Chapter 1 Introduction

High thoughts must have high language.
—Aristophanes

Objectives

- To understand what a computer programming is
- To be able to list the basic phases involved in writing a computer program
- To recognize two programming methodologies
- To know about the characteristics of object-oriented programming

1.1 Overview of Programming

1.1.1 What Is Programming?

Much of human behavior and thought is characterized by logical sequences. Since infancy, you have been learning how to behave and how to perform tasks. And you have learned to expect certain behavior from other people.

On the broad scale, mathematics could never have been developed without logically sequenced steps for solving problems and proving theorems. Mass production would never have worked without operations taking place in a certain order. Our whole civilization is based on the order of things and actions. We create order, both consciously and unconsciously, through a process that we call programming.

Programming is planning how to solve a problem. No matter what method is used—the pencil and paper, slide rule, adding a machine, or computer—problem solving requires programming. Of course, how one program depends on the device one uses in problem-solving.

Back to programming—"planning how to solve a problem", note that we are not actually solving a problem. The computer is going to do that for us. If we could solve the problem by ourselves, we would have no need to write the program. The premise for a program is that we don't have the time, tenacity or memory capabilities to solve the problem, but we do know how to solve it, therefore we can instruct a computer to do it for us.

Programming is planning the performance of a task or an event.

Computer programming is the process of planning a sequence of steps for a computer to follow.

Computer program is a sequence of instructions to be performed by a computer.

程序设计是规划一个任务或一个事件的执行过程。

计算机程序设计是设计一系列的步骤让计算机来执行的过程。

计算机程序是由计算机执行的一系列指令。

A simple example of this is calculating what the sum of all integers from 1 to 10,000 is. If you wanted to, you could sit down with a pencil and paper or a calculator and work this out however the time needed, plus the likelihood that at some point you might make a mistake, rendering that an undesirable option. Instead, you can write and run a program to calculate this sum in less than 5 minutes.

Example 1-1: An example of programming.

```
/*----------------------------------------------------------------------------------------------------------
* File: example1_1.c
* The program calculates the sum of all integers from 1 to 10,000.
*----------------------------------------------------------------------------------------------------------*/
1    #include <stdio.h>
2    #define MAX 10000
3
4    int main()
5    {
6        long sum = 0, number;
7        for( number = 1; number <= MAX; number ++)
8        {
9                sum += number;
10       }
11       printf("The sum of all integers from 1 to %ld is : %ld\n", MAX, sum);
12       return 0;
13   }
```

This example gives the result of 50,005,000. Meanwhile, you can verify this, as you know that the sum of integers from 1 to *N* can be calculated as

$(N+1) \times (N/2)$

$(10000 + 1) \times (10000/2) = 10001 \times 5000 = 50005000$

So, you have solved the problem of how to calculate the sum of all integers from 1 to 10,000 and the computer has solved the problem of calculating the sum of all integers from 1 to 10,000.

The computer allows us to do tasks more efficiently, quickly, and accurately than we could by hand—if we could do by hand at all. In order to use this powerful tool, we must specify what we want to do and the order in which we want to be done. We could do this

through programming.

However, we have to know the major differences between human and computers. Human has the judgment and free will and will not run any instruction they deem not required or nonsensical, whereas a computer will do exactly what it is told with no judgment on the need or sanity of the instruction.

1.1.2 How to Write a Program?

To write a sequence of instructions for a computer to follow, we must go through a three-phase process: problem-solving, implementation, and maintenance (see Figure 1-1).

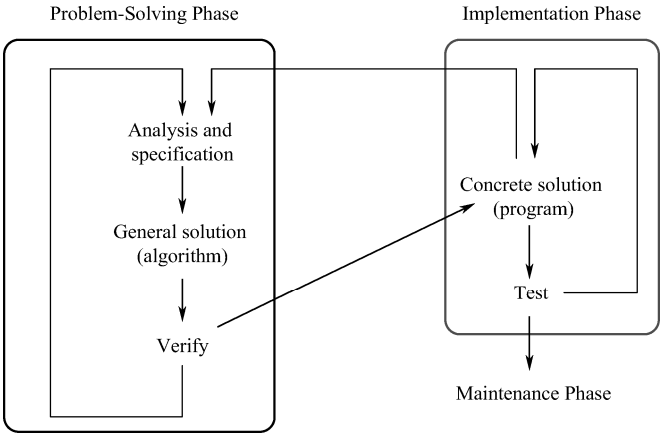

Figure 1-1 Programming process

Phase 1: Problem-Solving

(1) ***Analysis and specification***. Understand (define) the problem and what the solution must do.

Every program starts with the specification. This may be several hundred-page documents from your latest client or one small paragraph from your professor and pretty much anything in-between.

The specification is very important, and the specification writing is a whole sub-branch of programming. The important thing is that the specification is correct, otherwise, to use a well-known computing adage, garbage in garbage out.

However, the specifications are rarely perfect, and it is not at all uncommon for the specification to go through several iterations before being finally agreed on.

(2) ***General solution (algorithm)*****.** Develop a logical sequence of steps that solves the problem.

Having found out how to solve the problem you can then jump in and start coding. Right? Wrong! It is the time to do some program design, now. How much design is required, and

where the design is stored rather depend on the complexity of the program, the experience of the user, and the purpose of the program. For instance, a simple program being written by an experienced programmer just as a temporary project tool (i.e. will only be in use for a day or two days) probably only requires a little thought about the program design.

The design, once you have it, is not set in stone. It just gives the current ideas about how the program will be written. Normally, the final code of a project does not match the initial design exactly. The design will be also changed as the specification changes. However, the specification will give a clear start point for coding and will ultimately lead to a better and more maintainable code.

(3) *Verify*. Follow the steps exactly to see if the solution does solve the problem.

Phase 2: Implementation

(1) *Concrete solution (program).* Translate the algorithm into a programming language.

(2) *Test.* Follow the instructions to check the results manually. If you find errors, analyze the program and algorithm to determine the source of the errors, and then make corrections.

Testing does not have to involve running any code. Source review by your peers is a good form of testing too. You sit down around a table and go through the code line by line and examine it for logic errors, conformance to standards and programming errors. This can actually throw up errors that are not made obvious from testing the software by running it.

Once a program has been written, it enters the third phase: maintenance.

Phase 3: Maintenance

(1) *Use*. Use the program.

(2) *Maintain.* Modify the program to meet the changing requirements or correct any errors that showed up in using it.

The programmer begins the programming process by analyzing the problem and developing a general solution called an ***algorithm***. Understanding and analyzing a problem take up much more time than what Figure 1-1 implies. They are the core of the programming process.

> **Algorithm** is a step-by-step procedure for solving a problem in a finite amount of time.
> **算法**是在有限的时间内逐步解决问题的过程。

An algorithm is a verbal or written description of a logical sequence of actions (or events). After developing a general solution, the programmer tests the algorithm, walking through each step mentally or manually. If the algorithm does not work, the programmer repeats the problem-solving process, analyzes the problem again and comes up with another algorithm.

When the programmer is satisfied with the algorithm, he/she translates it into a programming language. Although a programming language is simple in form, it is not always

easy to use. Programming forces you to write very simple, exact instruction. Translating an algorithm into a programming language is called ***coding*** the algorithm.

Once a program has been put into use, it is often necessary to modify it. Modification may involve fixing an error that is discovered during the use of the program or changing the program in response to changes in the user's requirements. Each time the program is modified, it is necessary to repeat the problem-solving and implementation phases for the modified aspects of the program. This phase of the programming process is known as ***maintenance*** and actually accounts for the majority of the effort expended on most programs.

In a word, the problem-solving, implementation and maintenance phases constitute the *program's life cycle*.

1.2 Programming Methodologies

Programming methodology deals with the analysis, design and implementation of programs. There are many forms of programming methodology. The purpose of making explicit awareness of programming methodology is to allow the programmer to be aware of the processes and procedures which they use when constructing programs.

Data Abstraction

One of the keys to successful programming is the concept of abstraction. Abstraction is crucial to building the complex software systems. A good definition of abstraction comes from and can be summed up as concentrating on the aspects relevant to the problem and ignoring those that are not important at the moment.

The psychological notion of abstraction allows one to concentrate on a problem at a certain level of generalization regardless of the irrelevant low-level details; use of abstraction also allows one to work in an environment with familiar concepts and terms without the need to transform it to an unfamiliar structure.

There are two top approaches to programming design, that is, the structured approach and the object-oriented approach, which are outlined below.

1.2.1 Structured Programming

Dividing a problem into smaller sub-problems is called ***structured design***. Each sub-problem is then analyzed and a solution is obtained to solve the sub-problem. The solutions to all sub-problems are combined to solve the overall problem. This process of implementing a structured design is called ***structured programming***. The ***structured-design*** approach is also known as the top-down design, stepwise refinement and modular programming (see Figure 1-2).

For using structured programming, there are many good reasons: codes are easier to understand and errors are easier to find. An error will always be localized to a subroutine or function rather than buried somewhere in a mass of code. The scope of variables can be controlled more easily. With the reuse of codes—as well as being reused within a single application, the modular programming allows codes to be used in multiple applications. Thus, programs are easier to design—the designer just needs to think about the high-level functions. Collaborative programming is possible—modular programming enables more than one programmer to work on a single application at the same time while the codes can be stored across multiple files.

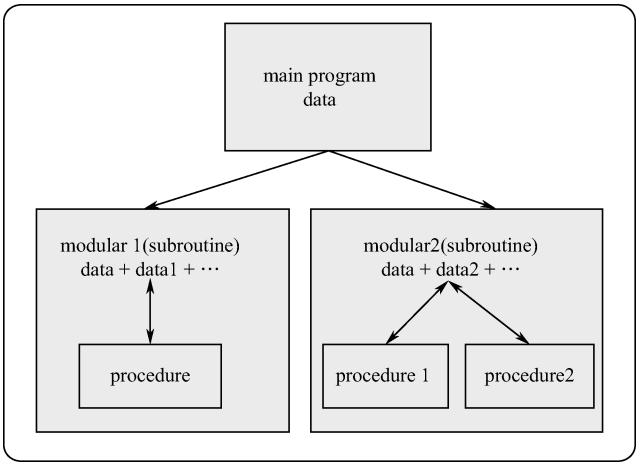

Figure 1-2 Structured programming (or Modular programming)

Now, we write a simple program which calculates the circumferences and areas of different sized circles. We design it in two ways.

Example 1-2: A simple example without structured programming.

```
/*----------------------------------------------------------------------------------------------------
* File: example1_2.c
* The program calculates the circumferences and areas of different sized circles without using
* structured programming.
*----------------------------------------------------------------------------------------------------*/
1       #include <stdio.h>
2       #define PI 3.14156926
3       int main()
4       {
5           float r1 = 1.0;
6           float c1 = 2 * PI * r1;
7           float a1 = PI * r1 * r1;
8           printf("The circumference is %6.2f, the area is %6.2f", c1, a1);
9
10          float r2 = 2.0;
```

```
11          float c2 = 2 * PI * r2;
12          float a2 = PI * r2 * r2;
13          printf("The circumference is %6.2f, the area is %6.2f", c2, a2);
14
15          float r3 = 3.0;
16          float c3 = 2 * PI * r3;
17          float a3 = PI * r3 * r3;
18          printf("The circumference is %6.2f, the area is %6.2f", c3, a3);
14
15          float r4 = 4.0;
16          float c4 = 2 * PI * r4;
17          float a4 = PI * r4 * r4;
18          printf("The circumference is %6.2f, the area is %6.2f", c4, a4);
19
20          return 0;
21      }
```

Here the same functionality is repeated and work indeed. However, the programmer can work more efficiently by using structured programming.

The aim of the programmer is now to ensure that the code can be reused as much as possible. Such goal can be achieved through by identifying the code that can be placed in separate functions and subroutines.

Example 1-3: A simple example with structured programming.

```
/*-------------------------------------------------------------------------------------------------------------
* File: example1_3.c
* The program calculates the circumferences and areas of different sized circles with using
* structured programming.
*-------------------------------------------------------------------------------------------------------------*/
1       #include <stdio.h>
2       #define PI 3.14156926
3
4       float circum (float r)
5       {       return 2 * PI * r;      }
6       float area (float r)
7       {       return PI * r * r;      }
8       void print (float r)
9       {       printf("The circumference is %6.2f, the area is %6.2f", circum(r), area(r));    }
10
11        int main()
12      {
13              print(1.0);
14              print(2.0);
15              print(3.0);
16              print(4.0);
17              return 0;
```

```
18  }
```

The output is the same as above, but this time the chance of the script operation being altered due to a typing error is greatly reduced. In addition, if there is an error, the task of correcting it is made much simpler. The programmer can also use the functionality throughout their application without worrying about having to rewrite the code, thereby improving the code and the time efficiency.

Structured programming is a programming paradigm that enforces a logical structure on the program being written to make it more efficient and easier to understand and modify.

It is a kind of top-down design, stepwise refinement and modular programming. It divides a problem into several sub-problems, and each sub-problem is then analyzed, afterward, a solution is obtained to solve the sub-problem. The solutions to all sub-problems are finally combined to solve the overall problem.

结构化程序设计是一种编程范式，它使得编写的程序更有逻辑结构，以便让它更有效、更容易理解和修改。

它是一种自顶向下、逐步精细、模块化的程序设计方法。它将一个问题被划分为若干子问题，对每个子问题进行分析和求解。最终所有子问题的解构成这个问题的解。

1.2.2 Object-Oriented Programming

Object-Oriented Programming **(OOP)** is a widely used programming methodology. In OOP, the first step is to identify the components called ***objects***, which form the basis of the solution, and to determine how these objects interact with one another, as shown in Figure 1-3. The next step is to specify for each object the relevant data and possible operations to be performed on the data.

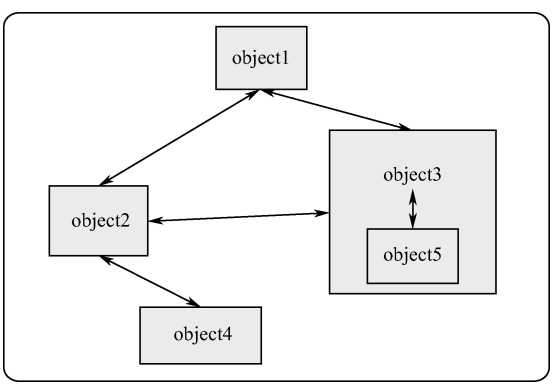

Figure 1-3 Object-oriented programming

Object-oriented programming is centered on the object. An object is a programming element that contains both the data and the procedures that can be operated on the data. The objects contain, within themselves, both the information and the ability to manipulate the

information.

Let us change the program in Example 1-3 using object-oriented programming.

Example 1-4: A simple example with OOP.

```
/*-----------------------------------------------------------------------------------------------------------
* File: example1_4.cpp
* The program calculates the circumferences and areas of different sized circles with using OOP.
*-----------------------------------------------------------------------------------------------------------*/
1    #include <stdio.h>
2    #define PI 3.14156926
3
4    class Circle{
5    public:
6       Circle(float R)
7       {    r = R;    }
8       float circum()
9       {    return 2 * PI * r;    }
10      float area()
11      {    return PI * r * r;    }
12      void print()
13      {    printf("The circumference is %6.2f, the area is %6.2f", circum(), area());    }
14   private:
15      float r;
16   };
17
18   int main()
19   {
20       Circle c1(1.0), c2(2.0), c3(3.0), c4(4.0);
21       c1.print();
22       c2.print();
23       c3.print();
24       c4.print();
25       return 0;
26   }
```

From the first glance in Example 1-4, you may see that everything following “public:” is the functions: *circum*, *area* and *print*, while “private:” denotes the *r* variable. This is the definition of an object, called a ***class***. We can see that class *Circle* combines data and operations on data into a single unit. In the *main* function, we create four different objects of class *Circle*, i.e. *c*1, *c*2, *c*3 and *c*4, and deal with them directly.

The primary differences between the two approaches are their use of data. In a structured program, the design centers on the rules or procedures for processing the data. The procedures, implemented as functions in C++, are the focus of the design. The data objects are passed to the functions as parameters. The key question is how the functions will transform the data they

receive to either storage or further processing. In an object-oriented program, the design centers on the objects that contain the data and the necessary functions to process the data. Procedural programming has been the mainstay of computer science since its beginning. However, object-oriented programming is used as a mainstream program method today.

Think it Over

Why use classes in OOP instead of functions?

1.3 Characteristics of Object-Oriented Programming

Object-oriented programming (OOP) is a programming paradigm that uses "objects"—data structures consisting of data fields and methods together with their interactions—to design applications and computer programs. Programming techniques may include features such as data abstraction, encapsulation, polymorphism, and inheritance. It was not commonly used in the mainstream software application development until the early 1990s. Many modern programming languages now support OOP.

The root of object-oriented programming can be traced to the 1960s. As hardware and software became increasingly complex, manageability often became a concern. Researchers studied ways to maintain software quality and developed object-oriented programming in part to address common problems by strongly emphasizing discrete, reusable units of programming logic. Such technology focuses on the data rather than the processes, with programs composed of self-sufficient modules ("classes"), each instance of which ("objects") contains all the information needed to manipulate its own data structure ("members"). This is in contrast to the existing modular programming which had been dominant for many years that focused on the *function* of a module, rather than specifically the data. Nevertheless, it equally provided for code reuse, and self-sufficient reusable units of programming logic, enabling collaboration through the use of linked modules (subroutines).

An object-oriented program may thus be viewed as a collection of interacting *objects*, as opposed to the conventional model, in which a program is seen as a list of tasks (subroutines) to perform. In OOP, each object is capable of receiving messages, processing data, and sending messages to other objects. Each object can be viewed as an independent "machine" with a distinct role or responsibility.

Class

A ***class*** is a user-defined data type which contains variables, properties and methods in it. A class defines the abstract characteristics of a thing (object), including its characteristics (its

attributes, fields or properties) and the thing's behaviors (the things it can do, or methods, operations or features). One might say that a class is a *blueprint* or *factory* that describes the nature of something.

For example, the *Dog* class would consist of traits shared by all dogs, such as breed and fur color (characteristics), and the ability to bark and sit (behaviors). Classes provide modularity and structure in an object-oriented computer program. A class should typically be recognizable by a non-programmer familiar with the problem domain, meaning that the characteristics of the class should make sense in context. Also, the code for a class should be relatively self-contained (generally using encapsulation). Collectively, the properties and methods defined by a class are called members.

Method

The ***method*** is a set of procedural statements for achieving the desired result. It performs different kinds of operations on different data types. In a programming language, methods (sometimes referred to as "functions") are verbs. For example, *Lassie*, being a *Dog* object, has the ability to bark. So *bark* is one of *Lassie*'s methods. She may have other methods as well, for example, *sit* or *eat* or *walk* or *save*. Within the program, using a method usually affects only one particular object; all dogs can bark, but you need only one particular dog to do the barking.

Message Passing

Message passing is the process by which an object sends data to another object or asks the other object to invoke a method which is known to some programming languages as interfacing. For example, the object called *Breeder* may tell the *Lassie* object to sit by passing a "*sit*" message which invokes *Lassie*'s "*sit*" method.

Abstraction

Abstraction is one of the most powerful and vital features provided by object-oriented programming. The concept of abstraction relates to the idea of hiding data that are not needed for the presentation. The main idea behind data abstraction is to clearly separate the properties of data type and the associated implementation details. This separation is achieved so that the properties of the abstract data type are visible to the user interface while the implementation details are hidden. Thus, abstraction forms the basic platform for the creation of user-defined data types called objects. ***Data abstraction*** is the process of refining data to its essential form. An ***Abstract Data Type*** (ADT) is a data type that is defined in terms of the operations that it supports but not in terms of its structure or implementation.

In object-oriented programming language C++, it is possible to create and provide an interface that accesses only certain elements of data types. The programmer can decide which user to give or grant access to and hide the other details. This concept is called data hiding

which is similar in concept to data abstraction.

For example, in a touch screen at the railway station or ATM machine in the bank, we just use the touch screen application to satisfy our needs, however, we don't see what is happening inside its software or about its OS.

Encapsulation

Encapsulation conceals the functional details of a class from objects that send messages to it.

For example, the *Dog* class has a *bark* method. The code for the *bark* method defines exactly how a bark happens (e.g., by an *inhale* method and then an *exhale* method, at a particular pitch and volume). *Timmy*, *Lassie*'s friend, however, does not need to know exactly how she barks. Encapsulation is achieved by specifying which classes may use the members of an object. The result is that each object exposes to any class a certain ***interface***—those members accessible to that class. The reason for encapsulation is to prevent clients of an interface from depending on those parts of the implementation that are likely to change in the future, thereby allowing those changes to be made more easily, that is, without changes to clients. For example, an interface can ensure that puppies can only be added to an object of the *Dog* class by code in that class. Members are often specified as public, protected or private, determining whether they are available to all classes, sub-classes or only the defining class.

Inheritance

Inheritance is a process in which a class inherits all the state and behavior of another class. This type of relationship is called child-parent or is a relationship. "Subclasses" are more specialized versions of a class, which inherit attributes and behaviors from their parent classes, and can introduce their own.

For example, the *Dog* class might have sub-classes called *Collie, Chihuahua*, and *GoldenRetriever*. In this case, *Lassie* would be an instance of the Collie subclass. Suppose the *Dog* class defines a method called *bark* and a property called *furColor*. Each of its sub-classes (*Collie, Chihuahua*, and *GoldenRetriever*) will inherit these members, meaning that the programmer only needs to write the code for them once.

Each subclass can alter its inherited traits. For example, the *Collie* subclass might specify that the default *furColor* for a *Collie* is brown-and-white. The *Chihuahua* subclass might specify that the *bark* method produces a high pitch by default. Subclasses can also add new members. The *Chihuahua* subclass could add a method called *tremble*. Thus, an individual *Chihuahua* instance would use a high-pitched *bark* from the *Chihuahua* subclass, which in turn inherited the usual *bark* from *Dog*. The *Chihuahua* object would also have the *tremble* method, but *Lassie* would not, because she is a *Collie*, not a *Chihuahua*. In fact, inheritance is an "a...is a" relationship between classes, while instantiation is an "is a" relationship between

an object and a class: a *Collie* is a *Dog* (“a... is a”), but *Lassie* is a *Collie* (“is a”). Thus, the object named *Lassie* has the methods from both classes *Collie* and *Dog*.

Polymorphism

Polymorphism allows the programmer to treat derived class members just like their parent class’s members. More precisely, Polymorphism in object-oriented programming is the ability of objects belonging to different data types to respond to calls of methods of the same name, each one according to an appropriate type-specific behavior. One method, or an operator such as +, -, or *, can be abstractly applied in many different situations. If a *Dog* is commanded to a *speak* method, this may elicit a *bark* method. However, if a *Pig* is commanded to *speak* this may elicit an *oink* method. Each subclass overrides the *speak* method inherited from the parent class *Animal*.

1.4 C++ Programming Language

“As long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem and now that we have gigantic computers, programming has become an equally gigantic problem. In this sense the electronic industry has not solved a single problem, it has only created them—it has created the problem of using its product” (E.W. Dijkstra, Turing Award Lecture, 1972).

Nowadays, there are many programming languages, for example, C, C++, Ada, Pascal, Prolog, FORTRAN, Modula3, Lisp, Java, and Scheme. This alphabet soup is the secret power of modern software engineering. As high-level computer programming languages, they provide enormous flexibility and abstraction. Programmers are separated from the physical machine, allowing them to create complex problem solutions without fretting with the troubles of ones and zeros. This idea is clarified by an analogy. If we had to think about every phonetic sound made while speaking, communication of abstract ideas would be nearly impossible. Much the same way, programming directly with ones and zeros would focus the designer’s attention on trivial hardware details instead of on designing abstract solutions. Considering the historical trend that created high-level programming, we believe that certain reasonable predictions can be made regarding future advances.

1.4.1 History of C and C++

C evolved from two previous programming language BCPL and B. BCPL was developed in 1967 by Martin Richards as a language for writing operating systems software and compilers. Ken Thompson modeled many features in his language B after their counterparts in BCPL and used B to create early versions of the UNIX operating system at the Bell

Laboratories in 1970 on a DEC PDP-7 computer. Both BCPL and B were “typeless” language—every data item occupied one “word” in memory while the burden of treating a data item as a whole number or a real number, for example, fell on the shoulders of the programmer.

The C language was evolved from B by Dennis Ritchie at Bell Laboratories and was originally implemented on a DEC PDP_11 computer in 1972. C uses many important concepts of BCPL and B while adding the data typing and other features. C initially became widely known as the development language of the UNIX operating system. Today, most operating systems are written in C and/or C++.

By the late 1970s, C has evolved into what is now referred to as “traditional C”, “classic C”, or “Kernighan and Ritchie C”. The widespread use of C with various types of computers (sometimes called hardware platforms) unfortunately led to many variations. There were similar, but often incompatible. This was a serious problem for program developers who needed to write portable programs that would run on multiple platforms. It became clear that a standard version of C was needed. In 1983, the X3J11 technical committee was created under the American National Standard Committee on Computers and Information Processing (X3) to “provide an unambiguous and machine-independent definition of the language”. In 1989, the standard was approved. ANSI cooperated with the International Standards Organization (ISO) to standardize C worldwide, when the joint standard document was published in 1990 and referred to as the ANSI/ISO 9899:1990. Copies of this document may be ordered from ANSI. As the second edition of the one published by Kernighan and Ritchie in 1988, reflects this version is called ANSI C, a version of the C language now used worldwide.

C++, an extension of C, was developed by Bjarne Stroustrup at the Bell Laboratories during 1983-1985. Prior to 1983, Bjarne Stroustrup added features to C and formed what he called “C with classes”. He had combined the use of classes and object-oriented features of Simula, having the power and efficiency of C. The term C++ was first used in 1983.

The name C++ was coined by Rick Mascitti in the summer of 1983. The name signifies the evolutionary nature of the changes from C; ‘‘++’’ is the C increment operator. The slightly shorter name ‘‘C+’’ is a syntax error; it has also been used as the name of an unrelated language. Connoisseurs of C semantics find C++ inferior to ++C. The language is not called D as it is an extension of C, and does not attempt to remedy problems by removing features.

Like C, ISO also published the first international standard for C++ in 1998, known as C++98. C++98 includes the Standard Template Library (STL), the conceptual development of which began in 1979. In 2003 and 2005, ISO revised problems in C++98 respectively. A new C++ standard (known as C++11) was approved by ISO in 2011. C++ 11 has added new features into the core language and the standard library. These new features are very useful for

advanced C++ programming.

1.4.2 Learning C++

As Bjarne Stroustrup mentioned, C++ is a general-purpose programming language with advantages towards the system programming that

- is a better C,
- supports data abstraction,
- supports object-oriented programming, and
- supports generic programming.

The most important thing to do when learning C++ is to focus on concepts and not get lost in language-technical details. The purpose of learning a programming language is to become a better programmer, that is, to become more effective at designing and implementing new systems and at maintaining old ones. For this, the appreciation of programming and design techniques is far more important than the understanding of details, which comes with time and practice.

C++ supports a variety of programming styles. All are based on strong static type checking, and most aim at achieving a high level of abstraction and a direct representation of the programmer's ideas. Each style can achieve its aims effectively while maintaining run-time and space efficiency. A programmer originally using a different language (say C, Fortran, Smalltalk, Pascal, or Modula-2) should realize that to gain the benefits of C++, they must spend time learning and internalizing programming styles and techniques suitable to C++. The same situation applies to programmers used to an earlier and less expressive version of C++.

Thoughtlessly applying techniques effective in one language to another typically leads to awkward, poorly performing, and hard-to-maintain code. Such code is also most frustrating to write because every line of code and every compiler error message reminds the programmer that the language used differs from ''the old language''. You can write in the style of Fortran, C, Smalltalk, etc., in any language, but doing so is neither pleasant nor economical in a language with a different philosophy. Every language can be a fertile source of ideas on how to write C++ programs. However, ideas must be transformed into something that fits with the general structure and type system of C++ in order to be effective in different contexts.

C++ supports a gradual approach to learning. How you approach learning a new programming language depends on what you already know and what you aim to learn. There is no single approach that suits everyone. My assumption is that you are learning C++ to become a better programmer and designer. Ergo, your purpose in learning C++ is not simply to learn a new syntax for doing things the way you used to, but to learn new and better ways of building systems. This has to be done gradually because acquiring any significant new skill takes time

and requires practice. Consider how long it would take to learn a new natural language well or to learn to play a new musical instrument well. Becoming a better system designer is easier and faster, but not as much easier and faster as most people would like it to be.

Word Tips

abstract *adj.* 抽象的
abstraction *n.* 抽象，抽取
accurate *adj.* 精确的
actually *adv.* 实际上
adage *n.* 言语，格言
algorithm *n.* 运算法则
analogy *n.* 类似，相似，类推
analysis *n.* 分析，分析报告
broad *adj.* 宽的，广的
calculate *vi./vt.* 计算，估计
chihuahua *n.* 吉娃娃狗
clarify *v.* 使清楚
client *n.* 委托人，顾客
code *n.* 代码
cohesion *n.* 聚合
concrete *adj.* 实体的，有形的
coupling *n.* 耦合
crux *n.* 难点，关键
deem *vi./vt.* 认为，相信
detail *n.* 细节，小事
discrete *adj.* 分离的，不相关的
distinct *adj.* 清晰的，明显的
domain *n.* 范围，领域
dome *n.* 圆屋顶
dominant *adj.* 占优势的，突出的
efficient *adj.* 有能力的，效率高的
emphasis *n.* 强调，重点
encapsulation *n.* 封装
evolution *n.* 演变，进化，发展
gigantic *adj.* 巨大的，庞大的
implement *vt.* 执行，实现
individual *adj.* 个别的，单独的
infancy *n.* 早期
instruction *n.* 命令，指示
integer *n.* 整数
internal *adj.* 内部的
internalize *v.* 吸收同化
invoke *vt.* 调用
iteration *n.* 反复，迭代
logical *adj.* 逻辑上的
mainstay *n.* 支柱，骨干
maintain *vt.* 维护
methodology *n.* 方法
modular *adj.* 模块的
module *n.* 单元，单位
multiple *adj.* 多重的，多种多样的
paradigm *n.* 范式
parameter *n.* 参数
phase *n.* 阶段，时期
phonetic *adj.* 语言的
platform *n.* 平台
polymorphism *n.* 多态性
prediction *n.* 预言
premise *n.* 前提
procedure *n.* 程序，过程
property *n.* 特性，属性
psychology *n.* 心理，心理学
puppy *n.* 小狗
pyrrhic *n.* 出征舞
refinement *n.* 精化

robust *adj.* 强壮的，健全的
script *n.* 脚本
segment *n.* 部分，片段
separation *n.* 分离，分开
sequence *n.* 顺序
specification *n.* 说明，详述
specify *vt.* 详述
syntax *n.* 句法，句法规则
temporary *adj.* 临时的，暂时的
thereby *adv.* 由此，因而
thoughtlessly *adv.* 轻率地，草率地
topology *n.* 拓扑结构
trace *vt.* 追踪，发现，找到
tremble *vi.* 发抖，颤动
unambiguous *adj.* 不含糊的，清楚的

Exercises

1. What is planning the performance of a task or an event?

2. What is a step-by-step procedure for solving a problem in a finite amount of time?

3. What is OOP? What is the name of the data structure we used? What are the components of this data structure?

4. What are the characteristics of OOP?

5. Explain the difference between the structured programming and object-oriented programming.

6. Why do we need object-oriented programming?