

目录

0. 序言	1
0.1 条件检查	1
0.2 本书的适用人群	1
0.3 本书的特色	1
0.4 本书的使用方法	1
1. 编程导论	1
2. 常识	2
2.1. 计算机基础	2
2.1.1 计算机组成	2
2.1.2 控制台、终端、Shell等易混淆概念	4
2.1.3 数据格式和编码	4
2.1.4 计算机程序	5
2.2 编程语言基础	5
2.2.1 编译型语言和解释型语言	5
2.2.2 编程语言的实现	6
3. 从C++开始	7
3.1 万物起源「Hello World」	7
3.1.1 头文件	7
3.1.2 主函数	8
3.1.2 注释	8
3.1.3 命名空间	8
3.1.4 输入和输出	8
习题	9
3.2 基础数据类型和变量	9
3.2.1 基础数据类型	9
3.2.2 变量	10
3.2.3 指针、数组和引用	11
3.3 程序流程控制	12
3.3.1 顺序结构	12
3.3.2 分支结构	12
3.3.3 循环结构	13
3.4 函数	14
3.5 面向对象初步	16
3.5.1 类	16
3.5.2 继承	17
3.5.3 运算符重载	18
-1. 结语	18
-1.1 编者的话	18

0. 序言

本书为NSFZ2023级开发者社教材，旨在帮助读者从零开始学习编程。它将从基础语法、数据类型、流程控制、函数、面向对象编程等方面进行讲解，并结合大量的实例，帮助读者快速掌握编程技能，同时对编程的思想逐步建立理解。

（也可以在[此处](#)线上阅读）

0.1 条件检查

1. 您已满**中华人民共和国法律规定的10周岁**。
2. 您认为您能在[韦克斯勒儿童智力检测四](#)中达到正常水平（约80分）。
3. 您没有**令人费解的语言逻辑缺陷**。
4. 您拥有一台尚可正常开机的，外设齐全的个人电脑。
5. 上述电脑**工作正常**，并**搭载主流操作系统**，一般是Windows（视窗操作系统）。
6. 您的电脑**RAM大小**≥4GB。（RAM的概念见#2.1.1章节）

0.2 本书的适用人群

这本书适用于所有想要学习编程的人（以及真的愿意花钱弄一本这玩意的实体书的人）。无论你是完全没有编程基础的初学者，还是想不太系统地老年复健的满级巨佬，都可以从本书中获益。

0.3 本书的特色

本书的特色主要体现在以下几个方面：

- **注重实用性**：本书的所有内容都以实用为导向，重点讲解与实际应用相关的知识点。
- **循序渐进和突然跳级**：本书将编程知识分为多个阶段进行讲解，循序渐进，让读者能够轻松掌握，并且穿插跳级内容，使读者时刻认识到学无止境。
- **巨™多代码量**：本书配有大量的代码示例，帮助读者理解和应用编程知识。

0.4 本书的使用方法

本书可以作为一本自学教程，也可以作为编程课程的辅助教材（最好别）。如果是自学，建议读者按照书中的章节顺序进行学习，并结合书中的代码示例进行练习。如果是使用本书作为编程课程的辅助教材，建议读者在课堂上认真听讲，并结合本书的内容进行巩固。

1. 编程导论

程序不过是更详细的需求罢了

——沃·兹基·硕德

众所周知，程序是由一条条指令构成的。每一条指令都是电脑的一个基本动作，如“使屏幕上某一个像素点变成某种颜色”，或是“将某个电路接口设置为高/低电平”。人们负责给出这些指令，由计算机**按顺序**完成这一系列指令，便可以达到理想的效果。然而，这里的指令不免过于基本。这时，人们会将用于实现某种功能的一系列指令捆绑在一起，形成一个较基本指令更为高级、简介、实用的指令，这就是编程中一个重要的思想：抽象。抽象思想使得人们避免一些无意义的重复劳动，但这又引入了一个新的问题：计算机不像人拥有如此先进的思想，它实际上只能完成那些最基本的

指令，因此我们需要一个工具，将这些利于人们使用的高级指令翻译回计算机能够完成的底层的基础指令。这个工具就是人们使用的各类语言的编译器、解释器等。

因此，编程本质上就是提出需求的过程，需求的简洁程度与抽象程度呈正相关。同时我们可以给出编程的一般过程，即“人类提出需求->工具将抽象的需求翻译为基础指令->计算机执行这些基础指令”。

需要指出的是，本章中使用了大量不准确的说法，目的是便于理解其中的思想，当读者阅读完第二章后，可以对本章内容进行规范化的整合。

2. 常识

在门外和门内之间，有一个小小的门槛
——沃·兹基·硕德

这些内容并不像编程语言的内容等对编程起决定性作用，但它们也是编程中时常会接触到的东西，初学编程的一部分时间总是在了解这些知识上，因此不妨先学习这些内容，再去进行实际编程的学习。

2.1. 计算机基础

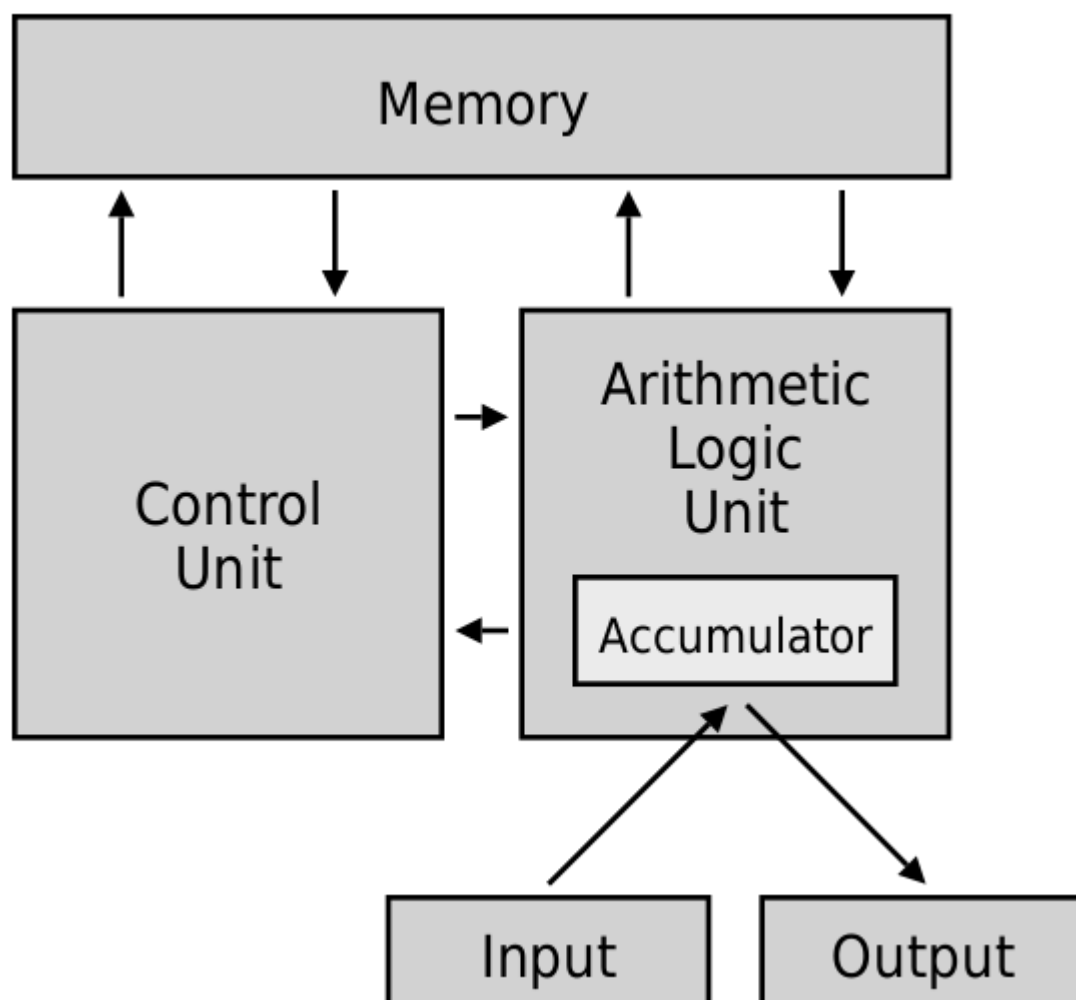
2.1.1 计算机组成

一般来说，计算机有五大组成部分：

- 控制器
- 运算器
- 存储器
- 输入设备
- 输出设备

其中控制器负责控制所有硬件的运行，其他都很好理解，这里不再解释。值得注意的是，存储器分为内部存储和外部存储，后者为常见的硬盘，软盘等，而前者又分为随机存储器（RAM）、只读存储器（ROM）和其他一些存储器。

这里借用维基百科上的一张图片，可以很好的展示这五个部分之间的关系。



前文所说的五大组成部分是逻辑结构，而计算机的实际结构又包含以下几部分：

- 中央处理单元（CPU）
- 图形处理单元（GPU）
- 主板、固件等硬件
- 键盘、屏幕等输入、输出设备

其中CPU对应逻辑结构中的控制器和运算器，是这两个逻辑结构的实现形式。

CPU有几个重要的性能指标，如主频和位宽。主频是描述CPU执行指令速度的量，它的单位即为频率的单位。位宽是CPU在一条指令中可以处理的二进制数据位数，例如32bit、64bit。

CPU有多种不同的实现形式，也就是架构和指令集。常见的架构有ARM系列、x86系列、MIPS系列和PowerPC系列。ARM架构的优势有价格低、能耗低，因此常用于嵌入式开发和手机等。x86架构的优势有兼容性强、性能高，常用于个人计算机。

关于CPU更详细的介绍可以参考以下文章：

<https://zhuanlan.zhihu.com/p/245119254>

<https://zhuanlan.zhihu.com/p/508557771>

我们日常生活中所使用的电脑一般是个人计算机（PC），它是计算机的一个主要的子集。按照层级来梳理，个人计算机的层级从底向上依次是硬件、操作系统、应用程序，其中操作系统和应用程序合称软件。在不引起歧义的情况下，有时软件也可仅指应用程序。主流的操作系统有Windows、Ubuntu、CentOS、Alpine等，它们的设计理念有所不同，但对于计算机的作用是类似的，即与硬件直接交互，为程序提供抽象的较高级的接口。

2.1.2 控制台、终端、Shell等易混淆概念

这些名词来源于大型机时代的一个个实体的设备，它们可能由不同的专业人员专职负责管理，到了今天，所有这些东西都在一台小巧的电脑里，但是由于惊人的历史遗留性，它们都被保留了下来（尽管是虚拟的），于是造成了如此混乱的体系。总的来说，它们可以分为两层：

1. 控制台、终端、串口终端：属于计算机五大组成部分的输入/输出设备，可以将用户指令（包括键盘事件）输入给操作系统并将操作系统返回的结果输出给用户（显示字符）。电传打字机（tty）是这类设备的具体实例。在现代计算机中，虚拟的终端被称为终端模拟器。特别地，终端模拟器在UNIX/Linux内核的驱动程序模块中为伪终端（pty）。
2. Shell：属于应用程序，负责解析、处理、运行命令，常见的Bash、Zsh、sh属于此类。

Windows系统中内置的命令提示符（CMD）和PowerShell，是终端和Shell的集合体，因为它们既处理输入/输出事件，也负责执行命令。也可以理解为它们是由终端和Shell两个独立的部分组成的。编者甚至觉得没有必要纠结这些模糊的概念性的东西

一般情况下，编程语言编写的程序使用其输入/输出模块与Shell直接交互。

2.1.3 数据格式和编码

我们知道，任何数据在计算机中都是以二进制的形式存储的。在操作系统中，数据以文件的形式被组织起来。那么，操作系统或程序是如何将这些文件中的数据（从底层上来说，都是二进制数据）识别为一张图片、一个文档、或是一份代码呢？这就需要数据格式。操作系统依靠文件的一些特殊信息（一般称为元数据）来识别文件的类型。一般有以下几种识别的方式：

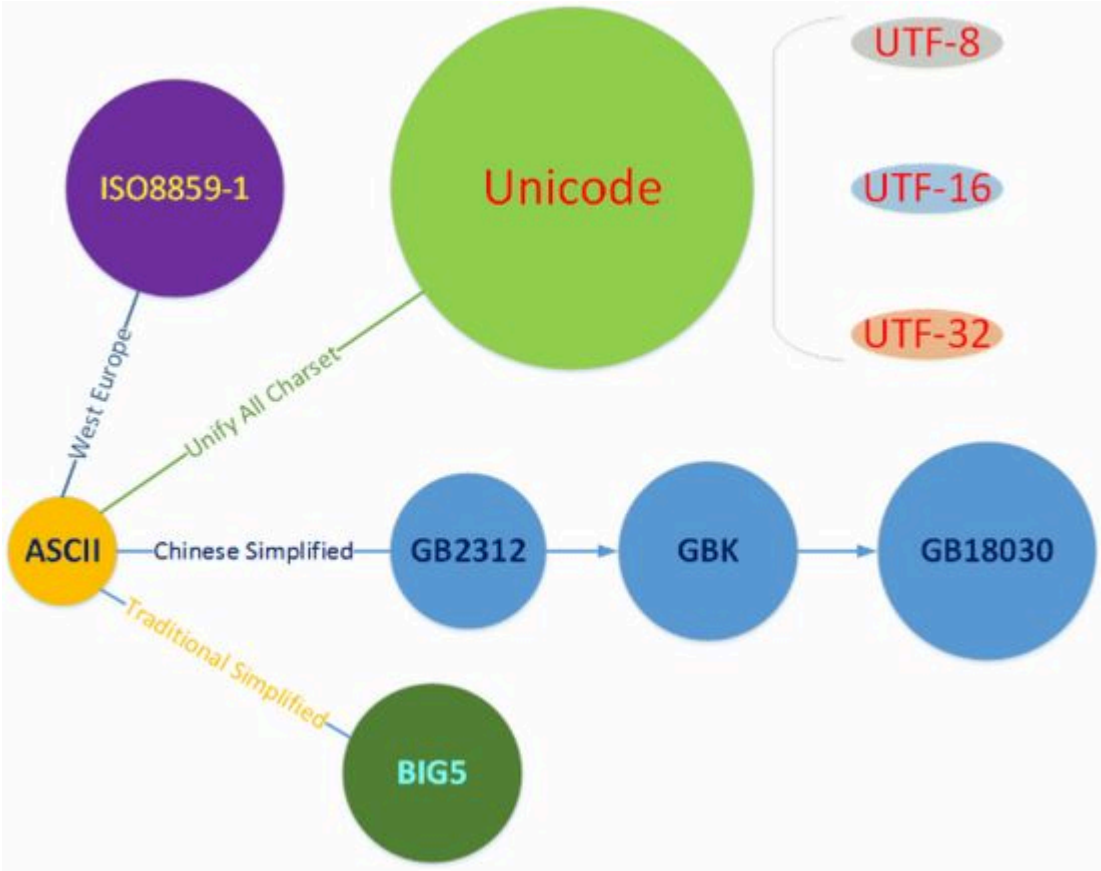
- 扩展名：即文件名最后一个 . 之后的内容，在Windows系统中，文件的默认打开方式与扩展名绑定。利用文件名识别的特性使得它易于更改，这既是优点也是缺点。由于扩展名没有官方统一的规定，不同类型的文件可能会使用相同的扩展名，从而在Windows系统中默认打开方式无法很好地分辨。
- 特征签名：即文件头部几个字节的数据，它通常与文件中数据的组织形式强相关。它不易被更改（因为更改后会影晌特定程序的读取），所以较扩展名更为可靠。

在很多时候，扩展名和特征签名会同时使用，因为它们被使用的时机和对象不同，存储的位置也不同。

对于一个文本文件，我们有更细致的需求，即将文件中的二进制数据识别为人类可读的，可以显示在屏幕上的字符。这时我们需要建议一个从二进制数据到字符的双向映射，于是就有了字符集。常用的字符集如下：

- ASCII（美国信息交换标准代码）：用一个字节表示一个字符，最初其二进制最高位始终是0，即只能表示128中字符，其中包括大小写字母、一些符号和控制字符。
- ISO8859-1（Latin1）：用一个字节表示一个字符，是对ASCII的扩展，增加了西欧语言、希腊语、泰语、阿拉伯语、希伯来语的符号。但是，由于出现得较晚，欧元符号没有被收录于其中。
- GB2312：1980年中国公布的国家标准，用两个字节表示一个字符，共收录6763个汉字，同时收录了包括拉丁字母、希腊字母、日文平假名及片假名字母、俄语在内的682个字符
- GBK：用两个字节表示一个字符，在兼容ASCII和GB2312的情况下可以表示的汉字达到了20902个，还有984个汉语标点符号、部首等，其中还包括繁体字。
- GB18030：用四个字节表示一个字符，在兼容以上字符集的情况下新增了大量字符，最新的GB18030-2022标准中收录的汉字达到了87887个，甚至包含了少数民族汉字。
- Unicode（万国码）：用四个字节表示一个字符，由美国加州的Unicode组织提出。Unicode的出现标志着字符集和字符编码的彻底分离，前文的几个既是字符集也是字符编码，而Unicode仅仅是字符集，其常用的字符编码有UTF-8、UTF-16等。

由于Unicode的独特设计，使得它可以包含世界上所有字符。借用CSDN上的一张图片总结：



关于这些字符集的具体编码方式可以参考这篇文章 <https://blog.csdn.net/wn084/article/details/80363792>

2.1.4 计算机程序

可以直接运行在操作系统中的程序，即本机程序，一般是一个二进制文件，其内容包括实际指令、预定义数据等。大多数还包含导入表，如果程序中调用了操作系统中或是其他程序中的代码，那么需要导入表来标示代码的位置。还有一类文件和本机程序的表现类似，例如在Windows系统下，如果安装了Python环境，那么可以直接双击一个 .py 文件来运行它，就像本机程序一样。但实际上这类文件并不是一个真正的程序，而是与它的运行环境绑定后得以实现的功能，即让另一个本机程序来解释这个文件的内容。其原理与直接打开一个 .docx 文档相同。

2.2 编程语言基础

2.2.1 编译型语言 and 解释型语言

上一节末尾，我们提到了本机程序和另一种类似本机程序的文件（姑且叫做非本机程序），由此可以引出编程语言的两大阵营：编译型语言和解释型语言。编译型语言的最终产物是一个本机程序，解释型语言的最终产物是一个非本机程序。之所以叫做解释型语言，就是因为编写完成后，需要另一个本机程序来解释它运行。将编译型语言的代码翻译为本机程序的一个特殊的本机程序，是编译器工具链（编译器及其周边组件，如链接器，稍后会详细介绍）。解释解释型语言的代码运行的本机程序，是解释器。编译器和解释器其实都是第一章中所说的那个将高级指令翻译回基础指令的工具。

常见的编译型语言有C、C++、Rust等，常见的解释型语言有Python、JavaScript等。虚拟机语言如C#/.NET、Java/JVM，介于编译型和解释型语言之间，它们的分类仍是个有争议的话题，但多数人将其归为解释型语言。由此可见，编译型语言和解释型语言的分类并不是完全清晰的。

关于虚拟机语言的扩展内容可以参考这篇文章 <https://www.iteye.com/blog/rednaxelafx-492667>

不难发现，编译型语言更加本质化，所以我们先介绍它。将编译型语言的代码翻译为本机程序是一个复杂的操作，以C++语言为例，它的编译过程如下：

1. 预处理 (Preprocess)：预处理器会对源代码进行操作，包括展开一些简写的代码，引入其他文件等。
2. 编译 (Compilation)：编译器将预处理后的源代码文件转换为目标代码，通常是汇编语言代码。编译器会进行语法检查，如果程序有语法错误，那么会报出错误 (Error) 和警告 (Warning)。
3. 汇编 (Assemble)：汇编过程将汇编代码转换成机器码，产生的文件叫做目标文件，是二进制格式。汇编过程需要为每一个源文件产生一个目标文件。
4. 链接 (Link)：链接器将目标文件组织起来，生成可执行文件。链接器会解析目标文件中的符号，将符号引用与符号定义关联起来，生成可执行文件。

解释型语言的运行过程是一个“模拟”的过程，即在运行时，解释器实时地根据代码执行相应的指令。可以这么理解：解释器作为一个本机程序，其中包含了解释型语言所有可能的代码所对应的的基本指令，运行代码时，解释器就根据代码找到对应的基本指令来执行。

2.2.2 编程语言的实现

由于编程语言通常只是一个标准，一套规范，它的具体实现依靠不同的编译器或解释器。对于编译型语言来说，不同编译器的差异无非在于其实现编程语言标准之外的扩展内容。如果一份编译型语言的代码完全符合语言标准，那么它理应可以被任何符合规范的编译器编译。解释型语言的不同解释器差异很大，本质上是因为运行的过程就是解释的过程，即解释器和代码在运行时是绑定且相互依靠的。

以典型解释型语言C++为例，它的编译器工具链有以下几种：

- GCC工具链：全称为GNU Compiler Collection，顾名思义，它可以处理C++、Objective-C、Go等语言，它包含预处理器/编译器cc1、汇编程序as、连接器ld (collect2) 等工具。它是大多数信息学竞赛的官方编译器工具链，因此广为人知。
- Clang/LLVM工具链：分为前端Clang和后端LLVM（这里指狭义LLVM，广义LLVM包含前端、优化器和后端，Clang是LLVM前端的一种），Clang将C++代码解析为统一的中间代码 (IR)，LLVM优化器负责优化IR，再交由后端生成本机代码。
- MSVC工具链：微软出品的工具链，通常用于开发Windows应用，它依靠Windows SDK，如果把MSVC理解为工具的话，那么SDK更偏向于工具所需要的物料。

以典型解释型语言Python为例，它的解释器有以下几种：

- CPython：Python官方默认的解释器，由C语言编写，它可以将Python代码编译为中间产物CPython字节码，再解释执行字节码。
- PyPy：著名的CPython替代品，由Python语言编写（事实上是RPython，Python语言的子集，去除了动态类型，详见第四章），它使用即时编译技术 (JIT)，可以在运行时将 Python 代码动态地编译成机器码，从而提供更高的执行速度，但它与许多使用C语言编写的Python拓展库不兼容，并且由于JIT，它的启动速度较慢。
- Stackless Python：一种 Python 的变体，它是在CPython的基础上进行扩展和改进的版本，提供了一些额外的特征。
- Jython：即Java Python，由Java语言编写。它运行在Java虚拟机 (JVM，与下文.NET虚拟机同类，类似于解释器) 上，因此提供了与Java的无缝集成。
- IronPython：运行在.NET虚拟机上，它可以与.NET平台无缝集成。

3. 从C++开始

There are only two kinds of languages: the ones people complain about and the ones nobody uses.
——Bjarne Stroustrup

3.1 万物起源「Hello World」

你说的对，但是「C++」是由Bjarne Stroustrup自主研发的一款全新开放世界冒险游戏。
——Anonymous

众所周知，学一门编程语言很显然要学习它的基础。「Hello World」是其一个非常经典的例子。考虑以下代码。

```
// helloworld.cpp (A hello world example for dummies)

#include <iostream>

int main() {
    // print "hello world"
    std::cout << "Hello world" << std::endl;
    return 0;
}
```

对以上代码稍作修改，可以**充分**体验到C++的魅力。例如：

```
// helloworld2.cpp (An example use of std::cout)

#include <iostream>

int main() {
    std::cout << "You're right, " << "\n";
    std::cout << "but" << "the C++ programming language" << "is a new, open-world adventure
game solely developed by Bjarne Stroustrup" << std::endl;
    return 0;
}
```

.....也就是本节的引言。

本节会介绍这个最基本的程序中的内容。（类似于第一章编程导论，为了易于理解、快速上手，本节也使用了不规范的说法）

3.1.1 头文件

C++中有许多内置的头文件，它们包含了C++标准库中的函数、类型等。可以用 `#include` 来引入它们。特别地，`#include <...>` 会使得编译器在内部目录查找文件，`#include "..."` 会使得编译器在用户定义的目录下查找文件（一般是源文件同级目录）。下面是部分常用的头文件和它们所包含的内容：

头文件	内容
<code><iostream></code>	输入/输出流

头文件	内容
<code><fstream></code>	文件流
<code><iomanip></code>	输入/输出流控制符
<code><cstdlib></code>	动态内存分配、随机数等常用函数和类
<code><cmath></code>	数学相关函数和常量

在C++中，以`#`开头的语句是给编译器看的，也就是说，并不会存在于编译出来的程序中。

3.1.2 主函数

主函数是C++程序的入口（*entry point*），即程序从这里开始执行。在标准的C++语言中必须返回`int`类型。函数的内容用两个花括号括起来。约定俗成的，在没有错误的情况下，程序总应返回整数0（`return 0;`）。

3.1.2 注释

在C++中，可以在代码中插入注释，这些注释不会影响编译和运行，但可以用来描述代码，让开发者（包括自己）获得更好的阅读体验。下面是单行注释和多行注释的示例：

```
// This is a single-line comment.
/*
This is a
multiline
comment.
*/
```

3.1.3 命名空间

命名空间用于处理命名冲突问题，即如果代码中需要出现两个同名函数，那么可以将它们至于不同的命名空间中，再用`namespace1::function()`和`namespace2::function()`来调用。定义一个命名空间的格式如下：

```
namespace ns {
    // ...
}
```

特别地，C++标准库中的大部分内容都在`std`命名空间中。

如果确保一份代码中不会存在命名冲突的问题，可以使用`using namespace std;`来引入命名空间，从而在之后的代码中不需使用`std::`前缀。虽然笔者不建议引入命名空间，尤其是`std`（你永远也不知道标准库里有什么）—

3.1.4 输入和输出

传统C++采用“及其先进”的流式输入输出，具体来说，使用控制台输出流（`std::cout`）和输出流操作符（`<<`）来进行输出，使用控制台输入流（`std::cin`）和输入流操作符（`>>`）来进行输入。输入或输出操作符可以连用，即可以使用`std::cout << var1 << var2;`的方式依次输出两个值。类似的，可以使用`std::cin >> var1 >> var2;`的方式依次输入两个值（一般情况下，以不可见字符空格、换行符、制表符Tab等分割）显然，一条语句不能既输入又输出，所以输入和输出流操作符不能连接使用。

C++标准库中包含多种输入/输出流控制元素，例如可以通过输出`std::endl`来进行换行。可以通过`std::cout <<`

`std::fixed << std::setprecision(n) << ...`; 来控制输出浮点数的小数位数为 n ，或去除 `std::fixed` 来控制输出浮点数的有效位数为 n 。

习题

为了帮助读者巩固知识，从本节开始，每节末尾都会有相关习题，参考答案可以在代码仓库找到。

习题 3.1-1

编写一段 C++ 程序，使得此程序输出以下内容。~~（经典老题）~~

```
I love playing
Genshin
Impact
```

3.2 基础数据类型和变量

3.2.1 基础数据类型

众所周知，程序最初就是用来计算的，在数学中，为了简化计算，或是重复利用已经计算的结果，我们常常会使用一些符号来表示具体数值。在编程中，我们也会定义一些变量代替具体值。编程中的变量较数学更为广泛，因为它可以表示数值，也可以表示字符、字符串，甚至是自定义的东西。在编程语言中，数据都是和类型相绑定的，一般来说，基础数据类型有以下几种：

- 整数（Integer）：与数学中的整数定义相同，因为二进制整数与十进制整数总是可以互相转换，所以不存在精度问题。在C++中，整数类型分为有符号整数（signed）和无符号整数（unsigned）两类。按照数据长度分类，可以分为16位（short）、32位（long）、64位（long long）。有符号整数可以表示的数的范围为 $[-2^{n-1}, 2^{n-1} - 1]$ ，无符号整数可以表示的数的范围为 $[0, 2^n - 1]$ ，其中 n 为数据长度（二进制位数）。整数类型的类型名组成为（符号限定符）（长度限定符）（int），其中符号限定符默认为 signed，长度限定符默认为 long，所以 int 表示32为有符号整数（在一些古老的机器上，int 也可能表示16位有符号整数）。符号限定符、长度限定符和 int 三者至少出现一个。~~（这不废话吗，都没了还算什么类型名）~~
- 浮点数（Float）：即小数，由于二进制小数与十进制小数不一定能恰好互相转换，所以存在浮点误差。单精度浮点（float）和双精度浮点（double）分别能表示7位和15位十进制有效数字。笔者测试得到下表。（测试代码见代码仓库）

类型	二进制位数	有效数字（十进制位数）	数值范围
float	32	6~7	$[\pm 1.17549 * 10^{-38}, \pm 3.40282 * 10^{38}]$
double	64	15~16	$[\pm 2.22507 * 10^{-308}, \pm 1.79769 * 10^{308}]$
long double	128	18~19	$[\pm 3.3621 * 10^{-4932}, \pm 1.18973 * 10^{4932}]$

- 字符（Character）：符合ASCII标准的字符，占用1个字节，类型名为 char。值得一提的是，为了支持国际化，C++还有宽字符类型 wchar_t 等类型。
- 布尔值（Boolean）：仅能表示真（true）、假（false）两个值，但仍占用1个字节，因为计算机中最小的单位为字节。

- 指针 (Pointer)：表示一个内存地址，因此其大小与去除指针后的类型大小无关，仅与程序位数有关，如32位程序的指针大小为4字节，64位程序的指针大小为8字节。一级指针的类型名组成为 `原类型*`，但定义多个变量时应在每个变量名前加 `*`，形如 `int *a, *b;`。指针可以嵌套，即多级指针，形如 `int**`。

C++的类型系统中包含非常多的基础类型，这里只介绍了常用的一部分，更多信息可参考<https://zh.cppreference.com/w/cpp/language/types>

3.2.2 变量

作为一门较为底层的语言，C++是静态类型的语言，即一个变量一旦定义好了，就只能表示这一个类型的值。同时它是可变的 (mutable)，除非在类型名前或后（一般习惯加在前）加上类型修饰符 `const` 使其成为常量。定义变量的语法为 `(类型修饰符) 类型名 变量名;`，如 `int x;`。之后可以用赋值运算符 `=` 对变量进行赋值操作：`x = 100;`。也可以在定义时同时进行赋值，如 `int x = 100;`，还可以写成 `int x{100};`，这两种的区别会在面向对象相关章节提到。还可以在一条语句中定义两个变量，如 `int a = 1, b = 2;`。需要注意的是，赋值运算符是有返回值的，其返回值之后被赋值的变量的引用（引用将在之后的章节中介绍，这里可以暂时忽略，理解为被赋值的变量），并且它是右结合的，即从右向左运算，所以连续使用赋值运算符，如 `a = b = c;` 等价于 `b = c; a = b;` 两条语句。

C++标准对标识符（变量名、函数名、类型名等）有一定要求，可以是小写字母、大写字母、数字和下划线的组合，并且第一个字符不能是数字。但现在甚至似乎可以用中文

定义在函数中的变量为局部变量，它在该函数运行完后就会被销毁；定义在 `main` 函数外的变量为全局变量，它会在程序运行结束后被销毁。

直接出现在代码中的值称为字面量 (literal)，如 `100`、`123.45`。一般编译器会自动推导字面量的类型，我们也可以通过字面量后缀手动指定类型，常见的后缀有 `l` (long)、`ul` (unsigned long)、`ll` (long long)、`ull`

(unsigned long long)、`f` (float)、`lf` (double)、`llf` (long double) 等。特别地，可以通过字面量前缀来指定数值类型的进制，常见的有 `0x` (16进制)、`0` (8进制)、`0b` (2进制，C++14及以后)。字符 (串) 字面量中有一类转义字符，即用 `\` 加一个字符表示一些常用的不可见字符，如 `\n` 表示换行符、`\r` 表示回车符、`\t` 表示制表符。

C++的数值类型变量和字面量都可以进行运算，与数学运算类似。特别地，两个整数进行除法操作时为整除，如果需要正常的除法，需要将其转换为浮点数再进行运算。除了常见的加减乘除运算符 (`+-*/`)，还有一个常用的整数间运算符为取余 `%`。一般的，可以将赋值运算符和其他运算符结合，构成自操作运算符，例如 `a = a + b;` 可以改写为 `a += b;`。当多个运算符同时使用时，运算符优先级规则，可以使用添加小括号的方式改变优先级。优先级顺序和结合性参考 https://zh.cppreference.com/w/cpp/language/operator_precedence

逻辑运算符是应用于两个 `bool` 值之间的运算符，其返回值也是 `bool` 值。常用的逻辑运算符如下表：

逻辑运算	符号	名称	真值判定
$\wedge$	<code>A&&B</code>	逻辑与	A与B同时为真
$\vee$	<code>A B</code>	逻辑或	A与B至少有一个为真
$\neg$	<code>!A</code>	逻辑非	A为假

与逻辑运算相对应，对于整数类型，有位运算：按位与 `A&B`、按位或 `A|B`、按位取反 `~A`、按位异或 `A^B`，表示对于整数的二进制表示中每一位进行相应的操作。

3.2.3 指针、数组和引用

由于指针的重要性以及它与引用的关系之密切，我们将其单独讲解。指针为“修改数据”和“表示数据”提供了新方法，区别于简单赋值和读取，它使得我们可以从更接近底层的角度访问数据。一般的，我们可以用取地址操作符 `&` 对一个已有的变量取地址，用解引用操作符 `*` 得到一个指针所指向的值。这样说其实并不准确，因为一个内存地址表示一个字节，但多数类型的大小是大于一个字节的，所以指针的值表示其指向的变量的第一个内存地址。记去除指针后的类型（下称原类型）的大小为 n ，则从这个地址开始的 n 个字节的位置实际存储这个变量的值。所以解引用实际上是从这个地址开始的连续 n 个字节的数据解释为原类型。这样讲似乎有些抽象，我们举一个具体例子。假设我们有一个 `int` 类型的变量 `v`，它在内存中占用1001、1002、1003、1004四个地址的空间（因为 `int` 类型大小为4字节），对其进行取地址操作就得到一个类型为 `int*`，实际值为1001的值。指针类型的值也可以进行加减操作，但与数值类型不同的是，`v + 1` 的实际值为1005，即按照原类型大小向后推 n 个字节（地址）。

指针可以嵌套使用，即多级指针。例如，`int**` 类型表示指向 `int*` 类型的值的指针。

还有一种特殊的指针：空类型指针 `void*`，顾名思义，这个指针类型的原类型为空（`void`），或者说不带有原类型信息。它可以被转换为任何类型的指针，因此在没有模板和泛型的C语言中常常被用来传递任意类型的参数，如C标准库中的 `qsort` 函数的比较函数的参数类型为 `void*`，用于对任意类型的数组进行排序。（这样做的另一个优点是不像值传参那样耗费资源，具体会在函数传参相关章节讲解）需要注意的是，`void` 类型不能单独使用，它只能作为指针或者返回值类型（表示不返回值）使用。

给指针加上一个长度信息，就成了数组，即一组在内存中连续的值。可以通过取下标操作符 `[]` 来获取相应下标（位置）的值的引用，需要注意的是，下标从0开始。例如，有如下定义 `int arr[5];`，表示包含5个 `int` 型元素的数组，则 `arr[3]` 表示第4个元素。与普通变量类似，我们可以在定义时赋初始值 `int arr[5] = {1, 2, 3, 4, 5};`。特别地，如果初始值是确定的，则可以省略中括号里的元素个数。如果指定了元素个数但设定的初始值个数小于指定的元素个数，则剩下的元素会被填充为默认值（对于数值类型来说，为0）。有一种常见的说法是数组本质上就是指针，其实不完全是，但由于C++的类型退化机制，使得数组类型在大部分情况下都被作为指针使用，甚至取下标操作符 `arr[3]` 都被编译器翻译为 `*(arr + 3)`（读者可根据上文提到的指针加减的意义来验证这样翻译的正确性），因此我们可以写出类似 `3[arr]` 的逆天代码（并且是完全符合语法的）。

和指针相同，数组也可以有多维。例如，对于定义为 `int arr[2][2]` 的二维数组进行一次取下标操作，可以得到一个类型为 `int*` 的值。

引用（reference）是C++相较于C特有的语法，它可以被简单理解为创建一个原变量的别名（而不是复制一份出来）。特别地，字面量不能被引用。引用的底层原理与指针相同，在大部分情况下可以用来代替指针，并且可以确保安全性，因为它必须引用到一个已定义的变量上，并且它引用的地址（可以理解为底层指针的实际值）不能被修改（对引用的操作就是对原变量的操作）。如果一个引用变量仅作为其他变量的别名来使用，它常常会被编译器优化掉，也就不占用实际空间。（但如果作为类的成员变量（在面向对象章节会介绍），它就会老老实实地用类似指针的方法来实现，因为编译器无法推导出它是否可以被优化）

对于指针类型，我们可以在其类型名的 `*` 之后加 `const` 修饰符，表示这个指针变量所指向的地址是不可修改的，即在赋初始值之后不能再给它赋其他地址值，但是仍然可以通过解引用来修改它指向的地址的值（除非原类型为常量类型）。给引用类型本身加 `const` 修饰符是不合语法的（本身也没有意义）。

习题 3.2-1

编写一段 C++ 程序，使其接受输入三个 `int` 范围内的正整数 a, b, c ，计算并输出三边长为 a, b, c 的三角形面积的平方。提示：使用海伦公式 ($S = \sqrt{p(p-a)(p-b)(p-c)}$, $p = \frac{a+b+c}{2}$)。

3.3 程序流程控制

3.3.1 顺序结构

编程中最常用的代码结构就是顺序结构，即按照代码中的顺序从前往后一行一行地执行程序。若无特殊操作（循环、多线程、异步等），直接编写的代码即为顺序执行的。

我们可以使用 `goto-label` 的方式来对顺序结构进行简单的修改，使用 `label`：在代码中加入一个名为 `label` 的标签，然后使用 `goto label`；来跳转至标签所在行开始向下执行。标签和 `goto` 没有先后顺序的要求。

3.3.2 分支结构

在生活中，我们常常需要根据外界环境给出相应的不同的反馈，可以将其抽象为“如果……，那么……，否则……”。类似的，在C++中，我们可以通过如下方式进行条件判断：

```
if (condition) {  
    // then...  
} else {  
    // otherwise...  
}
```

其中 `condition` 是一个返回值类型为 `bool` 的表达式或值（或任何可以隐式转换为 `bool` 类型的值），如果 `condition` 为 `true`，则执行 `// then...` 这一段代码，否则（即 `condition` 为 `false`）执行 `// otherwise...` 这一段代码。我们也可以去除 `else`，仅保留一个 `if`，或并列使用多个 `if-else`，如下：

```
if (condition1) {  
    // condition1 is true  
} else if (condition2) {  
    // condition2 is true (condition1 is false)  
} else {  
    // condition2 is false (condition1 is false)  
}
```

对于嵌套使用 `if-else` 的情况，我们推荐使用逻辑与运算符 `&&` 和逻辑或运算符 `||` 将其简化为一个或少数几个 `if-else`（可能需要根据优先级添加小括号），如下：

```
if (condition1) {  
    if (condition2) {  
        // both condition1 and condition2 are true  
    }  
}  
// is equivalent to the following  
if (condition1 && condition2) {  
    // both condition1 and condition2 are true  
}
```

对于并列使用很多 `if-else` 的情况，可以考虑使用 `switch-cases`，其完全形态如下：

```

switch (value) {
    case const_value_1:
        // value is const_value_1
    case const_value_2:
        // value is const_value_2
    default:
        // value is neither const_value_1 nor const_value_2
}

```

表示根据 `value` 的值跳转到相应的 `case` 所在行继续往下执行，如果没有匹配的 `case`，则跳转至 `default`（如果有，否则跳出 `switch`）。需要注意的是，一个 `case` 并不标识一个代码块，因此如果不想让程序从一个 `case` 执行到另一个 `case` 中，可以在上一个 `case` 末尾添加 `break;` 语句来跳出 `switch`。

3.3.3 循环结构

如果要将一段代码执行多次，就可以用循环结构来简化代码（虽然复制粘贴加 `if` 人肉展开也不是不行）。循环常常与数组搭配使用，用来用同一套逻辑处理大量数据。最常用的循环结构实现为 `for` 循环：

```

for (init; condition; expression) {
    // statement
}
// is equivalent to the following
{
    init;
    label:
    {
        if (condition) {
            // statement
            expression;
            goto label;
        }
    }
}

```

`goto` 的翻译似乎有些抽象，我们进行一些讲解。一般来说，`for` 循环小括号中的三个表达式依次是初始化循环变量（索引）、循环条件和修改循环变量。其执行过程为：首先执行初始化语句（第一个语句），之后判断是否满足循环条件（第二个表达式），若不满足则跳出循环，若满足则进入循环体，执行完循环体后执行修改循环变量，之后进行下一次循环（判断循环条件、执行循环体、执行修改索引为一次循环）。在循环体内，可以通过 `condition;` 语句来跳至下一次循环，通过 `break;` 语句来跳出循环。

`while` 循环也是常用的循环结构，如下：

```

while (condition) {
    // statement
}
// is equivalent to the following
label:
{
    if (condition) {
        // statement
        goto label;
    }
}

```

有了理解 `for` 循环对应的 `goto` 代码后，通过 `goto` 代码理解 `while` 循环也应是trivial的。

还有一类 `do-while` 循环，实践中并不常用，但为了完整性还是介绍一下：

```

do {
    // statement
} while (condition);
// is equivalent to the following
label:
{
    // statement
    if (condition) {
        goto label;
    }
}

```

不难发现其与 `while` 循环的区别仅在于条件判断和循环体的先后。

习题 3.3-1

编写一段 C++ 程序，使其接受输入一个 `int` 范围内的正整数 n ，输出1到 n （含）中所有3的倍数，一个一行。

3.4 函数

我们可以使用3.2节和3.3节的知识来实现各种功能，但当程序达到一定大小时，这往往会使得代码难以阅读和修改，成为屎山。因此，我们需要函数（function）来对代码进行分块、抽象。程序的入口 `main` 函数就是一个重要的函数，大部分代码都在它内部进行。

一个简单的函数示例如下：

```

int add(int x, int y) {
    return x + y;
}

```

其中 `add` 为函数名，其之前的 `int` 为返回值类型，其之后的 `int x, int y` 为参数列表，大括号内部为函数体。可以使用类似 `add(1, 2)` 的语法来调用此函数。

函数的执行流程为：遇到函数调用时，将参数（如果有）传递给参数列表中的参数，并进入函数体执行。遇到 `return` 时将返回值（如果有）传递回来，并回到调用处向下执行。

参数列表的语法与定义变量类似，多个参数中间用逗号分隔，但不能合并相同的类型（如 `int x, y`）。事实上，参数名对定义函数来说是无关紧要的。与变量类似，参数也可以有默认值，这时这个参数为缺省参数。缺省参数只能位于参数列表的末尾（也可以有多个），这时调用函数时可以不传入这个参数所对应的实际参数（其值就是定义的默认值）。参数列表中定义的参数为形式参数，调用函数时传入的参数为实际参数。函数体中只能对形式参数进行操作，如果需要修改传入的实际参数，可以使用引用类型。

函数被调用时返回的值的类型由返回值类型指定。特别地，如果一个函数不需要返回任何值，则可以使用空类型 `void`。通过 `return ...;` 在函数体内部结束函数并返回值，其中 `...` 可以是任何表达式。

函数体中可以调用任何函数，包括其自己。调用自己的函数为递归函数。如下面是一个利用递归函数计算阶乘的例子：

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

递归的进行并不是无限制的，这里需要引入调用栈（call stack）的概念。首先来了解栈：栈类似一摞积木，每块积木都是栈里的一个元素，并且栈只能从顶部进行操作，即只有拿走最上面的积木、往上加再一块积木两种操作。调用栈也是如此，简单来说，其中每个元素代表一个函数及其函数体中的局部变量。在程序运行时，操作系统负责维护这样一个栈，用于记录程序现在在哪个函数中执行。当元素数量达到一定限制时，就会溢出，俗称爆栈。（一般是达到编译器设定的限制，或是内存不足）

当递归函数的递归层数过多时，就有可能触发栈溢出，因此对于比较简单的代码，推荐使用循环等普通方法代替递归实现。（理论上所有递归代码都可以改写成只使用循环的普通方法，但过于复杂的递归代码会使人~~大脑升级~~大脑萎缩）

另一个典型的使用引用类型作为参数的例子是交换函数（顾名思义，该函数可以交换两个变量的值）：

```
void swap(int& x, int& y) {
    int t = x;
    x = y;
    y = t;
}
```

由于使用了引用类型，在函数体内部对 `x`、`y` 变量的操作就会反映到外部的作为实际参数传入的变量中。引用类型的另一好处为不需要把这个变量复制一份（函数体中使用的其实就是传入的变量），因此也常被用来优化。如果不想修改传入的变量同时不希望复制一份，可以使用常量引用（`const` 的引用）。

这里的 `swap` 函数只能对整数类型进行交换，如果我们想要交换浮点类型，则需要写另一个版本，方法也很简单，即将上面函数中的 `int` 改为 `double`。得益于C++的函数重载机制，这两个函数可以共用 `swap` 这个名称，并且在调用时编译器会根据传入参数的类型自动判断要调用的是哪个函数。（顺便剧透一下，在模板相关章节我们会写一个能对任意类型进行交换的 `swap` 函数）

有不同的参数列表（正如上文所说，参数列表的不同指的是参数类型的不同）的函数可以使用相同的函数名，这就是函数重载。函数重载带来的最大的好处在于方便了调用者，即只要是语义相同的参数类型都可以用同一个函数名进行调用，如C++标准库中的 `std::sin` 函数提供了 `float`、`double`、`long double` 三种类型的重载。

值得一提的是，运算符也是可以重载的，因为运算符本质上也是函数，但这只适用于自定义类型（将在面向对象编程相关章节详细介绍）。

3.5 面向对象初步

面向对象编程（OOP，Object-Oriented Programming）是一种重要的编程范式，它使得数据和处理数据的方法紧密结合

3.5.1 类

类是一系列变量和函数的集合体，也就是自定义类型，可以使用如下的语法来定义一个类：

```
class class_name {  
    // ...  
};
```

其中 `class` 也可以用 `struct` 代替，`// ...` 为任意多个变量定义和函数定义，类中的变量为成员变量，类中的函数为成员函数。

可以使用类型访问修饰符 `public:`、`private:` 和 `protected:` 来标记从该行开始到类定义结束范围内定义的变量和函数的可访问性，`public` 表示任何地方都可以访问，`private` 表示只能在该类的成员函数中访问，`protected` 表示可以在该类的成员函数和派生类的成员函数中访问。

`class` 和 `struct` 的唯一区别在于默认的可访问性（即不写类型访问修饰符时的可访问性）：`class` 为 `private`，`struct` 为 `public`。

习惯上使 `struct` 只包含成员变量而不包含成员函数，且不修改它的可访问性（即保持默认的 `public`）。

构造函数和析构函数是类中的特殊的成员函数，构造函数名与类名相同，析构函数名为类名前加上 `~`，构造函数在创建一个该类型的变量时被调用，析构函数在类的实例被销毁时被调用，构造函数和析构函数都没有返回值，并且析构函数不能有参数。

可以通过成员访问运算符 `.` 来访问类的成员变量和成员函数，也可以使用 `->` 在访问时进行解引用，如 `x->y` 等价于 `* (x.y)`。

`this` 为类中一个特殊的指针，它指向该类的实例自己。当局部变量和类中的成员变量重名时，可以通过 `this->` 的方式访问成员变量，而直接使用变量名访问局部变量。

创建一个该类型的变量称为类的实例化，这个变量为该类的实例，实例化时可以向构造函数传递参数，

为了更好地理解，我们以如下的类为例子：

```
class Point {  
public:  
    int x, y;  
    Point() {  
        x = 0;  
        y = 0;  
    }  
    Point(int x, int y) {  
        this->x = x;  
        this->y = y;  
    }  
    void set_x(int x) {  
        this->x = x;  
    }  
};
```

```

    int get_x() {
        return x;
    }
    // ...
    ~Point() {
        // ...
    }
};

int main() {
    Point point(1, 2);
    std::cout << point.get_x() << '\n';
    return 0;
}

```

该程序的输出应为 1。其中 `Point point(1, 2);` 调用了第二个构造函数，如果使用 `Point point;` 则会调用无参数的构造函数（即默认构造函数），也可以使用 `Point point = Point(1, 2);`。

由于特殊构造函数过于复杂，且牵涉到右值引用，这里暂时不介绍。

3.5.2 继承

面向对象程序设计中最重要的一個概念是继承。继承允许我们依据另一个类来定义一个类，这使得创建和维护一个应用程序变得更容易，这样也达到了重用代码功能和提高执行效率的效果。当创建一个类时，不需要重新编写新的成员变量和成员函数，只需指定新建的类继承了一个已有的类即可。这个已有的类称为基类，新建的类称为派生类。继承的语法为 `[class|struct] 派生类名: 访问修饰符 基类名`，其中访问修饰符为 `public`、`protected` 或 `private`，例如：

```

class Base {
};
class Derived : public Base {
};

```

这时 `Derived` 类会继承 `Base` 类的所有成员变量和成员函数，除了当使用不同访问修饰符的继承时，遵循以下几个规则：

- 公有继承 (`public`)：当一个类派生自 `public` 基类时，基类的 `public` 成员也是派生类的 `public` 成员，基类的 `protected` 成员也是派生类的 `protected` 成员，基类的 `private` 成员不能被派生类访问，但是可以通过调用基类的 `public` 和 `protected` 成员来访问。
- 保护继承 (`protected`)：当一个类派生自 `protected` 基类时，基类的 `public` 和 `protected` 成员将成为派生类的 `protected` 成员。
- 私有继承 (`private`)：当一个类派生自 `private` 基类时，基类的 `public` 和 `protected` 成员将成为派生类的 `private` 成员。

`class` 的默认继承为 `private`，`struct` 为 `public`。

3.5.3 运算符重载

我们可以在类中重定义或重载大部分运算符，这样就可以对自定义类型使用运算符。重载运算符与定义函数几乎相同，只有函数名变成了 `operator` 加运算符。运算符的参数列表是特定的，与运算符种类和运算符重载的位置有关。例如，在类外部定义的加号 `+` 运算符需要两个参数，在类里定义的加号 `+` 运算符只要一个参数（另一个就是该类本身的实例）。

例如，对于上面的 `Point` 类，可以这样重载加号 `+` 运算符：

```
class Point {
// ...
public:
    Point operator+(const Point& rhs) {
        return Point(x + rhs.x, y + rhs.y);
    }
};
```

或

```
// outside the class definition
Point operator+(const Point& lhs, const Point& rhs) {
    return Point(lhs.x + rhs.x, lhs.y + rhs.y);
}
```

事实上，更常见的情况是我们会把 `x` 和 `y` 设为 `private`，这时为了在 `public` 成员函数中访问到 `private` 成员变量，我们需要将其设为友元函数：

```
class Point {
// ...
public:
    friend Point operator+(const Point& lhs, const Point& rhs)
};
Point operator+(const Point& lhs, const Point& rhs) {
    return Point(lhs.x + rhs.x, lhs.y + rhs.y);
}
```

-1. 结语

编程是一项需要不断学习和实践的技能。希望读者能够通过本书掌握编程的基本技能，并在未来的学习和工作中有所收获。

-1.1 编者的话

在编写本书的过程中，参考了大量优秀的编程教材和资源。同时也结合了自己的学习经验，希望能够为读者提供一本系统、实用、易学的编程教材。

在本书的编写过程中，编者得到了许多人的帮助，在此向他们表示衷心的感谢。

最后，祝愿读者在编程学习的道路上取得成功！