching and should be left to students to figure out on their own time.

Besides of the two major programing paradigms, this book also touches a little bit of many other related fields of computer science, especially data structure and algorithm. However, instrutors should feel free to skip them if they are not required by the course or under time constraint.

To Students

Yubing Hou

February 1st, 2015

Chapter 1

Introduction to Computing

1.1 Chapter Introduction

What is computer? What is computer science? What is it about? Generally, people would agree with this definition: computer science is the study of computation and its applications. However, like many other subjects, computer science also covers a wide range of topics. Typically, computer science studies problems like these:

- How to build a faster computer? (Computer architecture)
- How to design a robust operating system? (Operating systems)
- How to sort a set of poker cards quickly? (Algorithm)
- How to tell what user is searching for? (Artificial intelligence)
- How to find the shortest path between two cities? (Graph Theory)

As you can tell, computer science is studying almost every corner of our daily life. When you make a video call to your friend on your smartphone, it involves the study of network, operating system, graphics, and computer architecture. It is fair to say that without computer science, we will not have our digital life today.

The definition of computer can be a little bit tricky. Technically, a computer is a machine that can perform computation, such as a digital calculator, cell phones, personal computer, cloud storage server, etc. They do not have to be electrical either: some ancient mechanical calculators are also considered as computer. In a technical definition, a computer is a machine that can perform computation.

Theoretically, a computer is an abstract machine: they do not have to be tangible at all. A theoretical computer is the prototype of those technical computers. Compared with a technical computer, a theoretical one is more

Topic: Computer science and Programming People tend to consider computer science and programming as the same thing. In fact, this is not the case. Computer science studies the process of computation and the application of computation, while programming is mostly about implementing that computation and its applications.

For example, the design of an operating system is usually the job of a computer scientist while the implementation of that design is the work of a programmer. A computer scientist, like Bill Gates or Linus Torvalds, comes up with an idea about how the system should be designed and run, and a group of programmers will write the corresponding code that matches the requirements of the design. If we call the design of an operating system as the "theoretical computer science", then the implementation of that design is the "experimental computer science".

Programming is really a small subset of computer science. Computer scientists generally have some programming skills, while programmers' understanding of computer science varies a lot.

1.2 History of Computing and Computer

It is very important for us to learn the history of the subject that we are going to learn. Learning history helps us understanding where we come from and why things are organized in the way we see today. Computer itself does not have very long history, but the history of computing can dates back to thousands of years ago, when human first learned writing.

1.2.1 Ancient History of Computer Science

The history of electrical computer (not even electronic) can date back to 1940s when ENIAC was first invented in MIT, but the history of programming and many other components of computer science date far earlier than that. In ancient Greece, Euclid proposed a group of axioms and theorems for some basic geometry and developed some strategy for solving basic geometry problems. This is one of the earliest use of algorithm.

1.2.2 Modern History of Computer Science

From 1940s to 1960s, technology did not evolve as rapidly as today. However, hardware had been constantly improving, but a few problems had emerged already: with the improvement of hardware, programming them became more and more challenging and difficult. This led to the "software crisis", which was first brought up in 1968.

However, two significant events changed the entire histroy from 1970s until today: the birth of Unix operating system in 1969 and the C programming language. The Unix operating system was first implemented by Ken Thompson and Dennis Ritchie, who were working at the Bell Lab of AT&T. Unix was the first operating system that realized multi-user, multi-programming, and file system on computer. The design philosophy of Unix operating system has influenced generations of computer scientists and operating systems. Linux, OS X, Android, and Chrome OS are all Unix-like systems. It is fair to say: without Unix, we would probably still in 1960s.

The C programming language is another milestone in technology history. C was first implemented by Dennis Ritchie and Brian Kernighan. In 1972, Unix was rewritten in C, thus became the first operating system than can be run on multiple platforms. This significantly changed the world of programming and software engineering. From then, writing programs on computer has been significantly easier.

In 1991, a Finnish college student, Linus Torvalds, published the very first version of Linux operating system, which later became the foundation of Chrome OS, Android, and many other embedded systems.

Programs and Software Computer science is the study of computation and its applications. What is computation? What are its applications? One of the most important applications of computation is program. A program is a set of instructions that tells computer to perform some specific work.

1.3 Basic Computer Architecture

In order to talk with computers, we need to use programming langauges.

1.3.1 Hardware

The hardware of a computer are usually the tangible parts of it, such as its monitor, keyboard, mouse, speaker, camera, processor, memory chips, etc. The essential part of modern computer consists of five basic components: processor, memory, input, output, and storage. The study of hardware components mostly fall into the study of electronics and electornic engineering. In these field, people study how to build effective and efficient computers using those fundamental physics laws.

1.3.2 Software

The intangible part of a computer, such as the operating system, the word processor, the music player, and web browser are software. They are programs that uses the resources given by hardware to perform computing works.

1.3.3 Operating System

Operating system is a special software that is responsible for allocating hardware resources among a variety of software: how much memory can an application use, which file that an application can modify, which application to run in the next 10 ms, etc. All these detailed work are done by the operating system instead of other application.

Today, the most three common operating systems that are heavily used are Microsoft Windows, Mac OS X, and GNU/Linux.

1.3.4 User Programs

User programs are software that mostly for user's special needs, such as word processor, multimedia player, graphical design, etc. These applications must be run on top of a specific operating system.

1.4 Binary Representation

Type System Suppose you have 4 light bulbs are connected in this fashion. As you can imagine, you can turn all these light bulbs on or off, or turn on some of them while leave the others off. Each light bulb only has two states: on and off. We represent the "on" state as 1 and "off" state as 0. Therefore, for 4 light bulbs, the combination of their states make a group of new states. Let's label them with alphabetic letters like this:

- 0000 'A'
- 0001 'B'
- 0010 'C'
- 0011 'D'
- 0100 'E'
- 0101 'F'
- 0110 'G'
- 0111 'H'
- 1000 'I'
- 1001 'J'
- 1010 'K'
- 1011 'L'
- 1100 'M'
- 1101 'N'
- 1110 'O'

- 1111 ‘P’

As you can tell, with only 4 light bulbs, you can make a combination of 16 states, which is 2^4 states. If you don’t like using them to represent letters, you can use them to represent numbers, such as:

- 0000 0
- 0001 1
- 0010 2
- 0011 3
- 0100 4
- 0101 5
- 0110 6
- 0111 7
- 1000 8
- 1001 9
- 1010 A, which is 10
- 1011 B, which is 11
- 1100 C, which is 12
- 1101 D, which is 13
- 1110 E, which is 14
- 1111 F, which is 15

Does it look familiar to you? If not, this is the hexadecimal representation of numbers.

You can label these states to make them mean something. For example, one way to label them as alphabetic letters. In this case, we can label 16 English letters from A to P. If you have more light bultbs, say 5, which allows you to get $2^5 = 32$ states. This is definitely enough for you to represent the entire alphabetic letters just by using on and off of those light bulbs. Yes, it will be a little bit tedious for you to translate the states of light bulbs into letters, but we can make a machine that do this translation for us.

Imagine that if you have more lightbulbs, following the induction above, you can label $2^6 4$ different states, which is more than $1.8 * 10^1 9$! This is a huge number! With 64 light bulbs, you can not only use them to represent alphabetical letters, but also many characters and letters in other languages, even a large range of numbers!

The more light bulbs you have, the more information you can represent with these light bulbs. Each light bulb has only two states. In computer science, we call this "binary" representation of information, and each light bulb is a "bit".

1.4.1 Primitive Type

With so many values that a computer can represent, it is better to categorize them so that manipulate them can be easier and faster. There are many way that you can categorize them, but one way in particular that computer scientists use is called type system. Type system is basically categorizing all kinds of values into following categories:

- Integer
- Decimal-point number
- Character
- Logical values

Types in the list above are generally refered as *primitive types*. They are "primitive" not because they are primitive, but because their variables directly contains them.

1.4.2 Reference Type

Besides of primitive type values, computer can also represent another type called *reference type*. Reference is not a particular kind of type but a general name of a variety of types. They are called "reference type" because their memory location does no contain values it has, but a memory address that contains those values. In another word, when you open the box that stores a reference type value, you will get a little piece of paper telling you which box that the actual values are in. This may sounds a little bit weird to you at this stage, but you will understand why reference type is very important in programming when you understand some basic computer architecture and operating systems.

A typical reference type in programming is string. A string is a sequence of characters. For example, "This is a great day!" is a string that contains 20 characters. As you may guess, a string can contain arbitrary number of [characters][1]. Since the size of a string can change all the time, it is better to store it somewhere else instead of within a single box of memory space.

[1]: Strictly speaking, the number of characters in a string is limited by the size of computer's avaiable memory.

1.5 Octal Representation

1.6 Hexadecimal Representation

1.7 Chapter Exercise

1. TBA

Chapter 2

Variable and Type System

2.1 Introduction

This chapter introduces the concept of variable and basic type system. Type system is a way of categorizing data that we use inside a computer. By putting data into different kind of categories, programming can be much easier and efficient. By allowing data of differnt type be cast to others, programming can be more flexible and creative, too. This chapter starts from some basic idea of computer memory structure, then jump into variables and types. Types introduced in this chapter are mostly primitive types (integer, double, and character).

2.2 Variables

When a program is running, there is a lot of data manipulation going on in the processor and memory of it. While some data are constant that remain the same value while programming is running, others are changable, and they are usually refered as 'variable'. In computer, a variable is a memory location that is associated with a symbolic name.

An analogy in math is function. For a linear function such as:

$$f(x) = x + 1$$

In the expression above, x is considered as a variable: you can assign any numerical value to it, and that expression will just evaluate what ever you assign to x. Of course, you can also assign the result of the function above to another variable, say y, and the expression will look like this:

$$y = x + 1$$

If we have a variable x and the valued stored in it is 20, then we will get 21 if we plug x in the expression above. If we have another variable y, we can assign 21 to it and y will have value 21.

However, you should also notice that you can only plug in a numerical value to x. It does not make any sense to evaluate the sum of “banana” and 1. This is a very important lesson for us: sometimes a function can only accept certain kind of input, and that kind is usually called “type” in computer science. In this case, our function above can only accept numerical inputs. In another word, x must be a number.

In programming, variables can be created by programmer or program, depends on the job that the programm wants to perform. Variables must be *declared* before they can be *referred* or *assigned*. The *declaration* of a variable is the creation of that variable: programmer ask the operating system to allocate a chunk of memory to store some value. The *reference* of a variable is to obtain or use the value stored in that variable. The *assignment* of variable is to assign a new value to a particular variable. Remember, you must declare the variable first before you can either refer to it or assign value to it.

2.2.1 Variable Declaration in C♯

C♯ is a statically-typed programming langauge, meaning that every variable you declared in C♯ programming language must have a type.

2.2.2 Variable Naming

When your program is running, the program accesses variables that you allocated by using their memory addresses instead of those symbolic names that you put in code. You may wonder: if the program does not necessarily need the name of variable, why do we care about the naming of it?

The truth is, your program is not only just for solving a programming problem. If you are working with a large programming team, your code is very likely to be read by tens, if not hunderds or thousands, of people, and people need to understand what your code is doing even when you left the team. Having good naming of variables make your code self-documented, thus brings good code and maintainable program.

Compare the following pairs of naming:

- An integer variable for age: age and x
- A string variable for user input: input and s
- A double variable for balance of savings account: savingBalance and b1

By comparing the naming choices above, you probably have noticed that good naming choice for variables can be very helpful in programming.

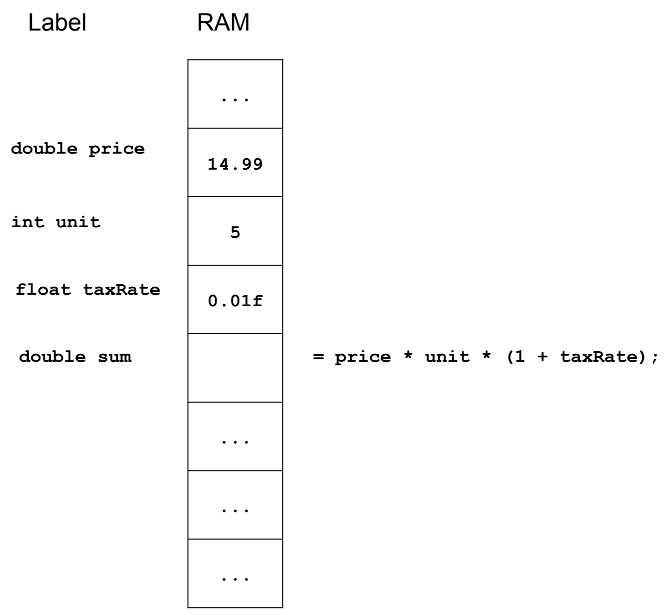
2.2.3 Primitive Type and Reference Type

Besides that values have all kinds of types, the storage of those values has type as well. There are two commonly used storage type in today's programming language: primitive type and reference type.

A primitive type (or variable type) is type of storage that the value of the variable is directly stored at where it is allocated. Imagine that you have a group of boxes, items are stored right instead of these boxes. For example, if we write a program like this:

```
using System;
public class Program
{
    public static void Main (string [] args)
    {
        double price = 14.99;
        int unit = 5;
        float taxRate = 0.01f;
        double sum = price * unit * (1 + taxRate);
        Console.WriteLine ("Unit price: " + price);
        Console.WriteLine ("Purchases: " + unit);
        Console.WriteLine ("Tax Rate: " + taxRate);
        Console.WriteLine ("Total Cost: " + sum);
    }
}
```

In this case, variable `price`, `unit`, `taxRate`, `sum` are primitive type and the value are stored in the memory box that they reside in. If we draw a memory diagram of those variables that we declared, it would look like this:



In this case,

Reference type (also called as *pointer type*) is a little bit different. Inside of directly putting items inside the box, reference type will put the address of the item inside the box. When you need that value, you open the box first, and notice that there is only the address of values that you want. Then you follow the address and find the value.

Why do we need both of them? Think about this way: each memory address has limited capacity for storing values. However, sometimes there are values that are so huge that would take many many addresses of spaces to store. In this case, it would be unrealistic to squeeze all those values in one little address. We need to split them and use a pointer to find out the first item of those values. Based on the type of that pointer, we can also easily find the other values of the variable.

Another advantage of using reference type is that using reference type can save time on memory reallocation. If you have a chunk of data of 3 gigabyte in memory, it would be time wasting to move this much data around the memory avery time we use it. However, with reference type, we can simply use a different reference to achive the 'movement' of data.

2.3 Integer

Integers are values that does not have fractional components. Examples of integers are: 0, -99, 10201022, etc. For any computer, the amount of integer that it can represent is limited. For a 64-bit computer, the maximum integer it can represent is 2¡sup¿64¡/sup¿ - 1 (minus 1 because it starts from 0 instead of 1). b

2.4 Real Number

In computer, real numbers are numbers that are not integers. This definition does not really match the definition of real number in mathematics. Another important difference between the definition of real number in computer science and in mathematics is that: real numbers in computer are discrete, while real numbers in mathematics are continuous. This is mostly because computer only has a limited precision, and most real numbers, such as 1/3 or π, has infinite number of digits after the decimal point. In computer, only the first few digits can be represented. For example, in Python, a very popular programming language, the default value of π is 3.141592653589793.

Depending on the precision, there are typically two types of real number in computer: `float` and `double`. For the C♯ programming language, there is another type for real number: `decimal`.

2.5 Character

2.6 Logical Values

2.7 Void

There is a special type in many programming languages, and this type is `void`. Technically, a `void` type is not a type, since it is impossible to declare a variable that has a type `void` in C♯ . However, `void` is still very important, especially in languages such as C and C++, where a pointer of `void` type is heavily used in programming.

So far, the only situation that we would use `void` type is in declaring the `Main ()` method:

```
using System;
public class Program
{
    public static void Main (string [] args)
    {
        // Put statements here.
    }
}
```

2.8 Type Cast

Sometimes we need to convert a value of one type to a value of another type. For example, we need to know what is the 127th character in the ASCII character table, or we need to find out the index of character ‘c’. In this case, if there is a mechanism that can allow us to

2.9 String

2.10 Implicit Type in C$\sharp$

Sometimes we want to use the same variable name for multiple types.

2.11 Expression and Statements

Definition: An expression

2.12 Summary

2.13 Exercise

1. Given a function $f(x) = x^2 + 22$, what type of value do you think is suitable for x? Is 5 a good value for x? Is 9.4 a good value for x? Is 1.41421... (the square root of 2) a good value?
2. Given a function $f(x) = x$, what type of value do you think is suitable for x? Is the word “popular” a good value to plug in to this function? Why or why not? Do you think the type of variable depends on what the function is doing in this question? Why or why not?
3. List all the differences between an integer-typed variable and a double-typed variable, and how they are represented in a computer.
4. List all the differences between an character-typed variable and an integer-typed variable, and how they are represented in a computer.
5. Which of the following values are suitable for an integer-typed variable? “word”, ‘T’, 99.38, -121, “5”, ‘5’.
6. What is type-casting? When do you need it?
7. Explain how to cast an integer value to a character value, using ASCII table.
8. Explain how to cast a character value to an integer value, using ASCII table.
9. Explain what will happen when casting 16.77 to an integer value.

10. Explain what will happen when casting 99 to a floating-point value.

11. What is a variable? How many type(s) can a variable have in a program?

12. Given a line code below: double potatoPrice = 6.44; Can you tell what the type, name, and value of this variable is?

13. What is a declaration of variable? How to declare a variable? Assume there is a type called F̈ruit,̈ declare a variable called m̈yFruitöf type F̈ruit.̈

14. What is the assignment of variable? How to assign a value to a variable?

15. Assume that there is a type called d̈ouble.̈ Write a line of code that declares a variable whose name is ẗemperatureänd whose type is d̈ouble,̈ and assigns 35.44 to this variable.

16. Assume that you have three integer variables: *price*, *quantity*, and *cost*, write some code that declares these three variable, and assign 20 to *price*, 5 to *quantity*, and the product of *price* and *quantity* to *cost*. Try to declare and assign them in as many different order as possible, what do you find? Does your code work in all these situations? Why or why not? Does the sequence of declaration ans assignment matter?

Chapter 3

Introduction to the C♯ Programming Language

3.1 Introduction

This chapter introduces the C♯ programming language: a langauge that is mainly designed and implemented by Microsoft Corporation.

3.2 Pseudocode

Computer needs very specific and clear instructions in order to run correctly. Computer itself can only recognize one kind of instruction, which is the machine code that consists of 0s and 1s. Since it is very difficult for human beings to read binary code, we designed assembly language.

3.3 Programming Languages

Though we have designed and implemented thousands of programming languages, there is no such thing called "perfect programming language" yet. This is because all programming languages are designed to achieve a certain goal or address a solution to a particular issue. Therefore, all programming languages have its strength and weakness. If you are going to major in computer science or be a professional programmer, it is very likely that you will master 5 or 6 programming languages in order to perform your work. By that time, you will learn what language is suitable for performing what kind of tasks.

This book uses a relatively popular programming language: C♯ . C♯ is designed and mainly implemented by Microsoft Corp. In a long time of its life, it is considered as a improved version of Java, which is a programming language that appeared before C♯ .

3.3.1 The History of Java Java is a programming language designed and implemented by James Gosling and his colleagues when they were working for Sun Microsystem Co. in 1995. Unlike many other programming languages at that time, Java was designed to run on multiple platforms. Today, Java applications almost run on every device you use: your laptop PC, your Android smartphone, the Web, etc..

How to Write a C♯ Program Everything must follow a set of rule in order to work correctly, especially computer programs. In fact, programs are so sensitive that even a tiny little errors can cause huge problems. When you write your programs from this book, you must copy exactly the words, character to your text editor and compile it with the correct command.

3.3.2 Your First C♯ Program

Let's get started by writing a simple C♯ program: print out some text on your screen. Suppose we want to print out "Hello, world!" on your screen, and the corresponding program will be like this:

```
using System;
public class Program
{
    public static void Main (string [] args)
    {
        Console.WriteLine("Hello, world!");
    }
}
```

Summary: How to write, compile, and run a program

1. Write down your program in a text editor, following good programming style and convention.
2. Read through your program, check and see if there is any obvious syntax error. Improve your program when necessary.
3. Compile your program. If no error messages printed, that's great! If there are errors coming out, read the error message, and see if you can fix the error.
4. Once it compiles, then there will be an executable program for you to run! Now just run the program, and see if it works in the way you designed.

3.4 Programming Paradigm

In real life, problems generally have more than one solutions, so do programming. In programming, these solutions are categorized into different paradigms. Each paradigm is a perspective of solving a problem. Today, there are three major programming paradigms: imperative programming, object-oriented programming, and functional programming. This book mostly focus on the first two programming paradigms.

Imperative programming is one of the oldest programming paradigm. It views solving a problem as giving instructions to computer. In another word, solving a problem is executing a group of procedures, thus this programming is sometimes called procedural programming. Typical programming languages that supports this programming paradigms are: C, FORTRAN, Go, and Pascal.

Object-oriented programming uses a different view. It considers a problem is made of many *objects*, and there is interaction among them. Thus a programming problem turns into how to make these objects interact with each other. In this programming paradigm, everything, especially data, is viewed as some sort of *object*.

C♯ is a programming language that can supports many different programming paradigms: imperative, object-oriented, and a little bit functional programming.

3.5 Console I/O

Programs are written for human use, thus they must be able to communicate with the outside environment while running. Such communication involves a mechanism called I/O (stands for "input and output"). Programs generally use different form of I/O to communicate with the outside world. When you use your word processor to write paper for your history classes, your word processor give you a graphical user interface (GUI), and what you see on the screen, such as menu and the content you typed in are the "output" of the program. When you type your paper into the word processor, the word processor will receive inputs from your keyboard. I/O is a very important aspect of modern computer.

Besides of the graphical user interface you see on your screen, there are other forms of I/O that does not have a graphical user interface but simply based on text. The only way that you can interact with these kind of programs is typing commands or inputs and program will read what you typed and perform certain jobs according to your input. This is called command-line user interface (CLI). In this book, all programs that you are going to write are in CLI. Most CLI programs run in console (or terminal). If you are using Microsoft's Windows operating system, it will be run in a command line environment. If you are using Apple's Mac OS X or a distribution of Linux, it will be run in your terminal.

3.5.1 Output: Print Text in Terminal

The very first thing we want to do is to print some messages on the screen. Let's make this program print out a sentence: "No legacy is so rich as honesty" from Willam Shakespeare.

First of all, let's start with our template:

```
using System;
public class MyProgram
{
    public static void Main (string [] args)
    {
    }
}
```

Let's name this program as `ioprac01.cs`, which stands for "I/O Practice No.1".

Compile this program and run it, and you immediately notice that this program does not do anything! This is because we have not tell what computer needs to print out yet. In order to tell computer to print out something, we need to use a *method*, `Console.WriteLine(string value)`, to print out the information that we want.

First of all, let's declare a string variable so that we can store the content somewhere in the memory:

```
using System;
public class MyProgram
{
    public static void Main (string [] args)
    {
        string message = "No legacy is so rich as honesty.";
    }
}
```

Now we use the method we mentioned: `Console.WriteLine()` to print out this message in console/terminal:

```
using System;
public class MyProgram
{
    public static void Main (string [] args)
    {
        string message = "No legacy is so rich as honesty.";
        Console.WriteLine (message);
    }
}
```

Compile this program and run it, you will see the output like this:

```
No legacy is so right as honesty.
```

Now you know that everytime you want to print out some message, simply use the `Console.WriteLine(string value)` method!

3.6 Program Structure: Basics

Every C♯ program must follow a certain structure. In this book, most of the programs that you write are executable, meaning that after compiling them to lower level code they can be run by your computer directly. For all these executable programs, they are all in this form:

```
public class <class identifier>
{
    public static void Main (string [] args)
    {
    }
}
```

If we have a program whose name is “Shopping”, you should replace `<class identifier>` by the program name. This program should be written in this form:

```
public class Shopping
{
    public static void Main (string [] args)
    {
    }
}
```

A few things must be memorized:

1. All program should start from `public class`, followed by a pair of curly braces.
2. All executable programs have a *method*called `Main (string [] args)`. This method should also be `public static void Main (string [] args)`.

When you write your program on a text editor, you should follow these steps:

Step 1. Write down libraries and the class and its braces:

```
using System;
public class MyProgram
{
}
```

Step 2. Write the `Main (string [] args)` between the curly braces. Also, notice that `Main (string [] args)` has its own pair of curly braces, nested in the curly braces of the class.

```
using System;
public class MyProgram
{
    public static void Main (string [] args)
    {
    }
}
```

This is the code skeleton that you are going to use for the first few chapters in this book.

3.7 Programming Convention

Like many other discipline such as mathematics, physics, and chemistry, computer science has her standards of notation and programming conventions. Keeping good and consistent programming style makes your code easier for your fellows to read and maintain, just a like a uniform standard notation in chemical elements makes the communication among chemists easier. Unlike other sciences that only has one single convention, there many programming conventions and styles for you to choose. *However, it is very important that you pick one style and convention and stay with it as long as you can.* This section will introduce some fundamental conventions in programming and code writing.

3.7.1 The 80-Column Rule

3.7.2 Indentation

Indentation helps you grouping your code into blocks so that you can easily tell which statements are executed together. There is a debate about the side of indentation: some organizations, like the Linux kernel development team, prefers 8-character indentation, while some programming langauges, such as Scala, set the indentation as 2-character long. They both have their rationales. For C♯, I recommend to use either 4 or 8-character indentation.

There is one thing that people commonly agree with: you should never have too many indentations in your code, and the Linux kernel development team even states that a program shall never have more than three levels of indentation. The reason behind this is: high level indentation usually comes from nested conditional statements, such as nested `if-else`, `for` loop, and `while` loop, and such nested logical control can be greatly simplified by using some external parameters or functions.

3.8 Summary

We introduce the basic I/O mechanism in the C♯ programming language. To print out message in terminal, use methods:

- `Console.WriteLine (string value)`
- `Console.Write (string value)`

To read input from user, use methods:

- `Console.ReadLine ()`
- `Console.Read ()`

Reading inputs means that new values are added into program while program is running, thus it is important to remember saving the input to a `string` variable:

```
string inputmsg = Console.ReadLine ();
```

3.9 Chapter Exercise

1. TBA

Chapter 4

Program Flow Control

4.1 Introduction

So far, you have learned how to write programs that read some inputs from user and print some information out. However, it is rather primitive if we want to make a bit more complex program. For example, if we want to write a program that can read a number from user input and tell if that number is even or odd, we won't be able to do that yet. Another example would be doing something repetitive, such as printing out all odd numbers that are between 1 and 1000. It would be very tedious for us to enumerate all odd numbers in this range and print them out individually. Thus we need some mechanism that can allow us to handle such job.

Flow control is one of the earliest programming mechanism introduced. In a procedure-oriented programming paradigm, having flow control adds more flexibility and complexity to programs so that programs could handle complex jobs as well. The most common flow control techniques that are used in today's programming are conditional statements and loops.

This chapter will start off with introduction to conditional statements, including `if`, `else if`, and `else` statements. Three loops will be introduced as well: `for`, `while`, and `do while` loop. At the end of this chapter, two loop-control statements will be introduced: `break` and `continue`. These statements are commonly used in all C-family programming languages. Besides of them, C♯ also has her own special `foreach` loop control, which will be left for instructor's decision.

4.2 If Statement

Sometime we only want to execute parts of our program under certain conditions. For example, we want to read a number from user input, and we want to print it out if it is an even number and do nothing if the input is an odd number. Therefore, we would want the print statement to execute only when the input number is even. Therefore, we would need some mechanism in our program such that the program can test if certain condition is met. This mechanism is the *if-statement*.

In C♯, an if-statement code block looks like this:

```
if (<condition>)
{
    <statement #1>;
    <statement #2>;
    ...
}
```

Evaluating the expression `<condition>` should return a Boolean value: `true` or `false`. If the returned value is `true`, statements inside the if-statement code block, such as `<statement #1>` and `<statement #2>`, will be executed. For the example mentioned at the beginning of this section, the if-statement code block should be like this:

```
int input = int.Parse (Console.ReadLine());
if (input % 2 == 0)
{
    Console.WriteLine ("It is an even number!");
}
```

```
start
read movie data
if (movie produced before 1950):
    then print ("It's an old movie!:)
end
```

Many programming languages offer a statement that allows program to choose an execution path based on a condition. This is the if statement.

The '<condition >' expression should always evaluate to a boolean value: true or false.

4.3 Else Statement

4.4 Else-if Statement

4.5 For Loop

4.6 While Loop

Sometimes we do not exactly know when to jump out of a loop, but we know that we should jump out a loop when certain condition(s) is satisfied. In this case, using a for-loop will be a little bit clumpsy and infleximble. For example, Bob and Jim want to play basketball after school, but Jim wants to finish his homework before that. Therefore, for Bob, the waiting process will be like this:

while (Jim is doing home work) wait for 5 more minutes

This will make Bob keep waiting for Jim until Jim finishes his homework, and Bob does not need to know how long it will take Jim to finish.

On contrast, if we use a for-loop, things will be a little bit different:

start = 4'o clock end = 5'o clock for (time between start and end): wait

This will make Bob waiting for one hour for sure, but what if Jim cannot finish his homework in an hour, or what if Jim finishes his homework in 30 minutes? In the former case, should Bob force Jim to play basketball with him regardless of the unfinished homework? In the later case, should Bob keep wait until that hour

Usually, there are two kinds of while loop in C-family programming languages: 1. while loop 2. do-while loop

They work in a very similar way, except they check conditions at different checkpoints.

4.7 Break Statement

4.8 Continue Statement

4.9 Flowchart

Flowchart is a very useful skill that can help programmers understand what they need to achieve in their code. Flowchart is not part of programming, but can be very helpful. When you are reading other programmers' code and trying to understand what their programs are doing, drawing a flowchart can often reduce the time you spend on reading the code.

4.9.1 Use Flowchart to Write Program

4.9.2 Use Flowchart to Read Program

4.10 Program Construction: Reverse Engineering

It is not unusual that newcomer programmers sometimes do not know how to start writing a particular program. This section will show one of the basic way of solving a programming program: reverse engeineering.

Example: You are required to write a program that reads two integers from user, calculate the quotient and remainders of them, and print it out in a certain format. The required programming langauge is C♯.

Let's take a look at these requirements and imagine what it should look like if we have had this program (bold letters are user inputs):

```
Enter numerator: <b>25</b>
Enter denominator: <b>4</b>
25 / 4 = 6 with 1
```

As you can tell, this program should be able to print out some prompts, read two integers, and print out the results of calculation.

Let's see how to use reverse engineering to solve this problem:

Firstly, this program prints out some prompts, which means 'Console.Write()' must have been used in this program.

Secondly, this program reads some inputs and interprets them as integers, thus `Console.Read()` and `int.Parse()` must have been used.

Thirdly, if we run this program multiple times, the results of calculation is in the same format all the time, thus formated string must have been used.

After the analysis above, we are sure that following C♯ methods should be used in this program:

- `Console.Write()` and `Console.WriteLine()`
- `Console.ReadLine()` and `int.Parse()`
- `string.Format()`

Besides of these, since `Console.Read()` will return the input from user, and quotient calculation needs integer variables, thus there must be some string and integer variables storing these values.

Therefore, we need variables like these:

- `string input`

- `int numerator`
- `int denominator`
- `int quotient`
- `int remainder`
- `string output`

With these specified, we can start writing our programs:
Step 1: Write down code skeleton:

```
public class MyProgram
{
    public static void Main (string [] args)
    {
    }
}
```

Step 2: Declare and initialize variables:

```
public class MyProgram
{
    public static void Main (string [] args)
    {
        string input = "";
        string output = "";
        int numerator = 0;
        int denominator = 0;
        int quotient = 0;
        int remainder = 0;
    }
}
```

Step 3: Write code that prints the output information (prompts and format strings):

```
public class MyProgram
{
    public static void Main (string [] args)
    {
        string input = "";
        string output = "";
        int numerator = 0;
        int denominator = 0;
        int quotient = 0;
        int remainder = 0;

        Console.Write ("Enter numerator: ");

        Console.Write ("Enter denominator: ");

        Console.Write (output);
    }
}
```

Step 4: Write code that reads input from users and convert input string into integers:

```
public class MyProgram
{
    public static void Main (string [] args)
    {
        string input = "";
        string output = "";
        int numerator = 0;
        int denominator = 0;
        int quotient = 0;
        int remainder = 0;

        Console.Write ("Enter numerator: ");
        input = Console.ReadLine ();
        numerator = int.Parse(input)

        Console.Write ("Enter denominator: ");
        input = Console.ReadLine ();
        denominator = int.Parse(input);

        Console.Write (output);
    }
}
```

Step 5: Perform the calculation and assign the results to proper variables:

```
public class MyProgram
{
        public static void Main (string [] args)
        {
            string input = "";
            string output = "";
            int numerator = 0;
            int denominator = 0;
            int quotient = 0;
            int remainder = 0;

            Console.Write ("Enter numerator: ");
            input = Console.ReadLine ();
            numerator = int.Parse(input)

            Console.Write ("Enter denominator: ");
            input = Console.ReadLine ();
            denominator = int.Parse(input);

            quotient = numerator / denominator;
            remainder = numerator % denominator;

            Console.Write (output);
        }
    }
```

Step 6: Produce the format string before print it out:

```
public class MyProgram
{
    public static void Main (string [] args)
    {
        string input = "";
        string output = "";
        int numerator = 0;
        int denominator = 0;
        int quotient = 0;
        int remainder = 0;

        Console.Write ("Enter numerator: ");
        input = Console.ReadLine ();
        numerator = int.Parse(input)
        Console.Write ("Enter denominator: ");
        input = Console.ReadLine ();
        denominator = int.Parse(input);

        quotient = numerator / denominator;
        remainder = numerator % denominator;

        output = string.Format ("{0} / {1} = {2} with {3}",
                                numerator,
                                denominator,
                                quotient,
                                remainder);

        Console.Write (output);
    }
}
```

Now your program is finished and ready to be compiled and run!

4.11 Program Construction: Divide and Conquer

Having methods in program not only can remove some tedious coding, but also improve the reliability and debugability of the program. For small and trivial programs, there is usually no strong reason to have methods for them. However, as problem gets complicated, there is need for having methods. Tackling such problems demands another important skill in programming: divide and conquer.

The philosophy behinde "divide and conquer" is that: all complex and complicated problems are made of a lot of small but repetitive problems. If these small problems can be solved, it won't be too hard to solve the entire problems. As an analogy, cooking a sophiscated dish can be very complex. However, if you divide the "cooking dish" into many small processes, such as preparing ingredients, making spice, and heat control, then this "cooking" process is instantly simpler and approachable.

Now, let's use divide and conquer to solve a problem.

4.12 Summary

This chapter mainly covers the technique of program flow control.

4.13 Chapter Exercise

1. TBA

Chapter 5

Elementary Data Structure

5.1 Introduction

Data structure is one of the most important topic in computer science. Many innovation of computer software comes from utilizing efficient data structure in computer programs. New data structures are usually the key that brings leaps to the development of computer science. In many areas of computer science, finding and efficient data structure usually can solve half of the probem of a project. Data structure is simply reorganizing data for efficient use.

Rob Pike, one of the greatest C programmer, propose that "Data dominates. If you've chosen the right data structures Tnd organize things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming." His quote later became one of the Unix philosophy in Eric Raymond's *The Art of UNIX Programming*.

This chapter will introduce following basic data structures: structure, enumeration, array, list, queue, and stack.

5.2 Structure

Struct, as known as structure, is a composition of data. For example, all laptops have brand and price. If we have 5 kinds of laptop, then we would need to have 5 string variables from their names and 5 integer variables for their prices. This can be difficult to manage. However, with struct, we can easily represent such data relationship in a program.

In C♯, a struct is declared like this:

```
public struct Laptop
{
    public string brand;
    public int price;
}
```

A struct declaration usually contains these statements:

```
public struct <identifier>
{
    public <type> <member identifier>;
    public <type> <member identifier>;
    ...
}
```

If we have a laptop of which brand is “Boat” and price is 500, we can use the structure above to represent such laptop like this:

```
Laptop boat; // declare this variable
boat.brand = "Boat"; // Initialize its brand
boat.price = 500; // Initialize its price
```

As you can tell, if you want to access the member of a structure, just use the dot operator (“.”) after the variable identifier, and followed by the variable identifier of the field. For example:

```
Console.WriteLine (boat.brand + ": " + boat.price);
```

5.3 Enumeration

Upcoming.

5.4 Array

Array is a data structure that allows a certain number of values that have the same type stored and accessed together. Array is one of the most useful and important data structure in computer science, as it is used to implement other more sophisticated data structures such as list, heap, and set. This book does not introduce the implementation of any of these data structures other than how to use them.

Array is a reference type, meaning that the variable name in a program is simply a reference that points to somewhere else in computer memory that actually stores the value of each value. In an array, each value is called as an *element*.

Generally, array is 0 indexed, meaning that the first item for us is the zeroth item for computer.

5.5 Multidimensional Array

Multidimensional array is essentially array of arrays.

To declare a two dimensional array using C♯ , use following syntax:

```
type [] a = new type [size of array];
```

5.6 List

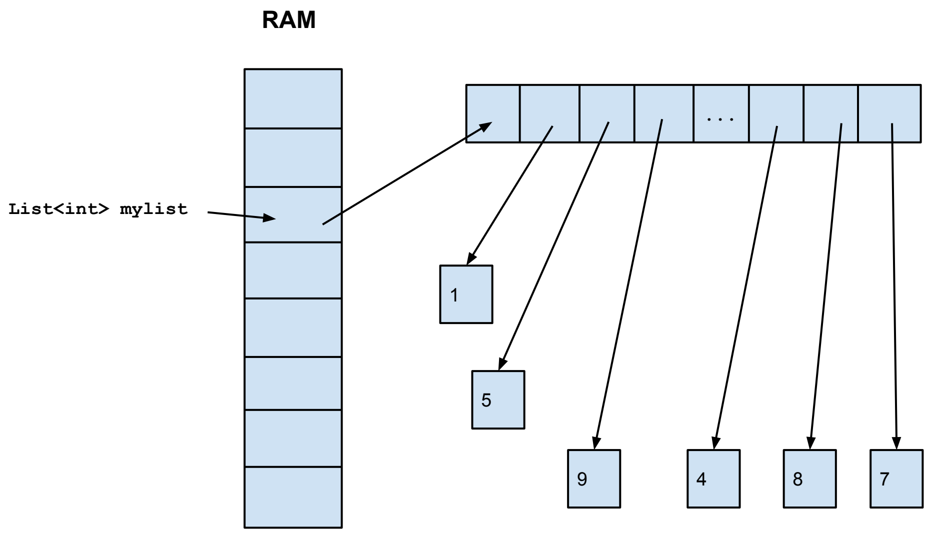
List is a kind of data structure that can handle data whose size changes while program is running. For example, you want to write a program that can manage your shopping list. When you go shopping, you are adding and removing items to and from this list, and you are not very likely to know what items and how many of them you are going to buy. A list offers a solution to this kind of problem.

You can imagine that a list is an array with changable size.

A list can perform following operations:

- Add an item: `Add`
- Remove an item from list: `Remove`
- Find if an item is in list: `Contains`
- Get the index of an item that is in list: `Find`
- Insert an item at a specific location in list: `Insert`

Here is how a list could look like in the memory of a computer:



5.7 Queue

Upcoming

5.8 Stack

Upcoming.

5.9 Generics

5.10 Summary

5.11 Exercise

1. tba

Chapter 6

Method

6.1 Introduction

Modern program consists of thousands even millions line of code. The Linux operating system kernel consists of more than 10 millions line of code, and most of the code are regular routines: things that needs to be done repetitively. *Method (or function in other programming langauges)* supplies one of the most important ways that can reduce the complexity of code. When a series of tasks need to be performed repetitively, it is better to write a method that can handle this job, and everytime the program just need to invoke a method when the routine work is needed.

The most common method that we have used so far in C♯ is the `Main ()` method. The `Main ()` method is where a program starts its execution. Let's take a look at the characteristics of the `Main ()` method.

6.1.1 The Main Method

All of our programs so far are written like this:

```
using System;
public class Program
{
    public static void Main (string [] args)
    {
        ...
    }
}
```

As you can tell, there are quite a few things associated with the `Main()` method. They are:

1. It is `public`
2. It is `static`
3. It has a *type* `void`
4. It has an *argument*, which is an array of `string`

Therefore, we can infer that a method in C♯ should have the keywords `public` and `static`. A method should have a *type* as well. Optionally, a method should have some *arguments*, depending on what the method needs to do.

6.2 Arguments

6.3 Returned Value

6.4 Recursion

Upcoming

6.4.1 Stack Overflow

You probably have heard of this. There is a famous programming Q& A website called StackOverflow, but what actually it is?

Let's take a look at the following code. Do you think it can be compiled? If yes, what do you think it will happen when you run it? Try write down this code and run it on your computer:

```
using System;

public class Program
{
    public static void Main (string [] args)
    {
        MethodA();
    }

    public static void MethodA ()
    {
        Console.WriteLine ("Method A is being called!");
        MethodB();
    }
```

```
        public static void MethodB ()
        {
            Console.WriteLine ("Method B is being called!");
            MethodA();
        }
    }
```

Yes, it will be compiled. The C♯ compiler will not complain anything. However, when you run it, something weird happens: it seems like the program is just printing `Method A is being called!` and `Method B is being called!` infintely, but if you wait for a while, it start printing something like `at Program.MethodA () <0x0001b>` and `at Program.MethodB () <0x0001b>` infintely. Finally, if you wait long enough, the last two lines printed out are something like this: `at Program.Main (string[]) <0x0000b>` and `at (wrapper runtime-invoke) <Module>.runtime_invoke_void_object (object,intptr,intptr,intptr) <0xffffffff>`.

We won't explain what they mean here: you will learn their meanings when you take a course in computer architecture and operating system. But we will explain what it tells you.

When your program is running, there is a chunk of memory, called *stack*, that is keep tracking of the method calls in your program. Everytime a method is called, some data is added to the stack. In our example above, when program starts, the `Main()` method is called, and some data is added to the stack. Then the program execute method `MethodA()`, thus some more data is added to stack. Since `MethodB()` is called in `MethodA()`, more data is added to stack. Then you will notice: `MethodB()` will call `MethodA()`, and in turn `MethodA()` will call `MethodB()`. This will just keep adding a lot of data to the stack.

However, the size of the stack is limited: it can only contain certain amount of data. When there is too much data on the stack, the stack will overflow. When stack is overflown, it will remove all the data in it, print them out, and terminat the program. This is called *stack overflow*. In this case, stack overflow is caused by recursion: recusively calling `MethodA()` and `MethodB()`.

6.5 Chapter Exercise

1. Upcoming

Chapter 7

Elementary Algorithm

7.1 Introduction

Computer science is not simply about writing a program that can solve a particular program. In fact, quite a large portion of computer science is about optimization. How to do things more efficiently? In which situation do we need to prefer one solution over another? How to measure the efficiency of a particular solution? To answer these question, we need to study some basic algorithm. Algorithm is all about how to solve some problem efficiently or averagely efficiently. This chapter will introduce some basic algorithms for two typical problems: sorting and searching.

Sorting is a kind of problem that we want to rearrage disordered items to put them in incremental order. There are many great sorting algorithms that have been developed so far, including quicksort, merge sort, and heap sort. In this chapter, we will study two kinds of sort: insertion sort and merge sort.

Searching is a kind of problem that we want to find the first occurence of a pattern in a larger environment. Typically, both pattern and environment are strings. There are two kinds of searching algorithms that are commonly used today: Boyer-Moore's algorithm and Kruth's algorithm. However, both of them are rather complex for an introductory level text. Therefore, this chatper will only introduce the naive algorithm and we will see why it is inefficient in most cases.

7.2 Sorting

Many problem can be solved easily once items are sorted. For example, finding the card Heart J is much easier in a sorted set of cards than unsorted. It is likely that you may get lucky with unsorted cards since the first card is Heart J, but most of the time you need to look up every single card in an unsorted set of cards to find out which one is Heart J. However, for a sorted set of cards, all you need to find out is where Heart cards are, and use that to find out where Heart J is.

There are a variety of items that can be sorted: numbers, alphabetic letters, poker cards, etc. For a particular item, there are many attibute to sort by. For example, a group of people can be sorted by their heights, last name, first name, or the address they live in. For a particular sort, such as sorting people by their heights, there are also a variety ways of sort them. Computer science studies the efficient way of sorting a particular thing to a particular order. This section introduces two basic sorting algorithm: insertion sort and selection sort. However, there are more efficient sorting algorithm out there, including heap sort, merge sort, and quick sort, and you will learn them from a more advanced level course.

7.2.1 Insertion Sort

7.2.2 Selection Sort

7.2.3 Sorting in C♯

7.3 Searching

7.4 Chapter Exercise

1. Do you think the naive algorithm of sorting efficient? Why or why not? Look up some other sorting algorithms and see if you can find any other algorithm that are efficient?

2. Is it easier to find items in a sorted collection or unsorted? Why or why not?

3. How is the efficiency of an algorithm measured?

Chapter 8

Introduction to Object-Oriented Programming

8.1 Introduction

Object-Oriented programming is a programming paradigm that treats everything in a program as an "object", thus the entire program is just a composition of interactions between many objects. Object-oriented paradigm has a long history in programming, and it remains the most important paradigm in industry today. Many areas of computing, such as graphics and gaming, benefit a lot from this paradigm. This chapter introduces some basic concepts of object-oriented programming, including *class*, *instance*, *inheritance*, *interface*, and *polymorphism*.

8.2 Class

So far, all the programs that you have written are in *imperative pardigm*: you are giving specific instructions to computer by writing down statements and declearing methods. The program will run sequentially in the way that you organize it. However, sometimes programming like this can be very tedious. For example, if you implement a sorting algorithm that takes an integer array as input and returns a sorted array, and your friend wants to use your code on a list of type double, she has to copy everything that you have written before except changing the type of the paramenter and return type. If this is not tedious, what if a group of people want to use your sorting algorithm to sort out a variety kinds of items: floating-point numbers, strings, toys, cards, ... As you can tell, if there is a method that can sort everything, everyone can save a lot of time from repeting writing the same code over and over again!

Object-oriented programming is the solution to this kind of scenario. Object-oriented programming emphasizes on the reusability of code so that lagacy code can be reused today. The key to understand object-oriented programming is to understand abstraction.

8.2.1 Abstraction

We are living in a very abstract world today. We have all kinds of electronics around us: electronic watch, microwave oven, computer, television, and game consoles, but you perhaps have never (or rarely) asked yourself: how do they work internally? When you get your new game console, you probably don't need to read the instruction manual to hook up wires with your television, insert your game disk into the console, and play your game with your joystick. When you use electronics, you don't care what is inside of them. All you need to know is that a game console needs power, has a DVD player, and can be connected to your television through a cable. You also know that your joysticks have buttons and control sticks so that you can control your game as indicated on the joystick. In a word, you don't really, and mostly importantL: you don't want to care about what is inside of them at all. All you need to know that they are supposed to work in the way like their names indicate.

This is abstraction: you don't have to understand how to wire a joystick to use it, and you don't need to know how to build a car to drive it. Similarly, you don't need to know how the function "Console.WriteLine()" implemented to use it.

8.3 Class Member

8.4 Inheritance

8.5 Interface

In many cases, instances of different classes have similar behaviour, and this is quite common in programming and real world. In real world, both human beings and parrots can "talk", but the way they talk are quite different: human beings actually commit meaningful talks while a parrot simply imitate what we teach him. However, since they both can "talk" in some way, it would be easier to treat them as a 'Talkable' type.

In programming, one way we can do is to define a 'Talkable' class, and if we have 'Human' and 'Parrot' classes, we can make both of them inherit methods and members of the 'Talkable' classes. But this can be very difficult to be clear to manage. For example: *add example here!*

To solve the problem of inheriting from multiple classes, we invented the idea of *interface*. An interface is a collection of method signatures that specify what methods must be implemented if this interface is going to be used by a class. Once a class implements all the required methods of this interface, instances of this class can also be considered as 'instances' of this interface, and thus we achieve the goal of multiple inheritance.

Programming Languages that Support Mutiple Inheritance

8.5.1 Declaration of Interfaces

To declare interface in C♯ :

```
public interface Talkable
{
    public void Speak (String content);
}
```

As you can tell, this interface only contains a method signature: `public void Speak (String content)`, but not define it at all! Yes, the implementation is left to individual class that is going to use this interface. For the example that we mentioned at the beginning of this section, we can make our two classes: `Human` and `Parrot` implement this interface:

For `Human` class:

```
    using System;

    public class Human : Talkable
    {
        private String name;

        public Human (String name)
        {
            this.name = name;
        }

        public void Speak (String content)
        {
            Console.WriteLine (this.name + ": " + content);
        }
}
```

For `Parrot` class, we would implement it slightly different so that it reflects the property of a parrot:

```
using System;

public class Parrot : Talkable
{
    private String name;
```

```
        public Parrot (String name)
        {
            this.name = name;
        }

        public void Speak (String content)
        {
            Console.WriteLine ("Hi, I am " + this.name + "!");
            Console.WriteLine ("Hi, I am " + this.name + "!");
            Console.WriteLine ("Hi, I am " + this.name + "!");
        }
    }
```

Then, in the main program, we can instantiate a human and a parrot, and make them talk:

```
    using System;
    public class Program
    {
        public static void Main (String [] args)
        {
            Talkable peter = new Human ("Peter");
            Talkable bob = new Parrot ("Bob");

            peter.Speak ("Today's lecture is about the life of Caesar...");
            bob.Speak ("I want to say something different!");
        }
    }
```

8.6 Polymorphism

8.7 Chapter Exercise

1. *Dog Class* If you want to create a class of dogs, which of these following attributes should be members, and which should be methods? Why or why not?
   (a) Hair color
   (b) Bark
   (c) Breathe

   (d) Weight
   (e) Eat
   (f) Run
   (g) Height
   (h) Sleep

2. *Public and Private* Which of the attributes above should be public? Which should be private? Why or why not?

3. *Bank Account Class* Consider a bank account as a class of objects. A typical bank account has checking and savings accounts, and the owner of the account can check the balance of each of them, and make deposit or withdraw money from both account. Write a class that can be instantiated.

4. *Candidate Class* Declare a class `Candidate`, which has attributes including `name`, `age`, and `party`. Then declare methods that can get the information of these attributes: `GetName()`, `GetAge()`, and `GetParty()`. Before you get started, think about the data type of each attibute, and how you would design and use this class. Once you complete, instantiate two candidates: one has name `"George W. Bush"`, age `58`, and party `"Republican"`, while the other one has name `"John Kerry"`, age `61`, and party `"Democrat"`. Use your getter methods to check your implementation.

Chapter 9

File I/O

9.1 Introduction

File and file system are one of the most important aspects of a computer and operating system. File system is a mechanism that allows data to be saved in non-volatile storage media, such as magnetic tape and disk, and many applications that we use today heavily involve with file input/output. This chapter will introduce the basic concepts of file and file system and how to interact with files using the C♯ programming langauge.

9.2 File System

File system is a mechanism that is provided by the operating system to store data persistently. Paper, songs, and movies that are stored on your computer are files. These kind of files can be read or written by their corresponding programs, such as a word processor or music editor. Programs are also made of many files, and they are stored on your computer for you to use.

Programs are special kind of file: they are not like your paper or music to be edited. Instead they are supposed to be *executed* by the computer. When a program is executed, it needs a *directory* to run under. For example, if you double click your music file, your defalut music player will start, and it is very likely that it only recognize this file in this directory, and may only identify other music files that you put in this directory. If you have other music files in other directory, it is not likely for that music player to find that file unless you add that music file to the library of your music player, in which case your music player will memorize the path to that file and the next time you start it up it will look up that file from the path you gave.

9.3 Standard I/O

There are many ways that you and your computer communicate with each other. For example, when your program prints out message in the console, it is a form of output. When your program is reading your input, it is a form of input. A typical computer system usually receives inputs as a stream of characters, and generates outputs in a similar form. This is called standard I/O.

Many operating systems support this standard I/O.

9.4 Command-Line Arguments

Sometimes we need to pass some information to a program in order to make it work

9.5 File

9.5.1 Definition of File

So far, all the data that our programs use reside in the memory of a computer, which means that once program exits, all the data that you had in your program will be lost. This is not desirable if you want to save the data for later use. File, as a mechanism of non-volatile storage, solves this problem. File can be considered as a sequence of bits or characters stored in a non-volatile media, such as magnetic disk or tape, optical disks (such as CD/DVD), and solid-state drives. Data stored in these kind of media are not volatile and data can be loaded to memory if a program decides to use them. However, since these storage media is usually slower than the speed of memory, reading and writing to these media can be very slow.

The classical definition of a file is a stream of character. Such file is usually represented as text file in many operating system. You can imagine that a file is a long list of characters. For example, if you open your text editor on your computer, write "Hello world!" and save it as hello.txt, then it probably would look like this: (put an image here).

Visiting a file is a complex process. When a program needs to visit a file, the operating system will first check and see if that program has enough privilege to access that file. Some files can be read and write only, while other files, such as system files, are only for reading by normal users and are editable for priviledged users (such as system administrator).

Once the operating system confirms that program has enough privilege to access that file, the operating system returns the control of computer to that program. Program will open that file and do whatever it is allowed to do on that file. Program should, but does not have to, close that file once it is done with using that file. If the program forgets to close that file, the operating system will close the file for the program once it exits.

The algorithm for a program to access a file should be like this:

1. Obtain the file path and name
2. Check if that file exists
3. Try to open that file and read it..
4. Do whatever it is allowed or commanded to do on that file.
5. Close the file once work is done.

However, if any other issue happens during the process above, the program usually crashes and exits. This is mostly because some exceptions happen while trying to access that file and that exception is not handled anywhere in that program. In the later section of this chapter, we will discuss how to handle exceptions in program.

9.5.2 Read a File

Since file is nothing but a sequence of character or bits, thus it make sense to read them sequentially: from the first character/bit all the way until the last character, which is EOF (End of File) in Unix system.

Before we can read a file, we must specify which file we want to read and where it is located. You usually need a to specify a path to that file, and the operating system will guide your program and show where that file is.

In C♯, to read a file, you need to specify the path to that file.

9.5.3 Absolute Path and Relative Path

There are two ways to specify the path of a file in a system: absolute path and relative path. They are named "absolute" and "relative" is because the location of a particular file can be different based on the "reference frame" that we specify. An absolute path is the path that is from the root directory to that file, and a relative path is the path that is from current working directory to that file. For example, in a typical Unix operating system, our target file is under Documents/ directory and is named as 'myfile.txt'. The absolute file path would be like this: '/home/user/Documents/myfile.txt'.

Suppose our current working directory is under the Desktop/ directory, and we want to use the file myfile.txt using relative path, we would use: '../Documents/myfile.txt'.

The major difference between an absolute file path and a relative one is that: the absolute file path is never change, while the relative path changes according to the path of current working directory.

9.5.4 File Reading Algorithm

A typical file reading algorithm is like this:

1. Get the path to target file.
2. Check if file exists.
   - If yes, proceed to next step.
   - If no, create that file or handle exception.
3. Check if program has permission to access that file.
   - If yes, proceed to next step.
   - If no, ask for permission or handle exception.
4. Use the given path to create a file object (Java or C♯) or create a file descriptor (C/C++)
5. Perform whatever actions that are required by the program on that file. (NOTE: exceptions may happen here as well!)
6. Close the file.

Let's write a simple C♯ program that opens and print out the content of a text file. Suppose you have a file named students.txt that contains the name of students in a class. We can simply write a program like this:

```
using System;
using System.IO;

public class ReadFile
{
    public static void Main(string [] args)
    {
            string path = "students.txt";
            StreamReader sr = new StreamReader(path);
            string line = sr.ReadLine();
            while (line != null) {
                Console.WriteLine(line);
                line = sr.ReadLine();
            }
            sr.Close();
    }
}
```

You may noticed quite a few flaws of the program above:

1. It can only read the file whose name is "students.txt", and this file must be in the same directory as the executable program. If either of those conditions fail, the program will not run at all.

2. It does not check if file exists or not. If file does not exist, this program is very likely to crash.

In order to handle the issues listed above, we need to improve the robustness of our program. An intuitive solution would be like this:

9.6 Exception

9.6.1 What is an Exception

Usually, evaluation of expression will generally result a concrete value, such as 3 when expression is 15/5, or g̈oodẅhen expression is ẗhisisgood̈.Substring(6). All these evaluation are based on the assumption that the input expression is valid. However, in many cases, the input expression is invalid, thus expression cannot be properly evaluated. For example, divide 20 by 0 can either crash the program or give a weird result; looking for a file that is not in the given path usually causes the main program to crash.

Therefore, to improve the stability of programs, we need a better mechanism to protect our programs from bad inputs. This mechanism is exception-handling.

An exception can be understood as an alert when

9.7 Chapter Exercise

1. What is a file in computer? What is the difference between a file and the data in memory?

2. Write a C♯ program that can open a text file and print out the contents. Use command-line argument to obtain the file name.

Chapter 10

Introduction to Software Development and Testing

10.1 Introduction

TBD

10.2 Debugging

10.3 Refactoring

10.4 Chapter Exercise

1. TBD.

Appendix A

Set up Programming Environment

Programming is not arcane at all if you have set up the right tools for this work. To programming in C♯ , all you need is just two programs: a text editor and a compiler.

A.1 Text Editor

A text editor is simply a program that can edit text files. Unlike other documents such as a Microsoft Word document or a LibreOffice Writer document, a text file does not contain any sophisticated format such as page margin or font style. Instead it usually only has one font style and very basic format. Since programs must be written in plain text in order for compiler to recognize and compile them, it is very important for us to write programs in text editor.

There are a variety of text editors avaiable for many platforms, including Windows, OS X, and Linux. This book mainly uses an open-source text editor: atom. You can download it and install it from this link: http://atom.io.

Once you have a text editor installed, you can start writing programs. However, unless you have a compiler or implement your own C♯ compiler, you still cannot turn your source code into an executable file for your computer to run.

A.2 C♯ Compiler

Depending on the operating system that you use, you have different options of C♯ compiler.

Mono is a free implementation of the C♯ programming language. Its compiler is available for download and use for free. You can visit http://www.mono-project.com/download/ for more detailed installation guide.

Appendix B

Compile C♯ Program

B.0.1 Ubuntu Linux

For users whose primary operating system is Ubuntu Linux, the default C♯ compiler is the one implemented by Mono. To invoke this compiler and compile your program, simply type:

```
$ mcs program.cs
```

This will generate an executable file named `program.exe`. You can run this program by typing:

```
$ ./program.exe
```

If this does not execute the program, then you have two ways to do:

1. Change the file to executable by using the `chmod +x` shell script.

2. Using Mono runtime environment to run this program: `mono program.exe`.

Either way would work.

B.1 Using Git to Manage Code

Git is a very useful tool for managing programs. It is firstly written by Linus Torvalds in 1990s, and today it is the most popularly used version control program among software developers.

There are two good sites for managing your program on a remote server: GitHub and BitBucket.

GitHub is a good choice if you decide to make all your work publicly available, while BitBucket is good for those projects that are required remaining private and not open to anyone but those who have privilege to view your repository.

Bibliography

[1] Raymond, E.S, *The Art of Unix Programming*. Addison-Wesley Professional, Massachusetts, 2003.

[2] Sipser, M, *Introduction to the Theory of Computation*. PWS Publishing Company, Boston, Massachusetts, 1997.

Solutions to Selected Problems

1. *dogclass*
2.