《C++ 语言程序设计》简明讲义*

潘建瑜

华东师范大学 • 数学科学学院

2025 年 9 月

*本讲义仅供课堂教学使用

https://math.ecnu.edu.cn/~jypan

纸上得来终觉浅，绝知此事要躬行。

When the terrain disagrees with the map, trust the terrain.

学而不思则罔，思而不学则殆。

目 录

第一讲 计算机基础 **1**

1.1 信息的表示与存储 1

1.1.1 计算机的数字系统 1

1.1.2 常见的数制及它们之间的转换 1

1.1.3 二进制数的编码表示 2

1.2 算法 4

1.2.1 算法的特征与评价 4

1.2.2 算法的描述方法与基本控制结构 4

1.3 课后练习 5

第二讲 C++ 编程基础 **6**

2.1 C++ 语言概述 6

2.1.1 C++ 源程序结构 7

2.1.2 C++ 源程序书写规范 8

2.1.3 程序编译 8

2.2 C++ 基础知识 8

2.2.1 C++ 字符集, 标识符, 关键字 8

2.2.2 C++ 数据类型 9

2.2.3 变量与常量 12

2.2.4 类型推导: auto 与 decltype 13

2.2.5 类型别名: typedef 与 using 14

2.2.6 枚举 14

2.2.7 运算与表达式 14

2.2.8 常用数学函数 16

2.3 C++ 简单输入输出 16

2.3.1 输入输出语句 16

2.3.2 操纵符 17

2.4 程序示例 17

2.5 上机练习 18

第三讲 选择与循环 **19**

3.1 关系运算与逻辑运算 19

3.2 选择结构 20

3.2.1 IF 语句 . . . 20
3.2.2 SWITCH 结构 . . . 20
3.3 循环结构 . . . 21
3.3.1 WHILE 循环 . . . 21
3.3.2 DO WHILE 循环 . . . 22
3.3.3 FOR 循环 . . . 22
3.3.4 基于范围的 for 循环 . . . 23
3.3.5 循环的非正常终止 . . . 23
3.4 程序示例 . . . 24
3.5 应用: 定积分的数值计算 . . . 25
3.6 上机练习 . . . 25

第四讲 函数 **27**
4.1 函数的声明、定义与调用 . . . 27
4.2 函数间的参数传递 . . . 30
4.3 内联函数 . . . 31
4.4 函数嵌套与递归 . . . 31
4.5 变量的作用域 . . . 32
4.6 形参带缺省值 . . . 34
4.7 函数重载 . . . 35
4.8 预编译处理与多文件结构 . . . 35
4.8.1 预编译 . . . 35
4.8.2 多文件结构 . . . 37
4.9 应用: 蒙特卡罗算法 . . . 38
4.10 上机练习 . . . 39

第五讲 数组 **42**
5.1 一维数组 . . . 42
5.2 二维数组 . . . 43
5.3 数组作为函数参数 . . . 44
5.4 字符串（字符数组） . . . 45
5.4.1 字符数组 . . . 45
5.4.2 字符操作 . . . 47
5.5 上机练习 . . . 48

第六讲 指针 **50**
6.1 指针与内存 . . . 50
6.2 指针与一维数组 . . . 52
6.3 指针与二维数组 . . . 53
6.4 行指针与二级指针 * . . . 55

6.4.1 行指针 . . . 55
6.4.2 二级指针 . . . 55
6.5 指针与函数 . . . 56
6.6 持久动态内存分配 . . . 57
6.6.1 智能指针 . . . 58
6.7 应用: 矩阵乘积的快速算法 . . . 59
6.8 应用: Gauss 消去法求解线性方程组 . . . 59
6.9 上机练习 . . . 59

第七讲 简单输入输出 61
7.1 C++ 基本输入输出 (I/O) 流 . . . 61
7.2 C 语言格式化输出 . . . 62
7.3 C 语言文件读写 . . . 63
7.3.1 文件的打开和关闭 . . . 63
7.3.2 文本文件的读写 . . . 64
7.3.3 二进制文件的读写 . . . 64
7.3.4 其他文件操作 . . . 65
7.4 上机练习 . . . 65

第八讲 排序算法及其 C++ 实现 67
8.1 引言 . . . 67
8.2 选择排序 . . . 68
8.3 插入排序 . . . 68
8.4 希尔排序 . . . 68
8.5 冒泡排序 . . . 69
8.6 快速排序 . . . 69
8.7 归并排序 . . . 70
8.7.1 算法基本思想 . . . 70
8.7.2 算法实现过程 . . . 71
8.8 上机练习 . . . 72

第九讲 类与对象基础 I 73
9.1 为什么面向对象 . . . 73
9.2 类和对象基本操作 . . . 74
9.3 构造函数 . . . 78
9.4 复制构造函数 . . . 79
9.5 匿名对象 . . . 80
9.6 类与对象举例: 游泳池 . . . 80
9.7 析构函数 . . . 80
9.8 上机练习 . . . 81

第十讲 类与对象基础 II **83**
10.1 类的组合 . . . 83
10.2 结构体与联合体 . . . 86
10.3 类作用域 . . . 86
10.4 静态成员 . . . 87
10.5 友元关系 . . . 88
10.6 类的 UML 描述 * . . . 89
10.7 上机练习 . . . 90

第十一讲 类与对象基础 III **91**
11.1 常对象与常成员 . . . 91
11.2 对象数组与对象指针 . . . 92
11.3 动态对象 . . . 94
11.4 数组类: array . . . 94
11.5 向量类: vector . . . 95
11.6 字符串类: string . . . 98
11.7 上机练习 . . . 101

第十二讲 运算符重载与自动类型转换 **103**
12.1 为什么要重载运算符 . . . 103
12.2 怎么实现运算符重载 . . . 104
12.3 运算符重载: 成员函数方式 . . . 104
12.4 运算符重载: 非成员函数方式 . . . 105
12.5 重载赋值运算 = . . . 106
12.6 左值与运算符 [ ] 的重载 . . . 107
12.7 自动类型转换 . . . 108
12.8 上机练习 . . . 108

第十三讲 继承与派生 **111**
13.1 继承与派生 . . . 111
13.2 派生类的定义 . . . 111
13.3 派生类构造函数 . . . 113
13.4 派生类的复制构造函数 . . . 114
13.5 派生类的析构函数 . . . 114
13.6 同名成员屏蔽规则 . . . 114
13.7 类型兼容规则 . . . 115
13.8 虚继承 . . . 115
13.9 上机练习 . . . 116

第十四讲 多态 **119**

14.1 什么是多态 . . . 119
14.2 虚函数 . . . 119
14.3 纯虚函数与抽象类 . . . 121
14.3.1 纯虚函数 . . . 121
14.3.2 抽象类 . . . 121
14.3.3 举例 . . . 122
14.4 模板 . . . 122
14.5 模板函数 . . . 122
14.6 模板类 . . . 123
14.7 上机练习 . . . 125

第十五讲 文件流与输出输入重载 **127**
15.1 输入输出流 . . . 127
15.2 文件流类与文件流对象 . . . 127
15.3 文件的打开与关闭 . . . 128
15.4 文件读写: 文本文件与二进制文件 . . . 129
15.5 移动或获取文件读写指针 . . . 130
15.6 重载 << 和 >> . . . 131
15.7 上机练习 . . . 131

第十六讲 标准模板库 **133**
16.1 STL 标准模板库 . . . 133
16.2 容器 . . . 133
16.3 算法 . . . 135
16.4 迭代器 . . . 137

主要参考文献 **138**

第一讲 计算机基础

本讲主要内容

- 数制: 二进制、八进制、十进制和十六进制, 二进制与十进制的相互转换
- 二进制的表示形式: 原码, 反码, 补码
- 算法基本概念, 算法的特征与评价, 算法的描述方法与基本控制结构

1.1 信息的表示与存储

1.1.1 计算机的数字系统

(1) 计算机内部信息的分类:

- **控制信息**: 指令集, 负责软硬件之间的交互, 如: X86, ARM, RISC-V;
- **数据信息**: 包括数值信息和非数值信息, 其中数值信息包括整数、浮点数等, 非数值信息包括字符（字符串）和逻辑数据等.

(2) 信息的存储单位:

- 信息存储的基本单位有: 二进制位（bit, 简记 b）和字节（Byte, 简记 B）;
- 计算机的最小存储单元是字节, 一个字节由 8 个二进制位组成, 即 1B=8b;
- 其它存储单位有: KB, MB, GB, TB, PB, EB 等等;
- 一个英文字符占一个字节, 而一个汉字占两个字节.

(3) 计算机数字系统:

- 采用的是二进制数字系统, 基本符号是 0 和 1;
- 优点: 易于物理实现、运算简单、可靠性高、通用性强;　缺点: 可读性差.

1.1.2 常见的数制及它们之间的转换

(1) 常见数制: 二进制、八进制、十进制和十六进制.

(2) 二进制转十进制: 各位数字与它的权相乘, 然后相加.

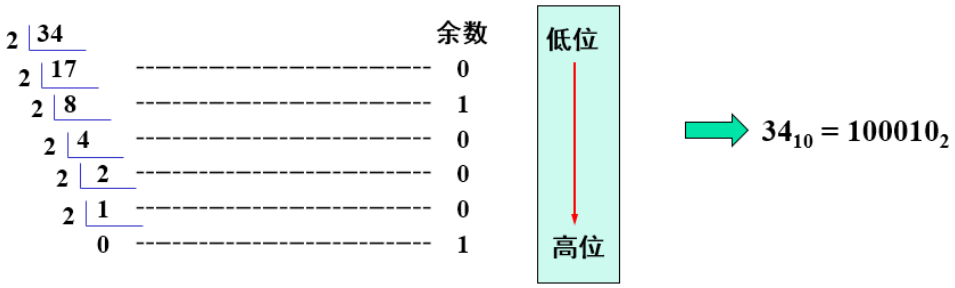
例 1.1 (二进制转十进制)

$$(101.11)_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^{-1} + 1 \times 2^{-2} = (5.75)_{10}$$

✍ 八进制和十六进制转化为十进制的方法类似.

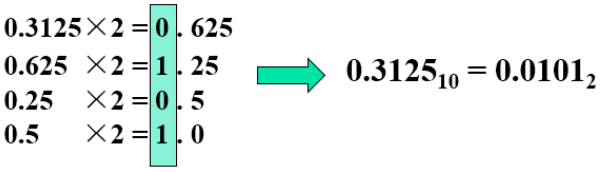
(3) 十进制转二进制

例 1.2 (十进制整数转化为二进制整数) : “除 2 取余, 逆序排列”, 即辗转相除法.



✍ 十进制整数转化为八进制或十六进制整数的方法类似.

例 1.3 (十进制纯小数转化为二进制纯小数) : “乘 2 取整, 顺序排列”, 即每次乘以 2 后去掉整数部分, 不断乘下去, 直到小数部分为 0 或达到指定的精度为止, 然后取每次相乘后的整数部分即可.



✍ **注记**: 需要指出的是, 绝大部分浮点数是无法用二进制数来精确表示的, 如 0.1, 0.2, 0.3, 0.4, 0.6, 0.7, 0.8, 0.9, 因此, 计算机中存储的浮点数基本上都是近似数.

(4) 二进制与八进制、二进制与十六进制之间的互换

- 每位八进制数对应于一个三位二进制数（见下表左）;
- 每位十六进制数对应于一个四位二进制数（见下表右）.

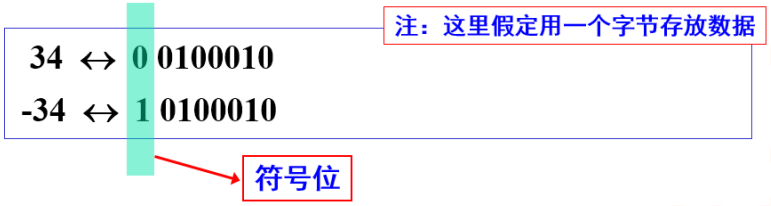
0 ↔ 000	
1 ↔ 001	
2 ↔ 010	
3 ↔ 011	
4 ↔ 100	
5 ↔ 101	
6 ↔ 110	
7 ↔ 111	

0 ↔ 0000	8 ↔ 1000
1 ↔ 0001	9 ↔ 1001
2 ↔ 0010	A ↔ 1010
3 ↔ 0011	B ↔ 1011
4 ↔ 0100	C ↔ 1100
5 ↔ 0101	D ↔ 1101
6 ↔ 0110	E ↔ 1110
7 ↔ 0111	F ↔ 1111

1.1.3 二进制数的编码表示

(1) 数在计算机内部的存储方式: 原码、反码、补码.

- 数的表示: 符号 + 大小. 用二进制表示带符号的数, 首位为符号位, “0” 表示正, “1” 表示负.
- 优点: 直观;　缺点: 进行四则运算时要考虑符号位, 规则复杂, 另外零的表示也不唯一.



(2) 反码

- 正数的反码: 与原码相同;
- 负数的反码: 符号位不变, 其它位取反（0 变 1, 1 变 0）

	原码	反码
34 ↔	0 0100010 ↔	0 0100010
-34 ↔	1 0100010 ↔	1 1011101

> ✍ **注记：** 反码一般不直接使用, 通常是作为求补码的中间码.

(3) 补码

- 正数的补码: 与原码相同;
- 负数的补码: 反码的最末位加 1;

	原码	反码	补码
34 ↔	0 0100010 ↔	0 0100010 ↔	0 0100010
-34 ↔	1 0100010 ↔	1 1011101 ↔	1 1011110

- 0 的补码表示唯一 (补码 `1 000 0000` 表示 `-128`), 因此可以多表示一个数;
- 补码的补码就是原码.

> ✍ **注记：** 数据在计算机中是以补码的方式存放的.

(4) 补码运算规则

- 符号位作为数值直接参加运算;
- 减法转化为加法进行运算;
- 运算结果仍为补码.

例：用 8 位字长计算 67 - 10

67_{10} = 01000011$_2$[原码] ↔ 01000011[补码]

-10_{10} = 10001010$_2$[原码] ↔ 11110101[反码] ↔ 11110110[补码]

01000011 + 11110110 = 1 00111001 ↔ 00111001 [补码] ↔ 00111001$_2$[原码] = 57_{10}

符号位加入正常运算，超出字长部分自然丢失

> **思考：** 如果用 8 位字长计算 85 + 44, 则结果会是什么？

(5) 非数值信息

- 西文字符: 每个西文字符与其 ASCII 码一一对应
- 中文汉字: 一个汉字占两个字节, 常见编码有 GB2312, GBK, UTF-8 等.

1.2 算法

程序 = 算法 + 数据结构 + 程序设计方法 + 语言工具和环境

—— 计算机科学家 Nikiklaus Wirth，1976

(1) 一个程序应该包括:

- 对数据组织的描述: 数据的类型和组织形式, 即**数据结构**;
- 对操作流程的描述: 即操作步骤, 也就是**算法**.

(2) 算法: 通俗地说, 算法就是为解决一个问题而采取的方法和具体步骤.

- 学习程序设计的目的不仅仅是学习编程语言, 而是学习程序设计的**一般方法**;
- 掌握了算法就是掌握了**程序设计的灵魂**, 再配合编程语言, 就能顺利写出程序, 解决问题;
- 脱离了具体的编程语言去学习程序设计是困难的.

1.2.1 算法的特征与评价

(1) 算法的特征

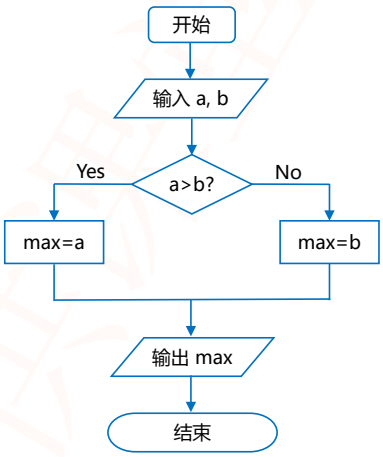
- 输入: 有零个或多个输入量;
- 输出: 通常有一个或以上输出量（计算结果）;
- 明确性: 算法的描述必须无歧义, 保证算法的正确执行;
- 有限性: 有限个输入、有限个指令、有限个步骤、有限时间;
- 有效性: 又称可行性, 能够通过有限次基本运算来实现.

(2) 算法性能的评测

- 时间复杂度: 运算量
- 空间复杂度: 内存资源占用量
- 实现复杂度: 编程实现与维护

1.2.2 算法的描述方法与基本控制结构

(1) 算法的描述方法: 自然语言, 流程图（见下图左）, 伪代码（见下图右）等.



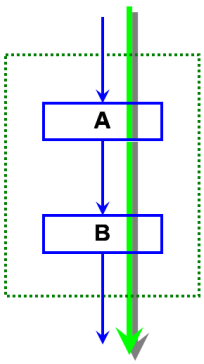
算法 1.2 LU 分解

```
1:  for k = 1 to n − 1 do
2:      for i = k + 1 to n do
3:          l_ik = a_ik / a_kk      % 计算 L 的第 k 列
4:      end for
5:      for j = k to n do
6:          u_kj = a_kj             % 计算 U 的第 k 行
7:      end for
8:      for i = k + 1 to n do
9:          for j = k + 1 to n do
10:             a_ij = a_ij − l_ik u_kj     % 更新 A(k+1:n, k+1:n)
11:         end for
12:     end for
13: end for
```

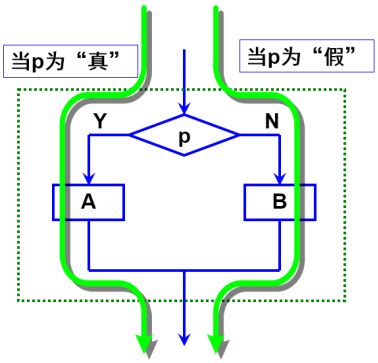
注记： 流程图: 简洁、直观、准确； 伪代码: 易于编程实现.

(2) 算法的三种基本控制结构: 顺序结构, 选择结构, 循环结构.

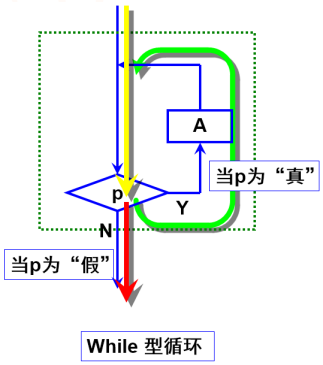
- 顺序结构是最基本的程序设计结构, 它按照语句行的自然顺序, 一条一条地执行程序;
- 选择结构, 又称分支结构或条件结构, 可根据条件, 判断应该选择哪一条分支来执行;
- 循环结构, 可根据给定的条件, 判断是否需要重复执行某一相同的程序段.



顺序结构



选择结构



循环结构

注记： 三种结构的共同点: (1) 只有一个入口; (2) 只有一个出口; (3) 结构内每一部分都有机会被执行; (4) 结构内不存在“死循环”.

1.3 课后练习

练习 1.1 将下列二进制数转化为十进制数:

$$101, \quad 100111, \quad 11010.011$$

练习 1.2 将下列十进制数转化为二进制数:

$$101, \quad 0.5625, \quad 93.328125$$

练习 1.3 如何计算小数的补码.

第二讲 C++ 编程基础

本讲主要内容

- C++ 语言概述
  - C++ 语言的发展
  - C++ 源程序结构与书写规范
  - C++ 编译器和集成开发环境
- C++ 编程基础
  - C++ 字符集: 标识符, 关键字
  - C++ 数据类型与类型转换
  - 变量、常量、符号常量
  - 运算符、运算优先级
  - 语句与表达式
- 数据的简单输入输出
  - 输出: `cout`, 输入: `cin`
  - 操纵符: 控制输出的格式

C++ 是一门广泛使用的面向对象编程语言, 其语法丰富, 且灵活高效, 是当前高性能科学计算和大型软件开发的首选编程语言. 但 C++ 同时也是一门不易掌握的编程语言, 尤其是使用其高级特性时, 需要深入了解它的底层.

学习编程是一个循序渐进的过程, 通过本课程, 希望大家能:

- 掌握 C++ 的基本语法规则和面向对象的基本知识, 熟练阅读和分析简单的 C++ 程序源代码;
- 掌握算法设计基本思想和方法, 培养面向对象的程序设计能力, 掌握基本的编程和调试技术;
- 培养编程兴趣, 了解内存、编译和链接相关知识, 以及编程语言的内在机理.

学习 C++, 不仅要理解 C++ 编程的基本概念, 掌握语言的技术细节, 更重要的是培养编程思维.

2.1 C++ 语言概述

C++ 是从 C 语言发展演变而来, 可以看成是 C 的超集[1]. 1979 年, 当时在美国 AT&T 公司贝尔实验室工作的 Bjarne Stroustrup 博士开始开发一门新语言, 当时称为 "C with Classes", 目的是将面向对象编程技术添加到 C 语言中. 1983 年, 在 Rick Mascitti 的建议下, 正式取名为 C++ （或 CPlusPlus, 在 C 语言中, ++ 运算符代表变量自增 ）. 1985 年, 第一个 C++ 商业产品发布, 即 CFront Release 1.0. 同年 10 月, Bjarne Stroustrup 完成了经典巨著 "The C++ Programming Language" 第一版. 1987 年 12 月, GNU C++ 发布.

[1] Bjarne Stroustrup 在其著作 "The C++ Programming Language" 的第一版序言中说: "Except for minor details, C++ is a superset of the C programming language." 有学者可能持不同的观点, 但这不影响 C++ 的学习.

1989 年, “The Annotated C++ Reference Manual”发布, 成为 C++ 标准的基础. 同年 12 月, ANSI (American National Standards Institute, 美国国家标准协会) 开始 C++ 的标准化工作. 1992 年 3 月, 第一个 Microsoft C++ 发布, 同年 5 月, 第一个 IBM C++ 发布. 1994 年, ANSI C++ 标准草案出台, 并于 1995 年提交公众审阅. 1998 年, ISO (International Organization for Standardization, 国际标准化组织) 批准其为国际标准, 发布了 C++ 的第一个国际标准, 通称 C++98 .

2011 年 8 月, C++11 标准发布, 增加了多线程支持、通用编程支持等, 标准库也有很多变化. 2014 年 8 月, C++14 标准发布, 对 C++11 进行小范围扩展, 主要是修复 bug 和提高性能. 随后, C++ 标准委员会每隔 3 年就会发布新的 C++ 标准, 如 C++17, C++20 等. 关于 C++ 语言的最新标准可参见 http://isocpp.org/std/status.

主要参考资料

- Y.D. Liang 著, 刘晓光等译, “C++ 程序设计” (第 3 版), 2015.
- S. Prata 著, 张海龙等译, “C++ Primer Plus” (第 6 版), 2012.
  (侧重于 C++ 语言的语法与基础, 适合初学者)
- S.B. Lippman 等著, 王刚等译, “C++ Primer” (第 5 版), 2013.
  (侧重于 C++ 语言的语法细节, 适合初学者当字典查阅)
- B. Stroustrup 著, “C++ 程序设计原理与实践” (第 2 版), 任明明等译, 2017; 张兴等译, 2024.
  (侧重于程序设计的思想和方法, 适合有一定编程经验的读者)
- B. Stroustrup 著, 王刚等译, “C++ 程序设计语言” (第 4 版), 2016.
  - https://cppreference.cn, C++ 语言参考手册, 含标准库.

2.1.1 C++ 源程序结构

- C++ 源程序由一个或多个源文件组成;
- 每个源文件可由一个或多个函数组成, 函数是 C++ 语言的基本模块;
- 一个源程序有且只能有一个 main 函数, 称为 **主函数**;
- 程序执行从 main 开始, 在 main 中结束;
- 源程序中可以有预处理命令 (以 “#” 开头) , 比如 #include;
- 一行可以写多个语句, 一个语句也可以分几行书写.

```
1 #include <iostream> // 预处理命令, 载入头文件
2 using namespace std; // 使用标准命名空间, 即导入其中声明的所有命令、函数等
3 int main() // 主函数
4 {
5     cout << "Hello" << endl; // 标准输出
6     cout << "Wellcome to C++!" << endl;
7     return 0;
8 }
```

☞ 代码中的 “//” 是注释符.

2.1.2 C++ 源程序书写规范

- 每条语句以分号“;”结尾, 但预处理命令, 函数头和右花括号“}”之后一般无需加分号;
- 标识符、关键字之间至少加一个空格表示间隔, 若已有明显的间隔符, 也可不加;
- C++ 区分大小写;

注记： 所有标点符号必须在英文状态下输入!

书写漂亮的程序: 使用锯齿形书写格式

(1) 花括号 { } 要对齐;

(2) 一行写一个语句, 一个语句写一行;

(3) 使用 TAB 缩进;

(4) 有合适的空行;

(5) 有足够的注释.

2.1.3 程序编译

- 编译器: 将“高级语言”翻译为“机器语言”的工具.
- 一个现代编译器的主要工作流程:

$$\text{源代码} \xrightarrow{\text{编译}} \text{目标代码} \xrightarrow{\text{链接}} \text{可执行程序}$$

- 常见的 C++ 编译器: GNU C++, Visual C++, Intel DPC++, NV C++, Clang 等 (后三个都是基于 LLVM 架构).
- 常用的 IDE (Integrated Development Environment, 集成开发环境):
  (1) Dev C++ : 小巧免费, Windows 平台上的 gcc, 适合学习 C++.
  (2) Visual Studio Code : 免费的轻量级可扩展代码编辑器, 支持 Windows、Linux 和 MacOS.
  (3) Visual Studio : Windows 平台上大型 C++ 集成开发环境.

LLVM, Low Level Virtual Machine

LLVM 是一个编译器基础设施项目 (与传统意义的虚拟机无关), 如今已演变为一个模块化、可重用的编译器工具链框架, 旨在为各种编程语言和硬件架构提供高效的编译、优化和代码生成能力. 其核心是通过统一的“中间表示, Intermediate Representation”连接编译器的各个阶段 (前端、优化器、后端), 使得不同语言的前端和不同硬件的后端可以灵活组合, 大幅降低了为新语言或新硬件开发编译器的成本. LLVM 革新了传统编译器的设计模式, 构建了一个灵活高效的编译器生态, 成为硬件厂商和工具开发者构建编译工具链的首选框架, 推动了编程语言和计算机体系结构的创新.

2.2 C++ 基础知识

2.2.1 C++ 字符集, 标识符, 关键字

- 合法的字符集有

(1) 字母: 包括大写和小写, 共 52 个;
(2) 数字: 0 到 9 共 10 个;
(3) 空白符: 空格符、制表符、换行符;
(4) 标点符号和特殊字符:

```
+ - * / = ! # % ^ & ( ) [ ] { } _ ~ < > \ ' " : ; . , ?
```

- C++ 标识符: 用来标识变量名、函数名、对象名等的字符序列.
  (1) 由字母、数字、下划线组成, 第一个字符必须是字母或下划线;
  (2) 区分大小写, 不能用关键字;
  (3) C++ 不限制标识符长度, 实际长度与编译器有关, 建议不要超过 32 个字符;
  (4) 命名原则: 见名知意, 不宜混淆.
- C++ **关键字**: 具有特定意义的字符串, 也称为 **保留字**, 包括: 类型说明符（也称类型标识符）、语句定义符（控制命令）、编译预处理命令等;

表 2.1. C++ 关键字（部分）

bool	break	case	catch	char	class	const
continue	default	delete	do	double	else	enum
extern	false	float	for	friend	goto	if
inline	int	long	namespace	new	operator	private
protected	public	return	short	signed	sizeof	static
struct	switch	template	this	true	try	typedef
typename	union	unsigned	using	virtual	void	while

† 全部关键字可参见 http://en.cppreference.com/w/cpp/keyword

- C++ 分隔符: 逗号、冒号、分号、空格、( )、{ }
- C++ 注释: 有两种注释方式, 分别是:
  (1) 单行注释: //
  (2) 块（多行）注释: /* */

2.2.2 C++ 数据类型

C++ 的数据类型可分为基本数据类型（fundamental types, 有时也称原生数据类型）和复合（compound types，也称派生或扩展）数据类型, 见图 2.1 和表 2.2.

- 基本数据类型包括: 整型, 实型, 字符型（char）和布尔型（bool）.
- 复合（派生、扩展）数据类型: 引用, 指针, 数组, 字符串, 枚举, 类, 结构, 联合, 等等.
- 其他: void, long long, long doulbe, size_t, wchar_t, char16_t, char32_t, 等.
- 可以通过头文件引入更多数据类型, 比如头文件 ctime 中的 clock_t, 头文件 cstdint 中的 int8_t, uint8_t 等.

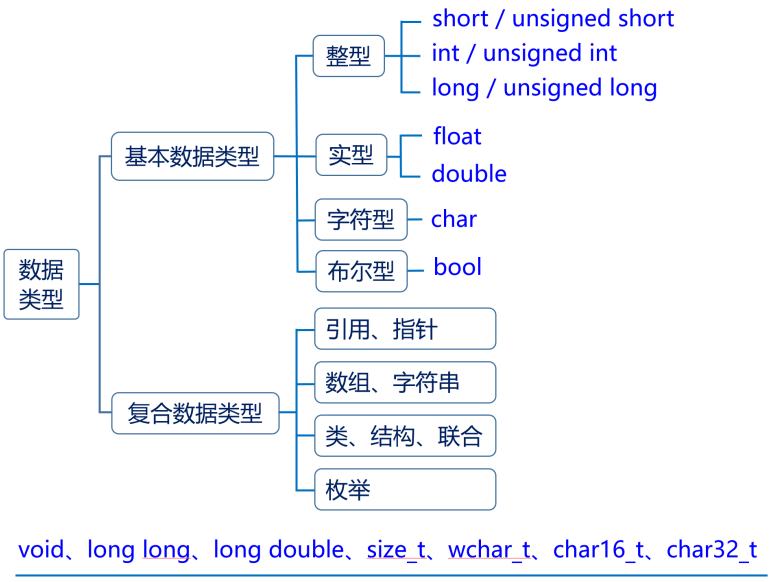


图 2.1. C++ 数据类型

表 2.2. C++ 基本数据类型

数据类型	类型标识符/类型说明符	所占字节	表示范围
整型	`short`	2	$-2^{15} \sim 2^{15}-1$
	`int`	2 / 4	$-2^{15} \sim 2^{15}-1$ / $-2^{31} \sim 2^{31}-1$
	`long`	4 / 8	$-2^{31} \sim 2^{31}-1$ / $-2^{63} \sim 2^{63}-1$
	`unsigned short`	2	$0 \sim 2^{16}-1$
	`unsigned int`	2 / 4	$0 \sim 2^{16}-1$ / $0 \sim 2^{32}-1$
	`unsigned long`	4 / 8	$0 \sim 2^{32}-1$ / $0 \sim 2^{64}-1$
实型	`float`	4 (6-7 位有效数字)	$10^{-38} \sim 10^{38}$
	`double`	8 (15-16 位有效数字)	$10^{-308} \sim 10^{308}$
布尔型	`bool`	1	`true`, `false`
字符型	`char`	1	

✍ **注记：** 事实上, C++ 标准没有规定每种数据类型的具体字节数和表示范围, 只规定大小顺序, 即长度满足下面的关系式

```
char <= short <= int <= long <= long long
```

具体长度由处理器和编译器决定.

✍ C/C++ 标准库中的 `size_t` 表示无符号整数 (64 位系统中为 `long unsigned int`), 通常用于计数和内存管理, 比如对象的字节数、数组下标等, `sizeof` 返回的结果就是 `size_t` 类型.

数据类型的转换

相同类型的数据才能直接运算, 运算结果为同类型数据. 如 `3/2` 的结果是 `1`. 不同类型的数据进行运算时, 需要根据相关的规则转化为同一类型的数据后才能进行运算.

数据类型的转换可以分为自动转换和强制转换.

(1) 自动转换（也称隐式转换）:

- 转换按数据长度增加的方向进行, 以尽可能地保证精度不会降低;
- 所有的浮点运算都是以双精度进行的;
- 赋值号两边的数据类型不同时, 需先将右边表达式的值转换为左边变量的类型, 然后再赋值;
- 字符变量直接参与算术运算时, 先转化位相应的 ASCII 码, 然后进行运算.

```
char --> short --> int --> unsigned --> long
     --> unsigned long --> double <-- float
```

(2) 强制转换（也称显式转换）: 三种实现方式

```
类型说明符(表达式) // C++风格, 将表达式的值转化为指定的数据类型
(类型说明符)表达式 // C 语言风格, 作用同上
static_cast<类型>(表达式) // C++, 静态强制类型转换
```

例 2.1 (类型转换) 类型转换示例. (ex02_datatype_conversion.cpp)

```
1 int a = 2, b = 5;
2 double x, y, z;
3 x = b/a;       // x = 2.0
4 y = double(b)/a; // y = 2.5
5 z = double(b/a); // z = 2.0
6 z = static_cast<double>(b)/a; // z = 2.5, 同 double(b)/a
```

(1) 由于 `a` 和 `b` 都是整型, 因此表达式 `b/a` 的值也是整型, 即 `b/a=2`. 将其赋值给 `double` 型变量 `x`, 因此 `x=2.0`.

(2) 在计算 `double(b)/a` 时, 先将 `b` 的值转化为 `double` 型 (注意, 不是将 `b` 转化为 `double` 型, 变量 `b` 的类型是不会改变的!), 然后与 `a` 相除. 由于是一个 `double` 型的数据与一个 `int` 型的数据进行运算, 所以系统会自动将 `int` 型的数据转化为 `double` 型的数据, 然后再做运算. 所以最后的结果是 `2.5`.

(3) 在计算 `double(b/a)` 时, 是先计算 `b/a`, 然后将结果转化为 `double` 型, 所以最后的结果是 `z=double(2)=2.0`

(4) 需要注意的是, 变量 `a`, `b` 的类型在整个计算过程中是始终不变的, 即一直是 `int` 型.

✍ **注记:** 类型转换是临时性的, 类型转换不会改变变量本身的数据类型!

(5) 类型转换规则:

- 浮点型转整型: 直接丢掉小数部分;
- 字符型转整型: 取字符的 ASCII 码;
- 整型转字符型: ASCII 码对应的字符.

例 2.2 类型转换示例. (ex02_datatype_conversion.cpp)

2.2.3 变量与常量

(1) 变量: 存储数据, 值可以改变

- 变量名: 命名规则与标识符相同
- 变量必须先声明, 赋值后才能使用.
- 变量声明:

```
类型说明符  变量名列表;
```

  - 可以同时声明多个变量, 用逗号隔开
  - 变量声明时可以初始化: 三种方式 (赋值运算符, 小括号和大括号) , 例如

```
1 int i = 20;
2 int j(2);
3 int k{4}; // C++11
4 float x = i/2.0 + 3.14; // 初始化时也可以用表达式
```

(2) 常量 (也称字面值常量): 在程序运行中值不能改变的量

- 整型常量: 整数, 后面加 `l` 或 `L` 表示长整型, 后面加 `u` 或 `U` 表示无符号整型, 后面加 `ll` 或 `LL` 表示 `long long`;
- 实型常量: 缺省为双精度实数, 后面加 `f` 或 `F` 表示单精度, 加 `l` 或 `L` 表示 `long double`
- 字符型常量: 用**单引号**括起来的单个字符和转义字符;
- 字符串常量: 用**双引号**括起来的字符序列;
- 布尔常量: `true` 和 `false`.

例: 下面都是常量

```
1 123, -456, 123L, 456U;
2 1.2, 1.2F, 1.2L, 1.2e8, 1.2e8F, 1.2e-8L
3 'M', 'A', 'T', 'H', '?', '$'
4 "MATH@ECNU"
```

(3) 符号常量: 用标识符表示常量, 在声明变量时加上关键字 const.

```
const 类型说明符 变量名 = 表达式;
```

```
1 const float PI = 3.1415926; // 可以用常数初始化符号常量
2 const float x = PI/2 + 2.6; // 也可以用表达式初始化符号常量
```

✍ **注记:** 由于符号常量的值在程序中不能被修改（即符号常量不能被重新赋值）, 因此在声明时必须初始化.

(4) 常量表达式: constexpr (C++11)

```
constexpr 类型说明符 变量名 = 表达式;
```

(1) 与 const 类似, constexpr 变量的值不能被修改.

(2) 与 const 的区别: 在编译时就能得到计算结果, 而 const 可能会在运行时才得到结果. 但 constexpr 变量初始化时的右边表达式必须是常量表达式.

(3) constexpr 可以用于修饰函数, 此时函数的返回值会尽可能在编译期间被计算出来, 并当作一个常量, 但是如果编译期间此函数不能被计算出来, 那它就会当作一个普通函数处理.

```
1 int x = 11;
2 const int y = 22;
3 const int z1 = x + y; // OK, z = 33
4 constexpr z2 = 11 + y; // OK, 由于 11 和 y 都是常量, 因此 11 + y 是常量表达式
5 constexpr z3 = x + 22; // ERROR, 由于 x 不是常量, 因此 x + 22 不是常量表达式
```

✍ **注记:** constexpr 变量（或函数）在编译时就能得到结果, 因此可以把运行期的计算迁移至编译期, 从而使得程序运行更快（但会增加编译时间）. (ex02_constexpr.cpp)

2.2.4 类型推导: auto 与 decltype

- 类型推导: auto

```
auto 变量名 = 值; // 根据所赋的值推导出变量的类型
```

例: auto 举例. (ex02_auto.cpp)

- 类型推导: decltype

```
decltype(表达式) // 分析表达式获取其类型
```

例: decltype 举例. (ex02_decltype.cpp)

✍ **注记：** decltype 的主要用途是推导表达式的类型，而不是像 auto 那样从变量的初始化表达式中推导类型.

2.2.5 类型别名: typedef 与 using

- 类型别名: 为一个已有的类型说明符另外命名（即取别名），可以取一个别名，也可以取多个别名.

```
typedef 已有类型说明符 类型别名;
using 类型别名 = 已有类型说明符; // C++ 11
```

例 2.3 typedef 和 using: 为一个已有的数据类型说明符另外命名. (ex02_typedef_using.cpp)

2.2.6 枚举

枚举对应的关键字为 enum, 与类型说明符不同, 枚举是用来定义新的数据类型, 同时指定该类型的变量所能取的值.

```
enum 枚举类型名 {变量可取值列表, 即枚举元素};
```

- “枚举元素”按符号常量处理;
- 枚举元素的缺省值, 按顺序依次为: 0, 1, 2, ...;
- 也可以在声明时指定“枚举元素”的值, 如:

例 2.4 enum 枚举示例 (ex02_enum.cpp)

- 枚举值可以进行关系运算;
- 整数值不能直接赋给枚举变量, 需进行强制类型转换.

✍ **注记：** C++ 提供了两种用户自定义数据类型的方法: 类和枚举.

2.2.7 运算与表达式

- 运算符
  (1) 算术运算符: +、-、*、/、%、++ (自增)、-- (自减)
  (2) 赋值运算符: =、+=、-=、*=、/=、%= 、&=、|=、^=、>>=、<<=
  (3) 逗号运算符: , (把若干表达式组合成一个表达式)
  (4) 关系运算符: 用于比较运算, >、<、==、>=、<=、!=
  (5) 逻辑运算符: 用于逻辑运算, &&、||、!
  (6) 条件运算符: ? :, 是一个三目运算符, 用于条件求值
  (7) 求字节数运算符: sizeof (变量/数据/类型说明符)
  (8) 位操作运算符: 按二进制位进行运算, &、|、~、^ (异或)、<< (左移)、>> (右移)

(9) 指针运算符: * (取内容)、& (取地址)

- 运算符优先级（可参见课程主页）
- C++ 语句: 以分号结尾
  (1) 空语句（只有分号）
  (2) 声明语句;
  (3) 表达式语句;
  (4) 复合语句（将多个语句用 { } 括起来组成的一个语句）;
  (5) 选择语句, 循环语句, 跳转语句, 等等.
- 表达式
  (1) 表达式: 由运算符连接常量、变量、函数等所组成的式子;
  (2) 包含赋值运算符的表达式为赋值表达式; 赋值表达式的值是赋值号左边变量的值;
  (3) 表达式可以包含在其它表达式中, 但语句不行!
- 赋值语句
  (1) 标准赋值语句

```
变量 = 表达式;
```

例: 赋值运算. (ex02_assignment.cpp)

  (2) 自增自减: ++, --
  - 前置: 先自增或自减, 然后参与表达式运算;
  - 后置: 先参与表达式运算, 然后自增或自减;
  - 不要在同一语句中包含一个变量的多个 ++ 或 --, 因为它们的解释在 C/C++ 标准中没有规定, 完全取决于编译器的个人行为. 另外, 也不建议出现 y = x++*x; 以及类似的语句.

例 2.5 自增自减运算. (ex02_assignment.cpp)

  (3) 复合赋值运算符: +=、-=、*=、/=、%=

例 2.6 复合赋值运算. (ex02_assignment.cpp)

- 逗号运算符:

```
表达式1 , 表达式2
```

  (1) 先计算 表达式 1, 再计算 表达式 2, 并将 表达式 2 的值作为整个表达式的结果.

例 2.7 逗号运算 (注意运算的优先级!) (ex02_comma.cpp)

✍ **注记:** 为了避免由运算优先级所导致的低级错误, 建议多使用小括号.

- 位运算符: 按二进制位进行运算

&、|、^（异或）、~（取反）、<<（左移）、>>（右移）

例 2.8 二进制位运算. (ex02_bitwise.cpp)

- 求字节数运算符: `sizeof` (是运算符, 不是函数)

```
sizeof(变量)      // 返回指定变量所占的字节数
sizeof(数据类型) // 返回存储单个指定数据类型的数据所需的字节数
sizeof(表达式)   // 返回存储表达式结果所需的字节数
```

例 2.9 `sizeof` 举例. (ex02_sizeof.cpp)

2.2.8 常用数学函数

需加入 `cmath` 头文件: `#include <cmath>`

绝对值, 平方根	`abs(x)`, `sqrt(x)`
e^x, x^y, 对数函数	`exp(x)`, `pow(x,y)`, `log(x)`, `log10(x)`
取整函数	`round(x)`, `ceil(x)`, `floor(x)`, `trunc(x)`
三角函数	`sin`, `cos`, `tan`, `asin`, `acos`, `atan`
双曲三角函数	`sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`

例 2.10 数学函数举例. (ex02_math.cpp)

2.3 C++ 简单输入输出

2.3.1 输入输出语句

```
cin >> 变量名;  // 从键盘读入数据，赋值为指定变量
cout << 变量名; // 输出指定变量的内容
cout << 字符串; // 原样输出字符串内容
```

▷ 输入运算符与输出运算符: `>>` `<<`

▷ 字符串中可以包含转义字符, 常见的转义字符有

`\a`	响铃	`\r`	回车	`\\`	反斜杠
`\b`	退后一格	`\t`	水平制表符	`\'`	单引号
`\n`	换行	`\v`	垂直制表符	`\"`	双引号

例: `cout` 举例. (ex02_cout.cpp)

例: `cin` 举例. (ex02_cin.cpp)

2.3.2 操纵符

- 操纵符: 控制输出格式. 常用的操纵符有 (需加入头文件 `#include <iomanip>`)

操作符	含义
`endl`	插入换行符, 并刷新流 (将缓冲区中的内容刷入输出设备)
`setw(n)`	设置域宽
`left`	左对齐
`right`	右对齐 (缺省方式)
`setfill(c)`	设置填充, `c` 可以是任意字符, 缺省为空格,
`fixed`	使用定点方式输出
`scientific`	使用指数形式输出
`setprecision(n)`	设置输出的有效数字个数, 若在 `fixed` 或 `scientific` 后使用, 则设置小数位数
`showpoint`	显示小数点和尾随零 (即使没有小数部分)

- 操纵符的作用范围:
  - `setw` 仅对紧随其后的输出起作用 (即作用是局部的);
  - 其它操纵符: 至下一个同类型命令为止 (即作用是全局的).

例 2.11 操纵符举例. (ex02_iomanip.cpp)

2.4 程序示例

例 2.12 (银行贷款问题)
已知贷款总额为 L 万元, 贷款年利率为 r, 贷款年限为 y 年, 计算每月需偿还的金额和偿还总额. 假设采用等额本息方式, 则每月的还款额为 (万元) (ex02_loan.cpp)

$$\frac{Lr_m(1+r_m)^{12y}}{(1+r_m)^{12y}-1},$$

其中 r_m 表示月利率, 即 $r_m = r/12$. (思考: 每月还款额公式是怎么推导出来的?)

例 2.13 (显示系统当前的时间)
头文件 `ctime` 中函数 `time(0)` 或 `time(NULL)` 返回当前时间与 1970 年 1 月 1 日零时的时间差 (格林威治时间, 以秒为单位) , 北京时间: 格林威治时间 + 8 小时. (ex02_showtime.cpp)

2.5 上机练习

练习 2.1 编写程序, 从键盘读入圆柱体的半径和高度, 计算其表面积和体积, 并将结果在屏幕上输出. (π 取值 3.14159265) (程序取名 `hw02_01.cpp`)

练习 2.2 银行提供两种 5 年定期存款方式: (1) 一年期方式: 年利率 `10%`, 每年到期后, 自动将本年度的利息加入本金中; (2) 五年期方式: 年利率 `11%`, 五年后本金和利息一起归还储户. 编写程序, 分别以两种方式存入 100 万元, 输出五年后各得多少? (程序取名 `hw02_02.cpp`)

练习 2.3 修改程序 `ex02_showtime.cpp`, 使得输出的时、分、秒都占两个位置, 如: 14 点 25 分 10 秒显示为 `14:25:10`, 9 点 8 分 5 秒显示为 `09:08:05`. (程序取名 `hw02_03.cpp`)

第三讲 选择与循环

本讲主要内容

- 关系运算, 即比较大小
- 逻辑运算: 与、或、非、异或
- 选择语句（也成为条件语句, 分支语句）: `if`, `switch`
- 循环语句: `while`, `do while`, `for`
- 循环的终止: `continue`, `break`

3.1 关系运算与逻辑运算

(1) 关系运算, 即比较大小: `>` `<` `==` `>=` `<=` `!=`

- 结论是真则返回 `true`, 否则返回 `false`
- C++ 中用 `1` 表示 `true`, `0` 表示 `false`
- `bool` 型变量的值为 `0` 时表示 `false`, 其他它值都表示 `true`
- 注意“`==`”与“`=`”的区别

> ✍ **注记：** 由于存在舍入误差, 对于浮点数, 慎用相等比较运算. (ex03_bool_float.cpp)

(2) 逻辑运算: `&&`（逻辑与）, `||`（逻辑或）, `!`（逻辑非）

- 表达式 1 `&&` 表达式 2
  - 先计算 **表达式** 1 的值, 若是 `true`, 再计算 **表达式** 2 的值;
  - 若 **表达式** 1 的值是 `false`, 则不再计算 **表达式** 2.
- 表达式 1 `||` 表达式 2
  - 先计算 **表达式** 1 的值, 若是 `false`, 再计算 **表达式** 2 的值;
  - 若 **表达式** 1 的值是 `true`, 则不再计算 **表达式** 2.

> ✍ **注记：** 一定要注意 `&&` 和 `||` 的运算方式.

- 优先级: `!` 优于 `&&` 优于 `||`.

(3) 条件运算符: `?` `:`

```
条件表达式 ? 表达式1 : 表达式2
```

- C++ 中唯一的 **三目运算符**;
- **条件表达式** 为真时返回 **表达式** 1 的值, 否则返回 **表达式** 2 的值;
- **表达式** 1 的值和 **表达式** 2 的值的数据类型要一致.

例 3.1 关系运算举例. (ex03_bool.cpp)

3.2 选择结构

在 C++ 语言中, 选择结构有两种实现方式: `if` 和 `switch`.

3.2.1 IF 语句

(1) 单分支: 如果“条件表达式”为 true, 则执行后面的语句, 否则不执行.

```
if (条件表达式) 语句
```

✍ 注记: 这里的语句可以是复合语句（如果是复合语句的话, 别忘了大括号!）

(2) 双分支: 如果“条件表达式”的值为 true, 则执行“语句 1”, 否则执行“语句 2”.

```
if (条件表达式)
    语句1
else
    语句2
```

(3) 多分支:

```
if (条件表达式)
    语句1
else if (条件表达式)
    语句2
else if (条件表达式)
    语句3
  ⋮
else
    语句n
```

✍ 注记: 条件表达式两边的小括号不能省略!

(4) `if` 语句可以嵌套;

(5) 嵌套时每一层 `if` 都要和 `else` 配套, 若没有 `else`, 则需将该层 `if` 语句用 `{ }` 括起来.

例 3.2 IF 嵌套举例: 计算一个整数的符号. (ex03_if_03.cpp)

3.2.2 SWITCH 结构

```
switch(表达式) // 这里的 表达式 的值可以是整型、字符型或枚举型
{
   case 常量表达式1:
语句    case 常量表达式2:
        语句
     ⋮
   case 常量表达式n:
        语句
   default:
        语句
}
```

(1) 先计算 switch 后面的“表达式”的值, 然后依次与每个 case 后面的“常量表达式”进行匹配, 一旦匹配成功, 则开始执行其后面的语句, 包括其后面所有 case 以及 default 的语句（除非遇到 break）;
(2) 如果没有匹配的, 则执行 default 后面的语句;
(3) default 不是必需的, 即可以没有;
(4) 每个 case 分支最后一般都会加上 break 语句;
(5) 每个 case 后面的常量表达式的值不能相同;
(6) 每个 case 后面可以有多个语句（复合语句）, 但可以不用 { }.

例 3.3 switch 结构举例. (ex03_switch.cpp)

3.3 循环结构

- 重复执行: 代码不变, 但数据在变;
- 循环结构的三种实现方式: while 循环, do while 循环和 for 循环.
- 循环可以嵌套.

3.3.1 WHILE 循环

```
while(条件表达式)
{
    循环体语句
}
```

- 执行过程
  (1) 计算 条件表达式 的值;
  (2) 如果是“真”, 则执行循环体语句; 否则退出循环;

(3) 返回第 (1) 步.

✍ **注记：** 如果循环体语句是复合语句, 别忘了大括号!

例 3.4 while 循环举例. (ex03_while.cpp)

3.3.2 DO WHILE 循环

```
do
{
  循环体语句
} while(条件表达式);
```

- 执行过程
  (1) 执行循环体语句;
  (2) 判断条件表达式的值, 如果是“真”, 则返回第 (1) 步; 否则退出循环.
- 与 while 循环的区别: 无论条件是否成立, 循环体语句至少执行一次.

3.3.3 FOR 循环

```
for (初始语句; 表达式1; 表达式2)
{
   循环体语句
}
```

- 执行过程
  (1) 执行初始语句;
  (2) 计算表达式 1 的值, 如果是“真”, 则执行循环体语句, 否则退出循环;
  (3) 执行表达式 2, 返回第二步.

† 初始语句, 表达式 1, 表达式 2 均可省略, 但分号不能省;
† 表达式 1 是循环控制语句, 如果省略的话就构成死循环;
† 循环体可以是单个语句, 也可以是复合语句;
† 初始语句 与 表达式 2 可以是逗号语句;
† 若省略初始语句 和 表达式 2, 只有表达式 1, 则等同于 while 循环.

- for 循环有时也可以描述为

```
for (循环变量赋初值; 循环条件; 循环变量增量)
{
  循环体语句
}
```

```
int i, s = 0;
for (i = 1; i <= 10; i++)
    s = s + i;
```

- 循环变量也可以在初始语句中声明, 此时循环变量只在该循环内有效, 循环结束后即被释放.

```
int s = 0;
for (int i = 1; i <= 10; i++)
    s = s + i;
```

3.3.4 基于范围的 for 循环

```
for (声明: 范围表达式)
{
   循环体语句
}
```

- “声明”通常是循环变量的声明, “范围表达式”则指定循环变量的取值范围, 通常是数组、列表或容器类对象等, 也可以是用大括号括起来的一组对象.

例 3.5 基于范围的 for 循环举例. (ex03_for_range.cpp)

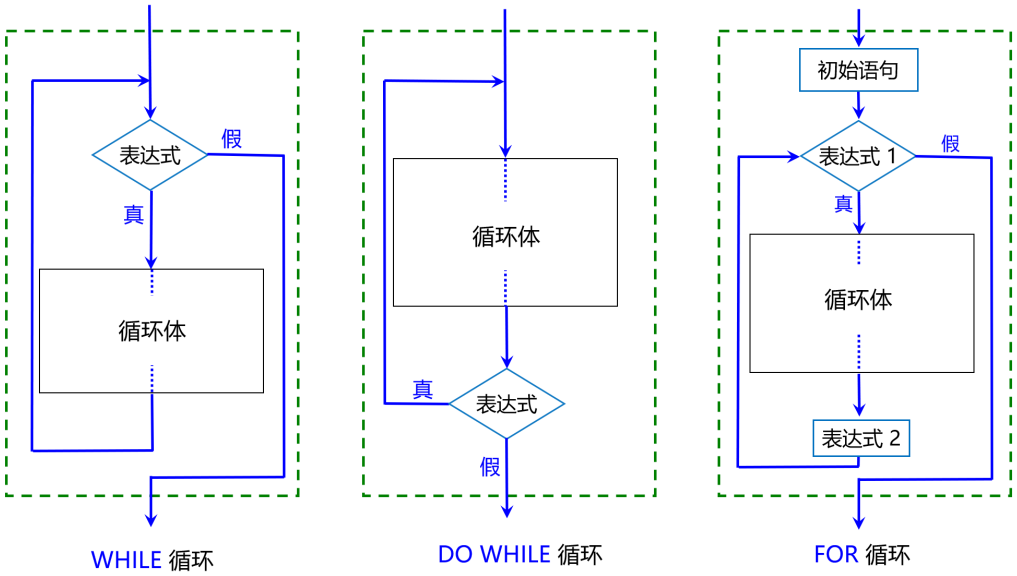


图 3.1. 三种循环示意图

3.3.5 循环的非正常终止

```
break    // 跳出循环体，但只能跳出一层循环，一般用在循环语句和 switch 语句中.
continue // 结束本轮循环，执行下一轮循环，一般用在循环语句中.
goto     // 跳转语句，不建议使用.
```

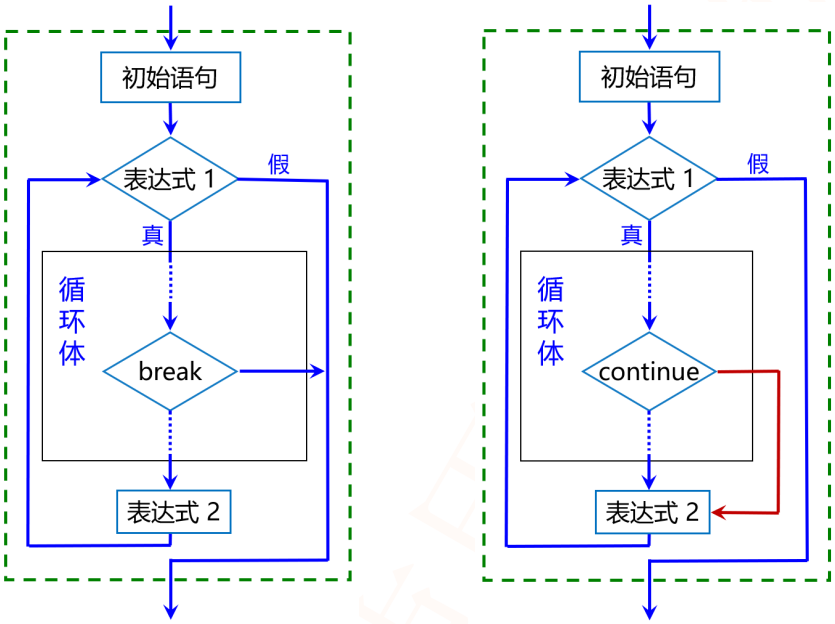


图 3.2. break 和 continue 图示, 以 for 循环为例: for (初始语句; 表达式 1; 表达式 2)

✍ **注记：** break 和 continue 通常与 if 语句配合使用.

3.4 程序示例

例 3.6 在屏幕上打印九九乘法表. (ex03_for_99.cpp)

例 3.7 给定一个正整数, 在屏幕上输出其所有互异的整数因子, 比如 12 的所有互异的整数因子为 1, 2, 3, 4, 6, 12. (ex03_for_factor.cpp)

例 3.8 判断一个给定的正整数是否为素数. (ex03_for_prime.cpp)

✍ 思考: 如何改进算法, 使得执行效率更高?

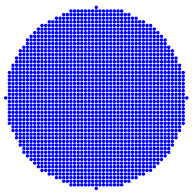
例 3.9 计算两个正整数的最大公约数, 穷举法. (ex03_for_gcd.cpp)
若考虑执行效率的话, 可以将其中的循环加以改进, 以减少循环次数. (ex03_for_gcd_new.cpp)

✍ 注: 计算最大公因数一般使用辗转求余法, 见习题 4.1.

例 3.10 猜生日: 请回答你的生日出现在下面五组数的哪几组中? 这几组数的第一个数字之和就是你的生日. (ex03_birthday.cpp)

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
8	9	10	11	12	13	14	15	24	25	26	27	28	29	30	31
4	5	6	7	12	13	14	15	20	21	22	23	28	29	30	31
2	3	6	7	10	11	14	15	18	19	22	23	26	27	30	31
1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31

例 3.11 打印图案: 打印右下图所示的圆. (ex03_plot.cpp)



3.5 应用: 定积分的数值计算

见课程主页, 或附录.

3.6 上机练习

练习 3.1 **成绩转换**: 按下面的要求编写四个程序. 学生的成绩有两种表示方法: 等级制和百分制, 对应关系如下:

等级	A	B	C	D	E
分数	90–100	80–89	70–79	60–69	0–59

请编写程序, 实现它们之间的互换, 即

(a) 从键盘输入一个分数, 然后输出其对应的等级, 分别用 `if` 语句和 `switch` 语句实现. (程序取名 `hw03_if_1.cpp`, `hw03_switch_1.cpp`)

(b) 从键盘输入一个等级, 然后输出其对应的分数区间, 分别用 `if` 语句和 `switch` 语句实现. (程序取名 `hw03_if_2.cpp`, `hw03_switch_2.cpp`)

注: 输入语句 `cin` 之前要有提示, 如“请输入成绩 (百分制) : ”

练习 3.2 **跳出多重循环**: 修改程序 `ex03_goto.cpp`, 避免使用 `goto` 语句. (程序取名 `hw03_02.cpp`)

练习 3.3 **最大素数**: 编写程序, 实现下面的功能: 从键盘输入一个大于 1 的整数 (需要判断输入的合法性, 即是否大于 1), 然后求出不超过这个整数的最大素数, 并在屏幕上输出, 要求使用 `for` 循环和 `if` 语句. (程序取名 `hw03_03.cpp`)

练习 3.4 **消去误差, 即大数吃小数** 当一个很大的数与一个很小的数进行加减运算时, 由于计算机的舍入误差, 小数可能被大数“吃掉”, 比如以双精度方式计算 10000000000+0.0000000001 时, 结果为

10000000000. 因此在做数值计算时应尽可能避免这种情况的发生. 在计算下面的级数时, 由右至左计算会比由左至右计算获得更精确的结果:

$$S = 10^{14} + 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

编写程序, 分别从右至左和从左至右计算上面级数的和, 并在屏幕上输出（显示小数点后六位）, 取 $n = 50000$. （程序取名 `hw03_04.cpp`）

（思考: 如果用 `long double`, 则结果怎样?）

练习 3.5 计算常数 e: 用下面的级数部分和来近似常数 e

$$e \approx 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots + \frac{1}{n!}$$

编写程序, 输出当 $n = 10, 20, 30, 40, 50$ 时的结果. （程序取名 `hw03_05.cpp`）

（提示: 用 `long double`, $e = 2.718281828459045235...$）

（思考: 怎样提高精度? → 从右向左计算, 并借助计算多项式的 Horner 技巧）

练习 3.6 闰年: 求出 21 世纪（2001 至 2099 年）中所有的闰年, 并在屏幕上输出.

闰年的判定标准: (1) 能被 400 整除; 或者 (2) 能被 4 整除但不能被 100 整除.

要求: 每行输出 6 个, 闰年之间用两个空格隔开. （程序取名 `hw03_06.cpp`）

练习 3.7 定积分的近似计算: 分别用复合梯形法和复合抛物线法计算定积分 $\int_0^{\frac{\pi}{2}} \sin x \, \mathrm{d}x$ 的近似值, 取 20 等分. （程序取名 `hw03_07.cpp`）

（思考: 如果要使得计算结果的误差不超过 10^{-6}, 怎么做比较好?）

练习 3.8 最大正整数与最小正实数: 编写程序, 找出计算机所能表示的最大 `short` 型正整数和最小的 `float` 型正实数, 并在屏幕上输出 (输出最小 `float` 型正实数时保留小数点后 8 位). （程序取名 `hw03_08.cpp`）

第四讲 函数

函数是程序设计中对功能的抽象, 是 C++ 语言的基本组成部分, 也是实现模块化程序设计的基本工具. 在 C++ 语言程序设计中, 不仅要掌握函数的定义与使用, 还要掌握如何利用函数实现程序的模块化设计.

本讲主要内容

- 函数基础
  - 函数的定义、调用与声明
  - 函数间的参数传递
  - 函数嵌套
  - 内联函数
- 数据的作用域
  - 什么是作用域, 局部作用域与全局作用域
  - 局部变量与全局变量
  - 作用域解析运算符, 命名空间
  - 生存期, 静态变量
  - 形参带缺省值的函数, 函数重载
- 编译预处理与多文件结构
  - 头文件
  - 宏定义
  - 条件编译
  - 多文件结构
  - 外部变量, 外部函数
- 系统函数

4.1 函数的声明、定义与调用

C++ 程序是由一个或多个函数构成的, 且必须有一个 `main` 函数, 通常称为**主函数**. C++ 程序总是从主函数开始, 在主函数中结束.

- 函数的定义: 函数由两部分组成, 分别是**函数头**和**函数体**.

```
类型说明符 函数名(形式参数列表) // 函数头
{
    函数体
}
```

(1) “类型说明符”指明了函数的类型, 即函数返回值的类型, 若没有返回值, 则使用 void

(2) 形式参数列表: 可以有多个形式参数, 也可以没有.

```
类型说明符 变量, 类型说明符 变量, ... ...
```

- ▷ **形式参数**（通常简称**形参**）需要指定数据类型.
  （形参在函数定义时不会分配任何存储空间, 也没有具体的值, 因此称为形式参数）
- ▷ 有多个形参时, 用逗号隔开, 每个形参需单独指定数据类型.
- ▷ 如果函数不带参数, 则形参可以省略, 但小括号不能省.
- ▷ 形参只在函数内部有效/可见, 即形参是局部变量（有关局部变量的含义见后面的“函数作用域”）

```
1 int my_max(int x, int y) // OK, 两个形参, 返回一个整型数据
2 int my_max(int x, y) // ERROR
```

(3) 函数的返回值

- ▷ 函数返回值通过 return 语句给出.
- ▷ 若没有返回值, 可以不写 return, 也可以写不带任何表达式的 return.

```
1 int my_max(int x, int y)
2 {
3     if (x > y) return x;
4     else return y;
5 }
```

- 函数的调用

```
函数名(实际参数列表)
```

(1) **实际参数**（通常简称**实参**）必须是实际存在的变量或表达式, 即具有具体的值.

(2) 实参与形参一一对应.

例 4.1 计算两个整数的最大值. (ex04_my_max_01/02.cpp)

主调函数与被调函数

为了描述方面, 如果在函数 A 中调用函数 B, 我们称 A 是 **主调函数**, B 是 **被调函数**.

- 函数的声明

(1) 如果被调函数在主调函数之前已经定义, 则可直接调用.

(2) 如果被调函数是在主调函数后面才定义, 则需要在调用前事先声明.
函数声明: 函数头加分号, 如: `int my_max(int x, int y);`

(3) 被调函数的声明: 可以在主调函数中声明, 也可以在所有函数之前声明.

(4) 被调函数可以出现在表达式中, 此时必须要有返回值.

(5) 被调函数执行过程中遇到 return 语句时, 就返回主调函数, 如果 return 后面带有表达式, 则将该表达式的值带回到主调函数, 如果 return 后面没有表达式, 则直接返回到主调函数.

例 4.2 计算 x^k. (ex04_my_pow.cpp)

例 4.3 编写函数, 将一个二进制正整数转化为相应的十进制数, 例如:

$$(10101)_2 = 1\times 2^4 + 0\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0.$$

分析: 需解决两个问题: (1) 如何提取每个数位上的数字; (2) 如何确定 2 的次数.
提示: 从右往左计算. (ex04_bin2dec.cpp)

思考: (1) 如何提高计算效率: 降低计算 2^k 的运算次数;
(2) 如何计算 1111 1111 1111 1111 对应的十进制数? (提示: 字符数组)

例 4.4 编写函数, 利用 Taylor 展开计算 $\sin(\pi/2)$ 的值. (直到级数某项的绝对值小于 10^{-15} 为止)

$$\sin(x) = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots = \sum_{k=0}^{\infty}(-1)^k\frac{x^{2k+1}}{(2k+1)!}$$

分析: 注意大数越界和误差累积, 并尽可能简化通项计算方法. (ex04_my_sin.cpp)

思考: 利用上面的公式计算 $\sin(41\pi/2)$, 结果如何?

例 4.5 找出 11 ~ 999 之间的数 m, 满足 m、m^2 和 m^3 均为回文数.
(回文数: 各位数字左右对称的整数, 如 11, 121, 1331)
分析: 利用除以 10 取余的方法, 从最低位开始, 依次取出该数的各位数字. 按反序重新构成新的数, 比较与原数是否相等, 若相等, 则原数为回文数. (ex04_huiwen.cpp)

例 4.6 计时函数 clock. (需包含头文件 ctime) (ex04_clock.cpp)

- clock() : 返回进程启动后所使用的 cpu 总毫秒数.

- 如果需要更高精度的时间 (如微秒、纳秒), 可以使用 chrono 库 (C++11) (ex04_chrono.cpp)

例 4.7 计时函数 time. (需包含头文件 ctime) (ex04_time.cpp)

- time(NULL) 或 time(0) : 返回从 1970 年 1 月 1 日 0 时 0 分 0 秒至当前时刻的总秒数.

例 4.8 随机数的生成. (需包含头文件 cstdlib) (ex04_rand_01.cpp)

- rand(): 返回一个 0 ~ RAND_MAX 之间的伪随机整数.
- srand(seed): 设置种子. 如不设定, 默认种子为 1.
- 相同的种子对应相同的伪随机整数.
- 每次执行 rand() 后, 种子会自动改变, 但变化规律是固定的.

- 如果需要满足不同分布的随机数, 可以使用 random 库（C++11） (ex04_random.cpp)

思考： 如何生成 $[a,b]$ 之间的随机整数？（其中 a,b 为正整数） (ex04_rand_02.cpp)

思考： 如何生成 $[0,1]$ 之间的随机双精度数？$[a,b]$ 之间的随机双精度数？(a,b 为任意双精度数)

例 4.9 (猜数游戏) 由计算机随机产生 $[1,100]$ 之间的一个整数, 然后由用户猜测这个数. 要求根据用户的猜测情况给出不同的提示: 如果猜测的数大于产生的数, 则显示 Larger; 小于则显示 Smaller; 等于则显示 You won! 同时退出游戏. 用户最多有 7 次机会. (ex04_game.cpp)

Tips: 如何使得代码在每次执行时产生不同的随机整数？

```
1 srand(time(NULL)); // srand(time(0)), 将当前时间作为生成随机数的种子
2 x = rand();
```

4.2 函数间的参数传递

- 传递方式一: 值传递
  (1) 将实参的值传递给对应的形参, 即单向传递;
  (2) 除了一些特殊情形（如形参带缺省值等), 实参和形参在数量、类型、顺序上要一一对应;
  (3) 形参只在函数被调用时才分配存储单元, 调用结束后即被释放;
  (4) 实参可以是常量、变量、表达式、函数 (名) 等, 但它们必须要有明确的值;
  (5) 形参获得实参传递过来的值后, 便与实参脱离关系, 也就是说, 函数中对形参的任何改变都不会对实参产生任何影响.

  思考： 如何交换两个变量的值? (ex04_myswap_01.cpp)

- 传递方式二: 引用传递
  (1) 引用 的声明: &

```
1 int a;
2 int & ra = a; // 声明 ra 是指向 a 的引用, 即 ra 是 a 的别名
```

  (2) 引用是一种特殊类型的变量, 可看作是变量的别名;
  (3) 声明一个引用时必须初始化, 指向一个存在的对象;
  (4) 引用一旦初始化就不能改变, 即不能再作为其它对象的引用（别名）;
  (5) 若引用作为形参, 则函数被调用时才会被初始化, 此时形参是实参的一个别名, 对形参的任何操作也会直接作用于实参, 也就是说, 如果形参的值被改变则实参的值也会被改变.

例 4.10 引用作为形参, 交换两个变量的值. (ex04_myswap_02.cpp)

4.3 内联函数

- 内联函数, 关键字: `inline`

```
inline 类型说明符 函数名(形式参数列表)
```

(1) 与普通函数的区别: 编译时在调用处用函数体进行替换;
(2) 内联函数的优势: 节省参数传递、控制转移等开销, 提高代码的执行效率;
(3) 内联函数要求: 通常应该是功能简单、规模小、使用频繁的函数;
(4) 内联函数体内不建议使用循环语句和 `switch` 语句.

> ✍ **注记:** 内联函数不会影响程序结果, 只会影响执行效率. 有些函数无法定义成内联函数, 如递归函数.

例 4.11 内联函数举例. (ex04_inline.cpp)

4.4 函数嵌套与递归

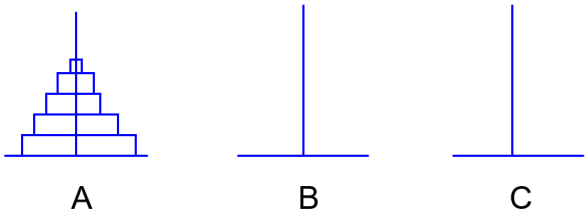
- 函数的嵌套调用
  (1) 函数可以嵌套调用, 但不能嵌套定义.
  (2) 函数也可以 **递归调用** (函数直接或间接调用自己)

例 4.12 利用下边的公式计算阶乘: (ex04_factorial.cpp)

$$n! = \begin{cases} 1, & n = 0, \\ n * (n-1)!, & n > 0. \end{cases}$$

> ✍ **注记:** 对同一个函数的多次不同调用中, 编译器会给函数的形参和局部变量分配不同的存储空间, 它们互不影响.

例 4.13 汉诺塔问题: 有三根针 A、B、C, A 针上有 n 个大小不同的盘子, 大的在下, 小的在上, 把这 n 个盘子从 A 针移到 C 针, 在移动过程中可以借助 B 针. 要求: (1) 每次只允许移动一个盘, (2) 在移动过程中在三根针上的盘子都要保持大盘在下, 小盘在上. (ex04_hanoi.cpp)



分析: 该问题可分解为下面三个步骤:

(1) 将 A 上 $n-1$ 个盘子移到 B 针上（借助 C 针）;

(2) 把 A 针上剩下的一个盘子移到 C 针上;

(3) 将 $n-1$ 个盘子从 B 针移到 C 针上（借助 A 针）.

上面三个步骤包含两种操作:

(1) 将多个盘子从一个针移到另一个针上, 这是一个递归的过程, 我们用 `hanoi` 函数实现;

(2) 将 1 个盘子从一个针上移到另一针上, 该过程用 `move` 函数实现.

4.5 变量的作用域

作用域就是一个范围, 或者区域. 每个变量都有作用域, 即在程序中哪些地方可以使用该变量.

- 变量的作用域: 变量在程序中有效（可见）的区域.
- 一般来说, 可以在三个地方定义变量:
  - 在函数或某个语句块内声明的变量, 这些变量是 **局部变量**
  - 在所有函数外部声明的变量, 这些变量是 **全局变量**
  - 在函数声明或定义时声明的形参, 这些变量也是 **局部变量**
- 常见的作用域:

(1) 函数原型作用域: 函数原型声明时形参的有效范围, 仅限形参列表的左右括号之间, 因此在函数声明时, 形参的变量名可省略（但类型不能省）.

```
1  int my_max(int x, int y); // x, y 的作用域仅限于形参列表的左、右括号之间
2  int my_max(int, int); // 变量名可以省略
```

✍ **注记**: 需要注意的是, 这里是函数原型, 不是函数定义.

(2) 局部作用域: 函数, 复合语句块（一对大括号括起来的部分）等.

✍ **注记**: 函数体内声明的变量, 作用域从声明开始, 到声明所在的语句块结束为止.

```
1  double my_pow(double x, int k) // x, k 的作用域为整个函数
2  {
3      if (k == 1) return x;
4      else
5      {
6          double y = 1.0; // y 的作用域到第 10 行为止
7          for (int i = 1; i <= k; i++) // i 的作用域仅限于 for 循环
```

```
            y = y * x;
        return y;
    }
    return 0;
}
```

(3) 类作用域（详见后面的类与对象）.

- 局部变量与全局变量
  (1) 函数定义时的形参, 或函数内部声明的变量均为**局部变量**, 只在该函数内有效;
  (2) 语句块中声明的变量是**局部变量**, 只在该语句块中有效;
  (3) for 循环的初始语句中声明的变量也是**局部变量**, 只在 for 循环中有效;
  (4) 在所有函数外定义的变量为**全局变量**, 在它后面的函数中均可以使用; 若要在它前面的函数中使用该全局变量, 则需在函数中声明其为外部变量.

```
extern 类型说明符 变量名;
```

✍ **注记：** 若局部变量与全局变量同名, 则优先使用局部变量!

例 4.14 局部变量和全局变量. (ex04_var_global_01.cpp)

- 作用域解析运算符: :: （两个连续的冒号）
  若存在同名的局部变量和全局变量, 则缺省引用局部变量, 此时若需引用全局变量, 需在变量名前加作用域解析运算符.

例 4.15 作用域解析运算符. (ex04_var_global_02.cpp)

- **命名空间 namespace**
  大型程序通常由不同模块组成, 不同模块中的类和函数等可能存在重名. 为了解决这个问题, C++ 引入命名空间概念.

```
namespace 命名空间名
{
    各种声明，包括函数声明，类声明等
}
```

(1) 命名空间内的元素, 可以是类、函数、变量等, 均称为名字;
(2) 命名空间的使用: using
(3) 可以将命名空间中的所有名称都导入到当前作用域中, 也可以只导入指定的某个名称;

例 4.16 命名空间举例. (ex04_namespace_01/02/03.cpp)

(4) **标准命名空间**: 标准库的所有函数、类、对象等, 都在 std 命名空间中.

```
using namespace std; // 导入标准命名空间中所有名字
using std::cout; // 只导入标准命名空间中的 cout
```

- 可见性
  (1) 可见性是指对标识符（变量, 函数等）是否可以访问;
  (2) 如果标识符在某处可见, 则就可以在该处引用此标识符;
  (3) 对于两个嵌套的作用域, 若内层作用域内定义了与外层作用域中同名的标识符, 则外层作用域的标识符在内层不可见.
- 生存期
  (1) **静态生存期**: 生存期与程序的运行期相同, 即一直有效. 静态变量和全局变量具有静态生存期.
  (2) **动态生存期**: 当对象所在的程序块执行完后即消失. 动态变量（含局部变量）具有动态生存期.
- 静态变量
  (1) 静态变量的声明: `static`

```
static 类型说明符 变量名;
```

  (2) 静态局部变量不会随函数的调用结束而消失, 下次调用该函数时, 该变量会保持上次调用结束后的值;
  (3) 没有初始化的静态变量（包括静态局部变量和全局变量）会自动初始化为 0;
  (4) 静态变量只能初始化一次.

例 4.17 静态局部变量. (ex04_var_static.cpp)

4.6 形参带缺省值

- 形参缺省值
  (1) 函数在声明或定义时可以预先给形参设定一个值（即缺省值）, 函数被调用时, 如果给定实参, 则采用实参的值, 否则采用预先设定的缺省值.
  (2) 好处: 调用时可以不提供或只提供部分实参.

```
int add(int x = 5, int y = 6) // 两个形参均带缺省值
{ return x + y; }

int main()
{
    add(10,20); // 10 + 20
    add();      // 5 + 6
    add(10);    // 10 + 6
}
```

- 形参可以全部带缺省值, 也可以部分带缺省值, 如果部分带缺省值, 则规定如下:
  (1) 形参缺省值必须从右向左顺序声明;

(2) 带缺省值的形参, 其后面不能有不带缺省值的形参（在函数调用时, 实参与形参的配对是按从左向右的顺序进行的）;

```
1 int add(int x, int y = 5, int z = 6); // OK
2 int add(int x = 1, int y = 5, int z); // ERROR
3 int add(int x = 1, int y, int z = 6); // ERROR
```

- 缺省值是在函数声明时设定, 还是在函数定义时设定?
  (1) 同一作用域中, 哪个在前就由哪个设定, 后面的不能再设定（如果先声明, 后定义, 则声明时设定, 定义时就不能再设定, 反之亦然）;
  (2) 在不同作用域内, 函数声明时可以设置不同的缺省值.

例 4.18 给形参设置缺省值: 先声明后定义. (ex04_DefaultValue_01.cpp)

例 4.19 给形参设置缺省值: 先定义后声明. (ex04_DefaultValue_02/03.cpp)

4.7 函数重载

C++ 允许**功能相近**的函数在相同的作用域内使用**相同函数名**, 从而形成重载, 不仅方便使用, 也便于记忆.

- 函数重载: 两个以上的函数, 具有相同的函数名, 但形参不同, 调用时, 编译器会根据实参和形参的最佳匹配, 自动确定调用哪一个函数.
  (1) 重载函数特点: 函数名必须相同, 形参必须不同（个数不同或类型不同）;
  (2) 函数名和形参都相同, 函数返回值类型不同, 不形成重载, 是语法错误!

✍ **注记:** 不建议重载功能不同的函数. 在使用带有形参缺省值的重载函数时, 要防止二义性!

```
1 int add(int x, int y = 1);
2 int add(int x);
3 add(10); // ??? 有歧义
```

例 4.20 函数重载举例. (ex04_overload.cpp)

4.8 预编译处理与多文件结构

编译预处理, 简称 **预编译**, 是指编译器在实际编译之前所需进行的一些预处理. 一般来说, C/C++ 程序的编译过程分为如下几个阶段: 编译预处理、编译、汇编、链接. 编译预处理负责读取源程序, 对其中的编译预处理指令和特殊符号进行处理（即扫描源代码, 对其进行初步的整理和相应的转换）, 生成新的源代码, 然后传递给编译器进行编译.

4.8.1 预编译

- 编译预处理主要包含以下三个功能: 文件导入（如导入头文件）, 宏定义和条件编译.

- 编译预处理指令以 “#” 开头.
- 文件导入: #include

(1) 导入头文件的两种方式:

```
#include <文件名> // 按标准方式导入，即在编译器指定的文件夹中寻找指定文件
#include "文件名" // 先在当前文件夹中寻找，然后再按标准方式搜索指定文件
```

(2) C++ 中的库函数通常是在头文件中声明, 使用这些函数需要导入其所在的头文件. 如数学函数对应的头文件是 cmath.

(3) 头文件可以由系统提供, 也可以由用户自己编写. 用户编写的头文件通常以 “.h” 为后缀名.

(4) 常用系统头文件

iostream	基本输入输出
iomanip	操纵符
cmath	数学函数
ctime	时间与日期, clock, time, clock_t, time_t, CLOCKS_PER_SEC, ...
chrono	时间与日期, chrono 类
cstdlib	abs, rand, srand, system, ...
cstring	C 语言字符串操作
cctype	C 语言字符操作
cstdio	C 语言文件操作: fopen, fclose, fread, fwrite, printf, ...
string	字符串类
vector	向量类
random	随机数
fstream	文件流

(5) 头文件中可以包含其他头文件.

(6) 在程序开发中, 有时需要在多个源文件中使用同一个函数, 此时可以将函数单独定义在某个 cpp 文件中 (函数只能定义一次), 然后将函数的声明写到一个头文件中, 需要使用该函数时, 只需导入该头文件即可.

- 宏定义: #define, #undef

```
1 #define PI 3.14159 // 定义宏
2 #undef PI // 删除由 #define 定义的宏
```

✍ **注记:** 在很多情况下可以由 const 实现该功能. 此外, #define 还可以用来定义带参数的宏, 但也可以用内联函数取代.

- 条件编译

```
#if 常量表达式
    程序正文  // 当“常量表达式”非零时编译
#endif
```

```
#if 常量表达式
    程序正文  // 当“常量表达式”非零时编译
#else
    程序正文  // 否则编译这段程序
#endif
```

```
#if 常量表达式1
    程序正文  // 当“常量表达式1”非零时编译
#elif 常量表达式2
    程序正文 // 否则，当“常量表达式2”非零时编译
#elif 常量表达式3
    程序正文 // 否则，当“常量表达式3”非零时编译
... ...
#else
    程序正文  // 否则编译这段程序
#endif
```

```
#ifdef 标识符
    程序正文  // 当“标识符”已由 #define 定义时编译
#else
    程序正文  // 否则编译这段程序
#endif
```

```
#ifndef 标识符
    程序正文  // 当“标识符”没有定义时编译
#else
    程序正文  // 否则编译这段程序
#endif
```

4.8.2 多文件结构

一个程序可以由多个文件组成, 编译时可以使用工程/项目来组合. 若使用命令行编译, 可一起编译, 也可以先单独编译, 生成各自的目标文件, 然后链接成一个可执行文件.

- 外部变量: 如果需要用到其它文件中定义的变量, 则需要用 extern 声明其为外部变量.

```
extern 类型说明符 变量名;
```

- 外部函数: 如果需要用到其它文件中定义的函数, 则需要用 extern 声明其为外部函数.

```
extern 函数声明/函数原型;
```

- 系统函数
  - ▷ 标准 C++ 函数（库函数): 可以直接使用, 使用时包含相应的头文件即可.
  - ▷ 非标准 C++ 函数: 编译器商家或者软件开发商提供的库函数.

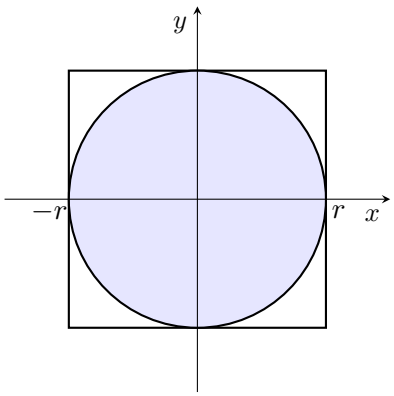
充分使用库函数不仅可以大大减少编程工作量, 还可以提高代码可靠性和执行效率. 为了提高执行效率, 部分库函数有可能是用汇编语言编写的.

4.9 应用: 蒙特卡罗算法

蒙特卡罗（Monte Carlo）算法是一种计算机随机模拟方法, 是在第二次世界大战期间随着计算机的诞生而兴起和发展起来的, 可用于计算一些复杂问题的数值解, 比如不规则区域的面积、体积等. 这种方法在物理、化学、生态、社会、经济等领域中得到广泛运用, 包括现在热门的机器学习和人工智能. 蒙特卡罗算法的名字来源于世界著名的赌城 — 摩纳哥的“蒙特卡洛”, 其思想可追溯到 1777 年法国科学家蒲丰（C. Buffon）提出的一种近似计算圆周率 π 的方法, 即著名的“蒲丰投针实验”.

下面以计算 π 的近似值为例, 描述蒙特卡罗算法的基本思想.

如右图所示, 设正方形的边长为 $2r$（$r>0$）, 其内切圆半径为 r. 我们采用蒙特卡罗算法来计算内切圆面积的近似值, 做法如下: 向正方形区域内随机投放 n 的点（满足均匀分布）, 然后统计落在内切圆中的点的个数, 记为 m. 根据几何概率原理, 当 n 充分大时, m 与 n 的比值非常接近圆的面积与正方形面积的比值, 即



$$\frac{m}{n}\approx\frac{\text{内切圆的面积}}{\text{正方形的面积}}=\frac{\pi r^2}{4r^2}=\frac{\pi}{4}.$$

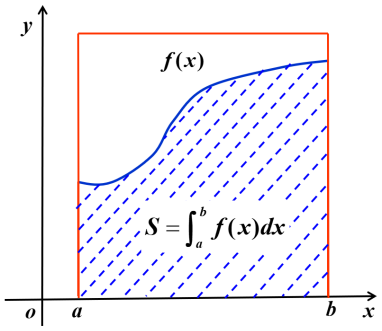
所以, 我们就可以通过下面的公式来计算 π 的近似值:

$$\pi\approx\frac{4m}{n}.$$

例 4.21 用蒙特卡罗（Monte Carlo）方法计算 π 的近似值. (ex04_Monte_Carlo_pi.cpp)

例 4.22 用蒙特卡洛算法计算定积分 $\int_a^b f(x)\,\mathrm{d}x$ 的近似值.

分析: (1) 绘图, 将定积分计算转化为计算曲边梯形的面积, 比如下图中的阴影部分.



(2) 选定一个简单的大区域, 要求覆盖阴影部分, 且容易计算面积, 通常为恰好包含阴影部分长方形, 比如上图中的红色长方形.

(3) 往这个大区域中随机投点（假定投 N 个点), 统计落在阴影部分的点的个数（设为 M 个）

(4) 计算阴影部分面积的近似值: $S = \frac{M}{N} \times$ 大区域的面积.

以函数 $f(x) = \sin(x)$ 为例, 实现该方法, 见练习 4.11.

4.10 上机练习

练习 4.1 编写两个函数, 分别计算两个正整数的最大公约数与最小公倍数, 要求用辗转求余法计算最大公约数, 最小公倍数则可以通过调用最大公约数函数实现. 在主函数中计算 2012 与 1509 的最大公约数与最小公倍数.（函数名分别为 `gcd` 和 `lcm`）（程序取名 `hw04_01.cpp`）

```
int gcd(int x, int y);
int lcm(int x, int y);
```

✍ **注记:** 我们可以用穷举法来计算两个非负整数的最大公约数, 但效率不高. 早在公元前 300 年, 欧几里得就提出了一种效率较高的计算方法, 其核心思想是基于下面的结论:

$$\gcd(a, b) = \gcd(b, a \bmod b) \qquad (\text{这里假定} a > b).$$

由此, 可以通过递归（辗转求余法, 类似于辗转相除法）设计求解最大公约数的高效算法.

练习 4.2 编写函数, 判断一个整数是否为素数, 并在主函数中找出三位数中所有的素数, 在屏幕上输出时每行输出 8 个.（程序取名 `hw04_02.cpp`）

```
bool is_prime(int n);
```

练习 4.3 编写函数, 判断给定的年份是否为闰年, 并在主函数中输出 21 世纪（2001 至 2099 年）中所有的闰年, 每行输出 6 个.（闰年: 能被 400 整除; 或者能被 4 整除但不能被 100 整除）（程序取名 `hw04_03.cpp`）

```
bool is_leap_year(int year);
```

练习 4.4 编写程序, 用 while 实现猜数游戏.（程序取名 `hw04_04.cpp`）

练习 4.5 Emirp 数: 如果一个素数反转后仍然是素数, 则称这个素数为 emirp 数. 如 13 是素数, 反转后 31 也是素数, 故 13 和 31 都是 emirp 数. 编写程序, 输出前 100 个 emirp 数, 每行输出 5 个. 要求: 先编写两个函数: `is_prime` 和 `reverse`, 分别用于判断素数和计算反序数. (程序取名 `hw04_05.cpp`)

```
bool is_prime(int n);
int reverse(int n);
```

练习 4.6 梅森素数: 如果一个素数可以写成 2^p-1 的形式, 则称该素数为梅森素数. 编写程序, 找出所有 $p<32$ 的梅森 (Mersenne) 素数, 并以如下的形式输出. (程序取名 `hw04_06.cpp`)

```
2    3
3    7
5    31
...  ...
```

练习 4.7 $3n+1$ 问题: 给定一个正整数 n, 不断按照下面的规律进行运算: 如果当前数是偶数, 则下一个数为当前数除以 2, 如果当前数为奇数, 则下一个数为当前数乘 3 加 1. 整个过程直到当前数是 1 为止. 这样形成的数列的长度称为数 n 的链数. 如: 从 3 开始, 得到的数列为: 3, 10, 5, 16, 8, 4, 2, 1, 所以整数 3 的链数为 8.

(a) 编写一个函数 (函数名 `num_chain`), 使用循环方法计算给定的正整数的链数;

(b) 找出 $[90,100]$ 中, 链数最大的那个数.

(程序取名 `hw04_07.cpp`)

```
int num_chain(int n); // 使用循环方法计算链数
```

练习 4.8 与上题相同, 但要求使用递归实现函数 `num_chain` (程序取名 `hw04_08.cpp`)

```
int num_chain(int n); // 使用递归方法计算链数
```

练习 4.9 编写函数, 用递归方法计算 Fibonacci 数, 并在主函数中计算第 40 个 Fibonacci 数. (程序取名 `hw04_09.cpp`)

```
long fibo(int n);
```

练习 4.10 给定一个正整数, 使用递归方法找出其所有的素数因子, 包含重复的. 比如 12 的所有素数因子为 2, 2, 3. (提示: 先找出最小的素数因子, 然后除以这个数, 得到商, 再找商的最小素数因子, 以此类推, 构成递归. 注意递归的出口. 只要在屏幕上输出这些素数因子, 不用保存.) (程序取名 `hw04_010.cpp`)

思考: 如果要求素数因子不能重复, 比如 12 的素数因子为 2, 3, 如何实现?

练习 4.11 蒙特卡罗 (Monte Carlo) 方法计算下面定积分 $\int_0^{\frac{\pi}{2}} \sin(x)\mathrm{d}x$ 的近似值. (提示: 利用定积分的几何意义, 即定积分与曲边梯形面积之间的关系) (程序取名 `hw04_11.cpp`)

练习 4.12 试统计汉诺塔问题总的移动步数.（程序取名 `hw04_12.cpp`）

练习 4.13 编写函数, 用于计算 $\log_x y$, 函数取名 `mylog`, 要求在调用该函数时: (1) 如果所给的 x 和 y 的值都大于 0, 则返回 $\log_x y$ 的值; (2) 如果只给 y 的值, 且其值大于 0, 则返回 $\log_2 y$ 的值（即形参 x 的缺省值为 2）; (3) 如果所给的 x 或 y 的值有负数, 则输出一个错误信息, 并返回 -1.（程序取名 `hw04_13.cpp`）

第五讲 数组

数组是科学计算和数据分析的主要处理对象, 是具有一定顺序关系的若干相同类型数据的集合, 是基本数据类型的推广.

本讲主要内容

- 一维数组
  - ▷ 一维数组的定义与引用
  - ▷ 一维数组在内存中的存放方式
  - ▷ 一维数组的赋值与初始化
- 二维数组
  - ▷ 二维数组的定义与引用
  - ▷ 与一维数组的关系, 在内存中是怎么存放的
  - ▷ 二维数组的赋值和初始化
- 数组作为函数参数
  - ▷ 数组中的单个元素作实参: 值传递
  - ▷ 数组名作为参数: 传递整个数组, 地址传递
- 字符串（字符数组, C 语言）
  - ▷ 字符串的表示
  - ▷ 字符串输入输出
  - ▷ 字符串操作 — 相关函数
  - ▷ 字符操作函数

5.1 一维数组

- 一维数组的声明:

```
类型说明符  数组名[n]  // 声明一个长度为 n 的一维数组（向量）
```

(1) 类型说明符: 数组元素的数据类型;
(2) 数组名代表的是数组在内存中的**首地址**;
(3) n 为数组的长度, 可以是一个表达式, 也可以从键盘输入, 但其值必须是一个确定的正整数.

- 一维数组的引用:

```
数组名[k]  // 注: 下标 k 的取值为 0 到 n-1
```

(1) 只能通过循环逐个引用数组元素;
(2) 数组元素在内存中是按顺序连续存放的, 它们在内存中的地址是连续的;

(3) 数组名代表整个数组在内存中的首地址.

✍ **注记：** 注意: 数组的下标不能越界, 否则可能会引起严重的后果!

例 5.1 一维数组举例. (ex05_array_01.cpp)

- 一维数组的初始化: 在声明时可以同时赋初值.

```
1  int x[5] = {0,2,4,6,8};
```

(1) 全部元素都初始化时可以不指定数组长度, 系统会根据所给的数据自动确定数组的长度, 如 `int x[] = {0,2,4,6,8};`

(2) 可以部分初始化, 即只给部分元素赋初值, 如 `int x[5] = {0,2,4};`

(3) 若数组声明时进行了部分初始化, 则没有初始化的元素自动赋值为 0;

(4) 声明数组时, 若长度为一个表达式, 且含有变量, 则不能初始化!

- ▷ 只能对数组元素赋值, 不能对数组名赋值!
- ▷ 若数组元素没有赋值, 则其值是不确定的（静态类型数据除外）;
- ▷ 注意**数组声明**与**数组引用**的区别;
- ▷ 注意**数组初始化**与**数组赋值**的区别.

5.2 二维数组

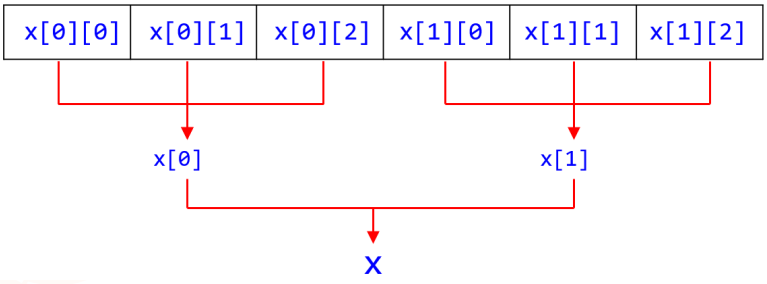
- 二维数组的声明: m 行 n 列

```
类型说明符 变量名[m][n] // 声明一个 m x n 的二维数组（矩阵）
```

- 二维数组的引用: i 的取值是 0 到 $m-1$, j 的取值是 0 到 $n-1$

```
变量名[i][j] // 注意下标不要越界!
```

- 二维数组的存储: 按行存储.



✍ **注记：** 在 C++ 中, 二维数组可以看作是由一维数组所组成的数组.

- 二维数组的初始化

(1) 全部初始化, 此时可以省略第一维的大小（但不能省略其他维数）

```
1  int x[2][3] = {1,3,5,2,6,10};
2  int x[][3] = {1,3,5,2,6,10}; // 省略第一维的大小
```

(2) 可以分组初始化, 也可以部分初始化

```
int x[2][3] = {{1,3,5}, {2,6,10}};
int x[2][3] = {{1}, {2,6}}; // 部分初始化
```

例 5.2 计算 n 阶 Hilbert 矩阵与全是 1 的向量的乘积, 其中 (ex05_array_hilb.cpp)

$$H = [h_{ij}]_{i,j=1}^{n}, \quad h_{ij} = \frac{1}{i+j-1}.$$

高维数组

高维数组的声明、赋值、引用、初始化与二维数组类似.

```
类型说明符 变量名[n1][n2][n3]...
```

5.3 数组作为函数参数

函数间涉及数组元素传递时, 有两种情形: 传递单个元素和传递整个数组.

- 传递 **单个元素**: 形参是普通变量, 实参是数组元素, 采用值传递方式.

```
void my_swap(int a, int b)
{ ... }

int main()
{
    int x[2] = {1,3};
    my_swap(x[0], x[1]);
}
```

- 传递 **整个数组**: 数组名作为实参, 通过 **地址传递** 实现, 即传递数组的首地址.
  (1) 形参和实参都是数组名, 类型一致;
  (2) 实参与形参代表的是同一个数组, 即 **在函数中对形参的任何改动都会影响到实参**;
  (3) 数组名作为函数参数, 传递的是首地址, 但不会传递数组长度.

例 5.3 交换两个数组. (ex05_array_swap.cpp)

例 5.4 数组名传递的是首地址, 不会传递数组长度. (ex05_array_sizeof.cpp)

几点说明

▷ 若形参指定了长度, 则必须是常量 (通常是全局常量);
▷ 为增加灵活性, 形参可以省略第一维的大小: 一维数组可以省略长度, 二维数组可以省略行数;

▷ 如果形参没有指定第一维的大小, 则需另加一个形参, 用来传递实参数组的大小, 或者通过全局变量实现;

▷ 函数调用时, 实参只需用数组名.

```
void my_swap(int a[], int b[], int n); // 可以省略长度，但中括号不能省
void sum_col(double A[][n], double s[]); // 这里的 n 必须是常量
  ...
int main()
{   ...
    sum_col(H, s); // 实参只需用数组名
    ...
}
```

例 5.5 计算矩阵各列的和（函数形式）. (ex05_array_fun.cpp)

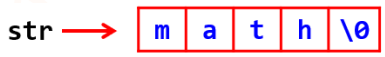
5.4 字符串（字符数组）

本节主要介绍 C 语言中字符串的实现与运用, 在 C++ 中同样适用.

5.4.1 字符数组

- 字符串: 在 C 语言中, 字符串是通过字符数组来实现的.
  (1) 字符串以 "\0" 为结束标志（称为**字符串结束标志符**）;
  (2) 使用双引号表示的字符串常量进行初始化时, 会自动添加结束标志符.

```
char str[5] = {'m','a','t','h','\0'}; // OK，只能用于初始化
char str[5] = "math"; // OK，只能用于初始化
char str[] = "math"; // OK，只能用于初始化
```



✍ **注记:** 字符串可以看作是由字符组成的数组, 但与普通数组 (数值型数组) 不同, 在运用上更加灵活方便.

- 字符串的输出, 有两种方式:
  (1) 利用循环逐个输出: 按普通数组方式输出, 但只输出其中有意义的字符, 即字符串结束标志符前面部分, 字符串结束标志符及其后面的部分无需输出.

```
char str[20] = "C++ and Matlab"; // 字符数组的长度为 20，但仅包含 14 个字符
for(int i = 0; i < 20; i++) // 逐个输出，只输出其中有意义的字符
    if (str[i] != '\0')
        cout << str[i];
    else
```

```
6           break;
```

(2) 利用字符串的特殊性, 整体输出.

例 5.6 字符串输出示例. (ex05_str_cout.cpp)

```
1  cout << str << endl; // 整体输出，自动逐个输出字符，直到遇见字符串结束标志符为止
```

✍ **注记**：输出的字符串中不含 **字符串结束标志符** “\0”

- 字符串的输入（ex05_str_cin_getline.cpp）

```
cin >> str;       // 输入单个字符串，中间不能有空格
cin.getline(str,N,结束符); // 整行输入
```

▷ cin.getline: 连续输入多个字符 (可以有空格), 直到读入 $N-1$ 个字符为止, 或遇到指定的 结束符, 结束符 不会被读入. 结束符 可以省略, 缺省为 '\n', 即换行符, 此时将整行作为输入.

- 字符串常用函数（头文件 cstring 和 cstdlib）

函数	含义
strlen(str)	字符串长度
strcat(dest,src)	字符串连接
strcpy(dest,src)	字符串复制
strcmp(str1,str2)	字符串比较
atoi(str)	将字符串转换为整数
atol(str)	将字符串转换为 long
atof(str)	将字符串转换为 double
itoa(int,str,raidx)	按指定进制 raidx 将整数转换为字符串

✍ **注记**：这些函数只能作用在字符串上, 不能作用在字符上.

例 5.7 字符串相关函数. (ex05_str_fun.cpp)

```
1  strlen(str);       // 返回字符串 str 的长度（不含结束符）
2  strcat(str1,str2); // 将 str2 的全部内容添加到 str1 后面，str1 要有足够空间
3  strncat(str1,str2,n); // 将 str2 的内容添加到 str1 中，但至多添加 n 个字符
4  strcmp(str1,str2); // 按字典顺序比较大小
5        // 若 str1>str2 则返回一个正数，若 str1<str2 则返回一个负数，相等则返回 0
6  strncmp(str1,str2,n); // 按字典顺序比较 str1 和 str2 的前 n 个字符的大小
7  strcpy(str1,str2); // 将 str2 拷贝到 str1 中 (str1 的长度不能小于 str2 的长度)
8  strncpy(str1,str2,n); // 拷贝前 n 个字符，若 n 大于 str2 的长度，则拷贝全部内容
```

例 5.8 字符串与数字. (ex05_str_atoi.cpp)

```
int x;
double y;
x = atoi("66"); // x = 66, 字符串中只能包括数字
y = atof("14.5"); // y = 14.5, 字符串中只能包括数字和小数点
```

5.4.2 字符操作

- 单个字符的输入: cin, getchar （ex05_char.cpp）

```
cin >> ch;  // ch 为字符型变量
ch = getchar();
```

- C++ 字符检测函数（头文件 cctype）

函数	含义	示例
isdigit	是否为数字	isdigit('3')
isalpha	是否为字母	isalpha('a')
isalnum	是否为字母或数字	isalnum('c')
islower	是否为小写	islower('b')
isupper	是否为大写	isupper('B')
isspace	是否为空格	isspace(' ')
isprint	是否为可打印字符, 包含空格	isprint('A')
isgraph	是否为可打印字符, 不含空格	isgraph('a')
ispunct	除字母数字空格外的可打印字符	ispunct('*')
iscntrl	是否为控制符	iscntrl('\n')
tolower	将大写转换为小写	tolower('A')
toupper	将小写转换为大写	toupper('a')

✍ **注记：** 以上检测和转换函数只针对单个字符, 而不是字符串.

- 字符与整数的运算: 字符参加算术运算时, 自动转换为整数 (按 ASCII 码转换)

```
char x = '2';
int y = x;
int z = x-'0';
cout << "x = " << x << endl; // x = '2' 是字符
cout << "y = " << y << endl; // y = 50 是整数, 即字符 '2' 的 ASCII 码
cout << "z = " << z << endl; // z = 2 是整数
```

5.5 上机练习

练习 5.1 均值与标准差: 给定一组数 $x_1, x_2, \ldots, x_n$, 其均值和标准偏差分别定义为:

$$\text{mean} \triangleq \frac{x_1 + x_2 + \cdots + x_n}{n}, \qquad \text{deviation} \triangleq \sqrt{\frac{\sum_{i=1}^{n}(x_i - \text{mean})^2}{n-1}}.$$

编写程序, 生成 100 个随机双精度数, 计算它们的均值和标准偏差. 要求: (1) 编写两个函数: `mean` 和 `deviation`, 分别计算一个数组的均值和标准偏差; (2) 在主函数中生成一个长度为 100 的随机双精度数组 (元素的值在 $[-10, 10]$ 内), 然后通过调用上面两个函数来计算其均值和标准偏差. (程序取名 `hw05_01.cpp`)

```
double mean(double x[], int n);
double deviation(double x[], int n);
```

练习 5.2 统计数字出现次数: 编写程序, 随机生成 100 个 1 9 之间的整数, 统计每个数字的出现次数. (程序取名 `hw05_02.cpp`)

练习 5.3 反转数组: 编写一个函数, 反转一个数组, 并在 `main` 函数中生成一个长度为 10 的随机整数数组, 然后输出其反转后的数组. (程序取名 `hw05_03.cpp`)

```
void reverse(int x[], int n);
```

练习 5.4 矩阵乘积: 编写函数, 计算两个 5 阶矩阵的乘积 $Z = X * Y$. 在主函数中生成两个 5 阶的随机矩阵, 元素为在 0 到 9 之间的正整数, 然后计算它们的乘积, 并将这三个矩阵输出. (程序取名 `hw05_04.cpp`)

```
const int N = 5;
void matrix_prod(int X[N][N], int Y[N][N], int Z[N][N]);
```

练习 5.5 找最小值的位置: 编写函数, 找出一维数组的最小值所在下标 (如果有多个最小值, 则输出第一个最小值的下标), 并在主函数中以数组 $[34, 91, 85, 59, 29, 93, 56, 12, 88, 72]$ 为例, 找出其最小数及其下标. (程序取名 `hw05_05.cpp`)

```
int findmin(int a[], int n); // 返回最小数所在下标
```

练习 5.6 有序数组中插入新元素: 编写函数 `insert`, 实现下面功能: 给定已经从小到大排列好的 n 个数, 在主函数中输入一个数, 调用 `insert` 函数, 把输入的数插入到原有数列中, 保持大小顺序, 并将被挤出的最大数 (有可能就是被插入数) 返回给主函数输出. 编写程序, 以 $[12, 29, 34, 56, 59, 72, 85, 88, 91, 93]$ 为例. (程序取名 `hw05_06.cpp`)

```
int insert(int a[], int n, int x);
```

练习 5.7 储物柜问题: 学校有 100 个储物柜, 100 个学生. 开学第一天所有储物柜都是关闭的, 第一个学生到校后将所有储物柜打开; 第二个学生到校后, 从第二个储物柜开始, 每隔 1 个储物柜, 将它们

关闭; 第三个学生到校后, 从第三个储物柜开始, 每隔 2 个, 将它们的状态改变, 即开着的关闭, 关闭的打开. 依此类推, 直至第 100 个学生到校后将第 100 个储物柜的状态改变. 问: 当所有学生都完成这个过程后, 有哪些储物柜是开着的? 编写程序求解该问题. (程序取名 hw05_07.cpp)
(提示: 使用一个数组, 保存每个储物柜的状态改变次数, 如果一个储物柜的状态改变次数为奇数, 则该储物柜是开着的.)

练习 5.8 **找最小值的位置**: 编写函数, 找出给定二维数组中最小值的下标 (i, j), 如果有多个最小值, 则输出第一个 (行标最小) 的下标. 并在主函数中以一个 8 阶随机矩阵 (其元素为 $[-1, 1]$ 中的随机小数) 为例, 在屏幕上输出其最小数及其下标. (程序取名 hw05_08.cpp)

```
void findmax2D(double A[M][N], int idx[2]); // 下标存放在数组 idx 中
```

第六讲 指针

指针变量, 简称**指针**, 用来存放其它变量的 **内存地址**. 通过指针, 可以直接访问系统内存, 从而提高程序执行效率.

本讲主要内容

- 为什么指针
  - 什么是指针, 指针与内存
  - 指针的定义与运算
  - 指针与一维数组, 指针与二维数组
  - 指针数组
  - 指针与引用
  - 指针与函数: 指针作为函数参数, 指针型函数, 指向函数的指针
- 持久动态内存分配
  - 动态内存申请: `new`
  - 动态内存释放: `delete`
  - 动态数组的申请与释放
  - 智能指针

内存管理

操作系统负责对内存的管理, 内存的每个字节都有一个编号, 即 **内存编号**, 通常也称为 **内存地址**.

✍ **注记:** 程序中的变量、函数等, 在内存中都有相应的地址.

6.1 指针与内存

- 指针的声明

  ```
  类型说明符 * 指针变量名
  ```

  (1) 类型说明符: 指定指针类型, 表示该指针可指向的对象的数据类型, 即该指针能存放哪类数据的地址.

  (2) 星号和指针变量名之间可以不加空格.

- 指针的两个基本运算

  (1) 提取变量的内存地址: `&变量名`

  (2) 引用指针所指向的变量（即目标对象）: `*指针`

- 指针的初始化: 声明指针变量时, 可以赋初值.

```
int x;
int * px = &x; // 初始化，px 中存放 x 的内存地址
```

✍ **注记：** 此时，我们通常称 px 是指向 x，称 x 为 px 的目标对象. 指针的类型必须与目标对象的数据类型一致. 在使用指针时，我们通常关心的是指针的目标对象！

- 指针赋值: 给指针赋值时，只能使用以下的值
  - ▷ 空指针: 0, NULL 或值为 0/NULL 的常量. C++11 标准中引入了 nullptr，用于表示空指针，避免使用 0 或 NULL 表示空指针时可能引发的歧义问题，增强了类型安全性.
  - ▷ 类型匹配的目标对象的地址;
  - ▷ 同类型的另一个有效指针;
  - ▷ 类型匹配的对象的前后地址 (相对位置).

✍ **注记：** 没有初始化或赋值的指针是 **无效的指针**，也称为 **野指针**，引用无效指针会带来难以预料的后果.

例 6.1 指针的定义与使用 (ex06_pointer_01.cpp)

Tips: 内存空间的访问方式: 1) 变量名; 2) 内存地址，即指针. 指针提供了一种访问变量的高效方法.

- void 类型的指针

```
void * 指针名
```

(1) void 类型的指针可以存储任何类型的对象的地址，因此也称为 **万能指针** 或 **乱指针**;
(2) 必须通过**显式类型转换**，才可以访问 void 指针的目标对象.

```
int x;
void * pv;
pv = &x;        // OK，void 型指针指向整型变量
*((int *)pv) = 3; // OK，使用 void 型指针时需要强制类型转换
```

✍ **注记：** 通过指针来操作其指向的对象时，不仅需要知道对象的地址（即指针本身的值），还需要知道对象的数据类型（通过指针的数据类型获取），否则就无法正确解析对象的值（本质上只是一串二进制数）.

例 6.2 指针与内存地址 (ex06_pointer_02.cpp)

- 指向常量的指针

```
const 类型说明符 * 指针名
```

(1) 这里的 const 限定了指针所指对象的属性, 即目标对象是常量, 而不是指针本身是常量;
(2) 允许把非 const 对象的地址赋给指向常量的指针, 即指向常量的指针也可以指向普通变量;
(3) 不允许使用指向常量的指针来修改其目标对象的值, 即使它所指向的对象不是常量.

```
const int a = 3; // 常量
int b = 5;
const int * cpa = &a; // OK, 指向常量的指针
*cpa = 5; // ERROR, 不允许通过指向常量的指针来修改其目标对象的值
cpa = &b; // OK, cpa 也可以指向普通变量
*cpa = 9; // ERROR, 虽然 b 不是常量, 但也不能通过 cpa 修改其值
b = 9;  // OK
```

- 常量指针, 简称常指针: 指针本身的值不能修改.

```
类型说明符 * const 指针名
```

```
int a = 3, b = 5;
int * const pa = &a; // OK
pa = &b; // ERROR, pa 本身是常量
```

- 指向常量的常指针

```
const 类型说明符 * const 指针名
```

(1) 指针本身的值不能修改, 也不能通过该指针修改其目标对象的值.

- 指针算术运算: 指针可以和整数或整型变量进行加减运算, 运算规则与指针的类型有关, 通常配合数组使用.

```
int x[5] = {0,1,2,3,4};
int * px = &x[0]; // px 指向 x[0]
cout << *px << endl; // 输出 x[0] 的值
cout << *(px+1) << endl; // 输出的 x[1] 值
px = px + 2; // px 改为指向 x[2]
```

- 指针数组: 由指针变量组成的数组

```
类型说明符 * 指针数组名[n]
```

例 6.3 指针数组 (ex06_pointer_array.cpp)

6.2 指针与一维数组

由于数组元素在内存中是连续存放的, 因此使用指针可以非常方便地处理数组元素.

引用数组元素的四种方式

(1) 数组名与下标;
(2) 数组名与指针运算;
(3) 指针与指针运算;
(4) 指针与数组运算.

例 6.4 指针与一维数组: 引用数组元素的四种方式. (ex06_pointer_array_01.cpp)

✍ **注记:** 数组名代表数组的首地址, 当数组名出现在表达式中时, 等效于一个常指针.

```
1 int a[] = {0,2,4,8};
2 int * pa = a; // OK, 数组名代表数组的首地址, 即 pa = &a[0]
3 *pa = 3;     // OK, 等价于 a[0] = 3
4 *(pa+2) = 5;  // OK, 等价于 a[2] = 5
5 *(a+2) = 5;  // OK, 等价于 a[2] = 5
6 *(pa++) = 3;  // OK, 等价于 a[0] = 3; pa = pa + 1;
7 *(a++) = 3;  // ERROR! a 代表数组首地址, 等效于常指针!
```

小结: 一维数组 a[n] 与指针 pa=a

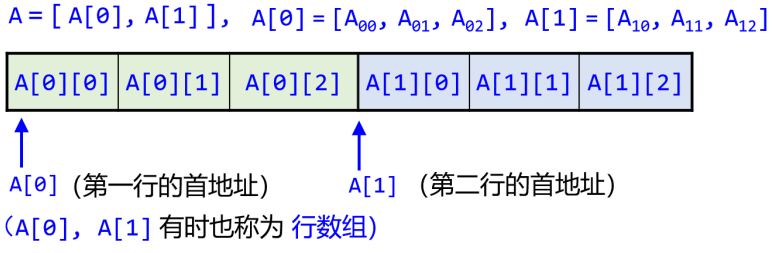
(1) 一维数组名 a 是地址常量, 数组名 a 与 &a[0] 等价;
(2) a+i 是 a[i] 的地址, a[i] 与 *(a+i) 等价;
(3) 若指针 pa 存储的是数组 a 的首地址, 则 *(pa+i) 与 pa[i] 等价;
(4) 数组元素的下标访问方式也是按地址进行的;
(5) 指针的值可以随时改变, 通过指针访问数组的元素更加灵活;
(6) 数组名等效于常量指针, 值不能改变: pa++ 或 ++pa 合法, 但 a++ 不合法.

```
a[i]  <=> pa[i]  <=> *(pa+i)  <=> *(a+i)
```

6.3 指针与二维数组

- 在 C++ 中, 二维数组是 **按行存储** 的, 可以理解为由一维数组所组成的数组.

设 A 是一个 2×3 的整数矩阵, 比如 `int A[2][3]={{1,2,3},{7,8,9}};` 则矩阵 A 在内存中的存放情况如下:



即 A 有两个元素: A[0] 和 A[1], 而 A[0] 和 A[1] 又各自包含三个元素.

- 对于二维数组 A, 虽然 A、A[0] 都是数组首地址, 但二者指向的对象不同:
  (1) A[0] 是一维数组名, 代表第一行, 指向的是它的首元素 A[0][0] , 即 *A[0] 与 A[0][0] 等价;
  (2) A 是二维数组名, 指向的是它的首元素 A[0], 因此 *A 与 A[0] 等价. 另外, 它的指针移动单位是“行”, 所以 *(A+i) 对应的是第 i 个行数组, 即 A[i].

*A[0] ⟷ A[0][0]
*(A[0]+1) ⟷ A[0][1]

*A ⟷ A[0]
*(A+1) ⟷ A[1]

*(*A) ⟷ *A[0] ⟷ A[0][0]
*(*A+1) ⟷ *(A[0]+1) ⟷ A[0][1]

```
int A[2][3]={{1,2,3},{7,8,9}};
int * p = A[0]; // OK, p 指向 A[0][0]
int * p = A; // ERROR! p 只能指向一个普通整型变量, 不能是行数组
```

- 指针与二维数组: 设指针 pa = &A[0][0], 则

```
A[i][j]  <=> *(pa+n*i+j)    // 这里 n 是 A 的列数
```

例 6.5 指针与二维数组举例. (ex06_pointer_array_2D.cpp)

指针与引用

- 指针与引用的区别:
  (1) 引用是变量的别名, 必须初始化, 且不能修改;
  (2) 引用只针对变量, 函数没有引用;
  (3) 引用能实现的功能, 用指针都能实现;

```
int a = 3;
int * pa = &a; // 指针
int & ra = a; // 引用
```

- 引用的优点: 底层实现与指针类似, 但语义更为明确且职责单一, 可读性更强, 更利于编译器优化; 同时, 引用作为函数的形参, 函数调用简单、安全.

✍ **注记:** C 语言只有指针, 没有引用, 引用是 C++ 新增的语法, 是在某些情况下实现指针作用的简便写法, 通常能使用引用的时候建议使用引用.

✍ 在函数的参数传递中, 如果传递的是数组, 建议使用指针. 如果想使用引用的话, 需要使用针对数组的引用, 不能使用普通的引用.

```
int a[5];
int & ra = (&a)[5]; // 针对数组的引用
```

6.4 行指针与二级指针 *

6.4.1 行指针

- 行指针, 也称为指向一维数组的指针: 以一维数组为基本单位, 即将一维数组看作一个整体.

```
类型说明符 (* 指针名)[n]
```

▷ 定义一个指向长度为 n 的一维数组的指针.

```
int A[3][4];
int (*pA)[4] = A; // OK! 注意与指针数组的区别, 也不能写成 (*pA)[3]
... ...
for(int i = 0; i < m; i++)
{
    for (int j = 0; j < n; j++)
        cout << *(*(pA + i) + j );
}
```

例 6.6 行指针与二维数组举例. (ex06_pointer_row.cpp)

6.4.2 二级指针

- 二级指针: 指向指针的指针

```
类型说明符 ** 指针名
```

▷ 二级指针存放的是另一个指针的地址

```
int a = 3;
int *pa = &a;
int **ppa = &pa;
```

例 6.7 二级指针举例. (ex06_pointer_pointer.cpp)

注记: 二级指针在数据结构中有着重要的应用, 特别是链表结构.

注记: 类似地, 可以定义更高级的指针. 事实上, C/C++ 语言不限制指针的级数, 每增加一级指针, 在定义指针变量时增加一个星号. 实际开发中可能会经常使用一级指针和二级指针, 但很少用到高级指针.

```
int a = 2024;
int *pa = &a;
int **ppa = &pa;
```

```
4  int ***pppa = &ppa;
5  cout << a << ", " << *pa << ", " << **ppa << ", " << ***pppa << endl;
```

6.5 指针与函数

- 指针作为函数参数:
  (1) 指针作为函数参数时, 以地址方式传递数据;
  (2) 形参是指针时, 实参可以是同类型指针或地址;

例 6.8 指针作为形参：取整数部分与小数部分. (ex06_pointer_arguments.cpp)

注记： 当函数间需要传递大量数据时, 开销会很大. 此时, 如果数据是连续存放的, 则可以只传递数据的首地址, 这样就可以减小开销, 提高执行效率!

Tips: 如果被调函数不需要改变实参的值, 则可以将形参中的指针声明为指向常量的指针.

关于主函数的形参

主函数也可以带形参:

```
int main(int argc, char* argv[])
```

这里 argc 代表命令行字符串个数 (包括命令本身), argv[] 存储命令行的所有字符串, 其中 argv[0] 通常是命令本身.

- 指针型函数: 函数的返回值是地址或指针, 一般形式如下:

```
类型说明符 * 函数名（形参列表）
{
    函数体
}
```

- 指向函数的指针, 即 **函数指针**
  (1) 在程序运行过程中, 不仅数据要占用内存空间, 函数也要在内存中占用一定的空间;
  (2) 函数名就代表函数在内存空间中的首地址;
  (3) 用来存放这个地址的指针就是指向该函数的指针;
  (4) 函数指针的定义:

```
类型说明符 (* 函数指针名)(形参列表)
```

  (5) 这里的类型说明符和形参列表应与其指向的函数相同;
  (6) 函数名除了表示函数的首地址外, 还包括函数的形参, 返回值类型等信息;
  (7) 可以象使用函数名一样使用函数指针.

例 6.9 函数指针举例. (ex06_pointer_fun.cpp)

6.6 持久动态内存分配

若在程序运行之前, 不能够确切知道数组中元素的个数, 如果声明为很大的数组, 则可能造成浪费, 如果声明为小数组, 则可能不够用. 此时需要动态分配空间, 做到按需分配. 此时可以使用 C++ 提供的持久动态内存分配方法.

✍ **注记**: 动态变量通常存放在内存的 **栈** 中, 从栈中读写数据相对比较高效。但栈通常比较小, 如果需要声明一个大数组, 则无法存放在栈中, 只能存在到 **堆** 中, 这时就需要用到持久动态内存分配功能.

- 申请单个存储单元

```
px = new 类型说明符;
px = new 类型说明符(初始值);
```

(1) 申请用于存放指定数据类型数据的内存空间;
(2) 若申请成功, 则返回该内存空间的首地址, 并赋给指针 px;
(3) 若申请不成功, 则返回 0 或 NULL.

- 释放由 new 申请的存储单元

```
delete px;
```

(1) px 必须是 new 操作的返回值.
(2) 通过 new 申请的存储空间一定要通过 delete 手工释放, 否则容易造成内存泄露.

- 创建一维动态数组

```
px = new 类型说明符[n];
px = new 类型说明符[n](); // 全部赋初值 0
px = new 类型说明符[n]{初值列表}; // C++11 标准支持
```

(1) n 是数组长度, 可以是整型变量或表达式, 但要有确定的值.

- 动态数组的释放

```
delete[] px; // 注意中括号是在 delete 后面, 不是在数组名后面!
```

例 6.10 创建一维动态数组. (ex06_new_01.cpp)

- 创建和释放多维动态数组

```
px = new 类型说明符[n1][n2]...[np];
delete[] px;
```

(1) n1 可以是整型变量或表达式, 要有确定的值, n2,...,np 必须是常量.
(2) 为了使用更加方便灵活, 实际使用时通常用一维数组代替二维数组.

例 6.11 动态数组: 给定一个正整数 N, 寻找前 N 个素数, 并在屏幕上输出. (ex06_new_02.cpp)

例 6.12 动态数组: 二维数组, 通过一维数组实现. (ex06_new_03_matrix.cpp/04/05)

被调函数返回大量数据 (数组) 给主调函数的两种方式

(1) 在主调函数中申请内存空间 (数组), 将首地址传递给被调函数, 再由被调函数对数组进行更新.
(2) 在被调函数中申请持久动态内存空间 (数组), 更新数据后返回该数组的首地址. 但需要注意的是, 此时被调函数申请的持久动态内存不能在被调函数中释放, 要在主调函数中释放.

6.6.1 智能指针

智能指针主要是为了方便管理动态分配的内存, 避免内存泄露. C++11 标准引入了三个智能指针: unique_ptr, shared_ptr 和 weak_ptr.

✍ **注记:** C++98 也有智能指针, 即 auto_ptr, 但在 C++11 标准中已经被废弃.

- 智能指针 unique_ptr
  - ▷ 独占资源所有权, 同一时间只能有一个 unique_ptr 指向特定内存.
  - ▷ 离开 unique_ptr 对象的作用域时, 会自动释放资源.

例 6.13 智能指针 unique_ptr 举例. (ex06_pointer_unique.cpp)

```
1 unique_ptr<int> px(new int{2}); // 申请动态内存并初始化
2 unique_ptr<int> px{new int{2}}; // 另外一种初始化方式
3 unique_ptr<int[]> py(new int[3]{2,4,6}); // 动态数组
```

✍ **注记:** 不能将 unique_ptr 直接赋值给其他指针, 但可以转移.

```
1 unique_ptr<int> px(new int{2});
2 unique_ptr<int> pz;
3
4 pz = px; // ERROR
5 pz = move(px); // OK, 将 px 的所有权移交给 pz, 移交后 px 就不再拥有资源的所有权
```

- 智能指针 shared_ptr
  - ▷ 共享资源所有权, 多个 shared_ptr 可以指向同一内存资源, 当最后一个 shared_ptr 被销毁时自动释放资源. 可以理解为对内存资源做引用计数, 当引用计数为 0 时, 自动释放资源.

例 6.14 智能指针 shared_ptr 举例. (ex06_pointer_shared.cpp)

- 智能指针 week_ptr
  - ▷ 与 shared_ptr 一起使用, 一个 weak_ptr 对象看做是 shared_ptr 管理的资源的观察者, 它不影响共享资源的生命周期.
  - ▷ 需要使用 weak_ptr 正在观察的资源, 可以将 weak_ptr 提升为 shared_ptr.
  - ▷ 当 shared_ptr 管理的资源被释放时, weak_ptr 会自动变成 nullptr.

6.7 应用: 矩阵乘积的快速算法

见课程主页, 或附录 ??.

6.8 应用: Gauss 消去法求解线性方程组

见课程主页, 或附录 ??.

6.9 上机练习

练习 6.1 编写函数, 交换两个双精度变量的值, 分别用引用和指针实现. 函数名分别为 swap_ref 和 swap_pointer, 并在主函数中生成两个双精度变量, 从键盘接受输入, 然后进行交换, 并将交换前、后的值都在屏幕上输出. (程序取名 hw06_01.cpp)

```
void swap_ref(double & ra, double & rb);
void swap_pointer(double* pa, double* pb);
```

练习 6.2 给定两个一维数组 a 和 b, 其中 a 中的数据是无序的, 而 b 中的数据按升序排列. 试统计 a 的所有元素中, 大于 b 的第 k 个元素且小于第 $k+1$ 个元素的数据个数, 其中

$$a = [98, 12, 34, 71, 43, 54, 28, 33, 65, 56], \quad b = [10, 30, 50, 80, 100].$$

要求将结果存放在数组 c 中, 其中 $c[k]$ 表示数组 a 中大于 $b[k]$ 而小于 $b[k+1]$ 的元素个数. (程序取名 hw06_02.cpp)

练习 6.3 **围圈报数**: 有 17 人围成一圈, 编号分别为 1 至 17, 从 1 号开始报数, 报到 3 的倍数的人离开, 一直数下去, 直到最后只剩一人, 编程输出最后一个人的编号. (程序取名 hw06_03.cpp)

练习 6.4 **二进制转十进制**: 编写函数, 将一个**用字符串表示**的二进制数转化为十进制数, 如“10001”所对应的十进制数为 17, 在主函数中用“1100110011001100”测试. (程序取名 hw06_04.cpp)

```
int bin2dec(const char* const str);
```

(提示: 将一个字符转化成数字, 可借助字符加减运算 (推荐) 或字符串函数)

练习 6.5 **字符易位破译**: 编写函数, 测试两个字符串是否字符异位相等, 即两个字符串中包含的字母是相同的, 但次序可以不同, 如“silent”和“listen”是字符异位相等, 但“baac”与“abcc”不是. (程序取名 hw06_05.cpp)

```
bool isAnagram(const char* const str1, const char* const str2);
```

(提示: 可以先对字符串进行排序, 然后再比较)

练习 6.6 编写函数, 实现两个有序 (升序) 数组的快速合并, 使得合并后的数组仍然有序, 并在主函数测试, 取 $x = [3, 5, 8, 11, 23, 39], y = [1, 5, 12, 13, 18, 41, 58, 68]$ (程序取名 hw06_06.cpp)

```
void vec_merge(int* px, int m, int* py, int n, int* pz);
```

（注: m 和 n 分别是数组 x 和 y 的长度, 合并后的数组存放在 z 中）

练习 6.7 找出前 100 个素数, 存放在数组 p 中, 并分别使用下列方式在屏幕上输出 p, 每行输出 10 个.（程序取名 hw06_07.cpp）

方式一: 数组名 + 下标运算;

方式二: 数组名 + 指针运算;

方式三: 指针 + 指针运算.

练习 6.8 **矩阵乘积**: 编写函数, 计算两个矩阵的乘积 $Z = X * Y$, 其中 $X \in \mathbb{R}^{m\times p}$, $Y \in \mathbb{R}^{p\times n}$, $Z \in \mathbb{R}^{m\times n}$, 要求对任意正整数 m, p, n 都能实现矩阵相乘.（程序取名 hw06_08.cpp）

```
void matrix_prod(double* X, double* Y, double* Z, int m, int p, int n);
```

练习 6.9 编写函数, 实现矩阵乘积的 Strassen 算法, 在主函数中生成两个 $n\times n$ 的随机整数矩阵进行测试, 比较与普通矩阵乘积的计算效率（消耗的时间）, 分别对 $n = 2^8, 2^{10}$ 进行测试.（提示：大矩阵需要用 new 来申请存储空间）（条件许可的话可以对更大的 n 进行测试, 如 $n = 2^{12}, 2^{14}$ 等）（程序取名 hw06_09.cpp）

```
void matrix_prod_strassen(double* px, double* py, double* pz, int n);
```

练习 6.10 编写函数, 实现求解线性方程组的 Gauss 消去法, 并在主函数中进行测试: 求解线性方程组 $Hx = b$, 其中

$$H = [h_{ij}] \in \mathbb{R}^{8\times 8}, \quad h_{ij} = \frac{1}{i+j-1}, i, j = 1, 2, \ldots, 8, \qquad b = [1, 1, \ldots, 1]^{\mathsf{T}}.$$

（程序取名 hw06_10.cpp）

```
void GE(double A[n][n], double b[n], double x[n], int n);
```

练习 6.11 编写函数, 实现求解线性方程组的 **列主元 Gauss 消去法**, 并在主函数中进行测试: 求解线性方程组 $Ax = b$, 其中

$$A = \begin{bmatrix} 0 & 1 & 1 & & & & & \\ 1 & 0 & 1 & 1 & & & & \\ 1 & 1 & 0 & 1 & 1 & & & \\ & 1 & 1 & 0 & 1 & 1 & & \\ & & 1 & 1 & 0 & 1 & 1 & \\ & & & 1 & 1 & 0 & 1 & 1 \\ & & & & 1 & 1 & 0 & 1 \\ & & & & & 1 & 1 & 0 \end{bmatrix}_{8\times 8}, \qquad b = [1, 1, \ldots, 1]^{\mathsf{T}} \in \mathbb{R}^8.$$

（程序取名 hw06_11.cpp）

```
void GEPP(double* a, double* b, double* x, int n);
```

第七讲 简单输入输出

本讲主要内容

- C++ 基本输入输出
  - 数据流
  - 操纵符
- C 语言格式化输出: `printf`
  - 格式控制字符串: 普通字符串, 格式字符串, 转义字符
- C 语言文件读写
  - 打开文件: `fopen`, 文件指针, 文件类型, 打开方式
  - 读、写文本文件: `fscanf`, `fprintf`, `fputc`, `fputs`
  - 读、写二进制文件: `fread`, `fwrite`
  - 关闭文件: `fclose`

7.1 C++ 基本输入输出 (I/O) 流

C++ 本身没有输入输出语句, 其输入输出是通过相应的 I/O 流类库实现的.

- 数据流: 将数据从一个对象到另一个对象的流动抽象为“流”.
  (1) 提取（读）: 从流对象中获取数据, 运算符为 “`>>`”;
  (2) 插入（写）: 向流对象中添加数据, 运算符为 “`<<`”;
  (3) 提取和插入运算符是所有标准 C++ 数据类型预先设计的;
  (4) 插入运算符与操纵符一起使用, 可以控制输出格式.
- 头文件 `iostream` 中预定义的四个输入输出对象

```
cin    标准输入（键盘等）
cout   标准输出（屏幕、打印机等）
cerr   标准错误输出，没有缓冲，立即被输出
clog   与 cerr 类似，但有缓冲，缓冲区满时被输出
```

- 常用操纵符（头文件 `iomanip`）

操作符	含义
`endl`	插入换行符, 并刷新流（将缓冲区中的内容刷入输出设备）
`setw(n)` `cout.width(n)`	设置域宽, 若数据超过设置的宽度, 则显示全部值

fixed	使用定点方式输出
scientific	使用指数形式输出
setfill(c) cout.fill(c)	设置填充, c 可以是任意字符, 缺省为空格, 如 setfill('*'), setfill('0')
setprecision(n)	设置输出的有效数字个数, 若在 fixed 或 scientific 后使用, 则设置小数位数
left	左对齐
right	右对齐 (缺省方式)
showpoint	显示小数点和尾随零 (即使没有小数部分)
defaultfloat	缺省浮点数输出格式 (C++11, GCC5)

✍ **注记:** 除 setw 和 cout.width 外, 其它操纵符一直有效, 直到遇到相同类型的操纵符为止.

例 7.1 操纵符的使用: 域宽和填充. (ex07_setw_fill.cpp)

例 7.2 操纵符的使用: 小数输出形式和精度设置. (ex07_setw_fill.cpp)

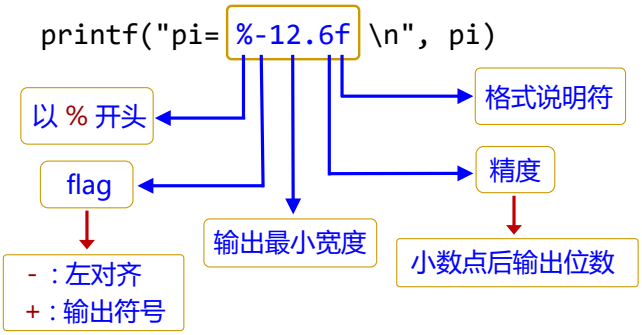
7.2 C 语言格式化输出

```
printf("格式控制字符串", 输出变量列表); // 需加头文件 cstdio
```

(1) **格式控制字符串**: 包括 "普通字符串"、"格式字符串"、"转义字符"

(2) 格式字符串: 以 % 开头, 后面跟**格式说明符**和其它选项 (格式说明符必须要有, 其他可选)

```
%[flag][输出最小宽度][.精度]格式说明符
```




```
1 int k=5;
2 double a=3.14;
3 printf("k=%d, a=%f\n", k, a); // 普通字符串按原样输出; 一个格式字符串对应一个变量.
```

- 常见的格式说明符

c	字符型	g	浮点数（系统自动选择输出格式）
d	十进制整数	o	八进制
e	浮点数（科学计数法）	s	字符串
f	浮点数（小数形式）	x/X	十六进制

- 常见的转义字符（输出特殊符号）

\b	退后一格	\t	水平制表符
\f	换页	\\	反斜杠
\n	换行	\"	双引号
\r	回车	%%	百分号

7.3 C 语言文件读写

C 语言提供了专门的函数来实现文件的读写, 即从文件中读取数据, 或者将数据写入文件中, 以便实现数据的长时间保存或者与其他程序的进行数据交换.

7.3.1 文件的打开和关闭

按存储方式 (即数据的组织形式), 文件可以分为: 文件文件和二进制文件.

文件读写分三个步骤: (1) 打开文件, (2) 进行相应的读取或写入操作, (3) 关闭文件.

- 在 C 语言中, 用 fopen 打开文件.

```
FILE * pf; // 声明一个文件指针
pf = fopen(文件名, 打开方式);
```

(1) fopen 按指定的方式打开文件, 返回一个文件指针, 之后就可以通过该文件指针进行读写操作.
(2) 文件指针由关键字 FILE 声明, 是头文件 stdio.h 中定义的一种特殊数据类型.
(3) 文件名: 普通字符串, 可包含路径, fopen 将文件名对应的文件与文件指针 pf 绑定在一起.
(4) 打开方式: 指定读写方式和文件类型, 取值有

```
rt、wt、at、rb、wb、ab、rt+、wt+、at+、rb+、wb+、ab+
r为读, w为写, +为读写, t为文本, b为二进制
```

(5) 若文件打开成功, 返回指向文件的指针; 若打开失败, 则返回一个空指针 (NULL)

- 文件关闭:

```
fclose(pf);
```

(1) 正常关闭则返回值为 0; 出错时, 返回值为非 0

7.3.2 文本文件的读写

- 写文本文件: fprintf

```
fprintf(pf, "格式控制字符串", 输出变量列表); // fprintf 的用法与 printf 类似
```

例 7.3 C 语言文件读写: 文本文件 fprintf (ex07_fprintf.cpp)

- 写文本文件: fputc 和 fputs

```
fputc(c, pf); // 写入单个字符, c 是待写入的字符, pf 是文件指针
fputs(str, pf); // 写入字符串, str 是待写入的字符串, pf 是文件指针
```

例 7.4 C 语言文件读写: 文本文件 fputc, fputs (ex07_fputs.cpp)

- 读文本文件: fscanf

```
fscanf(pf, "格式控制字符串", 地址列表); // 注意最后一个参数是地址
```

  - 从文件中读取数据 (可以是数值型或字符型).
  - 如果是读取数值型数据, 则建议文件中不要含有字符型数据, 并且每个数据单独占一行.
  - 如果是读取浮点数, 则格式控制字符串要使用 %lf, 不是 %f.
  - 如果是读取字符串, 则缺省以空格为结束符.

例 7.5 C 语言文件读写: 文本文件 fscanf (ex07_fscanf.cpp)

- 读文本文件: fgetc 和 fgets

```
c = fgetc(pf); // 从文件读取单个字符, 赋给 c
fgets(str, n, pf); // 从文件读取字符串, 赋给 str, 最多读取 n-1 个字符
```

7.3.3 二进制文件的读写

- 写二进制文件

```
fwrite(buffer, size, count, pf);
```

将 count 个长度为 size 的连续数据写入到 pf 指向的文件中, buffer 是这些数据的首地址 (可以是指针或数组名)

- 读二进制文件

```
fread(buffer, size, count, pf);
```

从 pf 指向的文件中读取 count 个长度为 size 的连续数据, buffer 是存放这些数据的首地址 (可以是指针或数组名)

例 7.6 C 语言文件读写: 二进制文件. (ex07_fwrite_fread.cpp)

7.3.4 其他文件操作

文件打开后, 默认是从最前面开始读或者写. 有时可能需要从中间某个地方读取数据, 此时需要进行一些位移操作.

- fseek 可以像对待数组那样对待文件

```
fseek(pf, offset, whence);
```

(1) pf 是文件指针, offset 是偏移量 (相对于 whence), 以字节为单位, 长整型 long, 可正可负.

(2) whence 是位置, 取值有

SEEK_SET : 文件开头

SEEK_CUR : 当前位置

SEEK_END : 文件末尾

```
1 fseek(pf, 0L, SEEK_SET); // 定位到文件开头
2 fseek(pf, 10L, SEEK_SET); // 定位到文件第 10 个字节数
3 fseek(pf, -2L, SEEK_CUR); // 从当前位置后移 2 个字节
4 fseek(pf, -10L, SEEK_END); // 定位到文件末尾到时第 10 个字节处
```

- ftell 返回当前位置, 即距离文件开头的字节数, 长整型.

```
long int ftell(FILE * pf);
```

- feof 判断是否到达文件末尾.

```
int feof(FILE * pf);
```

例 7.7 C 语言文件读写: feof (ex07_feof_namelist.cpp)

7.4 上机练习

练习 7.1 **文本文件读写**: 生成一个 6×6 的矩阵 A, 其元素为 $[0,1]$ 之间的随机双精度数, 编写程序实现下面功能:

(a) 将其按矩阵形式写入到一个文本文件 out71.txt 中;

(b) 将其写入到一个二进制文件 data71.dat 中;

(c) 从二进制文件 data71.dat 中读取前 12 个数据 (双精度), 构成一个 2×6 的矩阵 B, 并将 B 按行输出.

(程序取名 hw07_01.cpp)

练习 7.2 **二进制文件读写**: 从课程主页上下载二进制数据文件 data72.dat, 从文件中读取前 60 个元素 (双精度), 构成一个 30×2 的矩阵 A. 然后将其按矩阵形式写入到一个文本文件 out72.txt 中. (程序取名 hw07_02.cpp)

练习 7.3 计算一个正整数的所有数字之和（循环和递归）：编写两个函数，分别用**循环**和**递归**计算一个整数的所有数字之和，并在主函数中分别调用这两个函数计算 2012112118 的所有数字之和.（程序取名 `hw07_03.cpp`）

```
int sum_loop(long x);
int sum_recursion(long x);
```

练习 7.4 多项式乘积运算：编写函数，计算两个多项式 $a(x)$ 和 $b(x)$ 的乘积 $c(x) = a(x) * b(x)$

(a) 多项式用其系数所组成的数组来表示，比如 $a(x) = 2x^3 - 6x + 1$ 对应的数组为 $[2, 0, -6, 1]$；

(b) 函数原型见下面，其中 `pa`, `pb` 分别是指向数组 a 和 b 的指针，这里 a 和 b 是两个数组，分别代表两个多项式 $a(x)$ 和 $b(x)$；`pc` 是指向数组 c 的指针，这里的 c 代表多项式 $c(x)$；`m` 和 `n` 分别是数组 a 和 b 的长度；

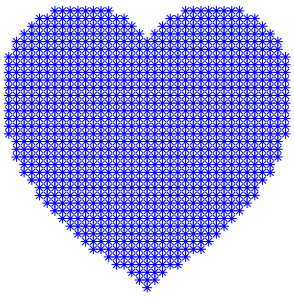
(c) 在主函数中，调用该函数计算下面两个多项式的乘积：$a(x) = 2x^3 - 6x + 1, b(x) = 3x^2 + x - 2$.

（程序取名 `hw07_04.cpp`）

```
void ploy_prod(double *pa, double *pb, double *pc, int m, int n);
```

练习 7.5 心型图案：编写程序，绘制由下面的曲线所表示的心型图案.（程序取名 `hw07_05.cpp`）

$$(x^2 + y^2 - 1)^3 - x^2y^3 = 0, \quad -1.5 < x < 1.5.$$



第八讲 排序算法及其 C++ 实现

本讲主要内容

- 选择排序
- 插入排序
- 希尔排序
- 冒泡排序
- 快速排序
- 归并排序

排序是计算机内经常进行的一种操作, 其目的是将一组“无序”的数据排列成为“有序”的序列. **排序算法**就是如何实现排序的方法, 是计算机科学中非常重要的算法之一, 一个优秀的算法可以节省大量的资源.

8.1 引言

- 排序算法的重要评价指标
  (1) **时间复杂度**（运算量、操作次数）: 设有 n 个数据, 一般来说, 好的排序算法性能是 $O(n\log n)$, 差的性能是 $O(n^2)$, 而理想的性能是 $O(n)$.
  (2) **空间复杂度**: 算法在运行过程中临时占用内存空间的大小.
- **稳定排序算法**: 相等的数据维持原有相对次序.

表 8.1. 常见排序算法的复杂度

算法	平均时间复杂度	最坏时间复杂度	最好时间复杂度	空间复杂度	稳定性
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
希尔排序	$O(n\log^2 n)$	$O(n^2)$	$O(n\log n)$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
快速排序	$O(n\log n)$	$O(n^2)$	$O(n\log n)$	$O(n\log n)$	不稳定
归并排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$	稳定
堆排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$	不稳定
图书馆排序	$O(n\log n)$	$O(n^2)$	$O(n\log n)$	$O(n)$	稳定

基数排序	$O(n \cdot k)$	$O(n \cdot k)$	$O(n \cdot k)$	$O(n+k)$	稳定
桶排序	$O(n+k)$	$O(n^2)$	$O(n)$	$O(n+k)$	稳定
计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	稳定

本讲主要介绍选择排序, 插入排序, 希尔排序, 冒泡排序, 快速排序, 归并排序.

本讲中假定是对数据进行**从小到大**排序.

8.2 选择排序

选择排序也称**最小排序**, 基本思想是: 找出最小值, 将其与第一个位置的元素进行交换, 然后对剩余的序列重复以上过程, 直至排序结束.

例 8.1 选择排序. `(sort_selection/selection100000.cpp/sort_selection.m)`

8.3 插入排序

- **插入排序**基本思想:
  (1) 假设前面 k 个元素已经按顺序排好了, 在排第 $k+1$ 个元素时, 将其插入到前面已排好的 k 个元素中, 使得插入后得到的 $k+1$ 个元素组成的序列仍按值有序;
  (2) 然后采用同样的方法排第 $k+2$ 个元素;
  (3) 以此类推, 直到排完序列的所有元素为止.
- 关键点:
  (1) 如何将第 $k+1$ 个元素插入到前面的有序序列中?
  (2) 策略: 依次与其左边的元素进行比较, 直至遇见第一个不大于它的元素为止.
- 优化: 可以将比较与移位同时进行.

例 8.2 插入排序的 MATLAB 实现 `(sort_insert.m)`

8.4 希尔排序

希尔排序又称为“缩小增量排序”（Diminishing Increment Sort）, 由 D. Shell 于 1959 年提出, 是对插入排序的改进, 主要是基于以下的观察: 插入排序在元素基本有序的情况下, 效率会很高.

- 基本过程
  (1) 把序列按照某个增量（gap）分成几个子序列, 对这几个子序列进行插入排序;
  (2) 不断缩小增量, 扩大每个子序列的元素数量, 并对每个子序列进行插入排序;
  (3) 当增量为 1 时, 子序列就是整个序列, 而此时序列已经基本有序了, 因此只需做少量的比较和移动就可以完成对整个序列的排序.
- 增量（gap）的选取: 初始值可设为 `n/2`, 然后不断减半.

例 8.3 希尔排序的 MATLAB 实现 (sort_shell/shell_comp.m)

8.5 冒泡排序

冒泡排序的基本过程可以描述如下:

(1) 走访需要排序的序列, 比较相邻的两个元素, 如果他们的顺序错误就把他们交换过来.

(2) 不断重复上述过程, 直到没有元素需要交换, 排序结束.

(3) 这个算法的名字由来是因为越大的元素会经由交换慢慢“浮”到数列的顶端.

冒泡排序实现过程

(1) 将第 1 个和第 2 个元素进行比较, 如果前者大于后者, 则交换两者的位置, 否则位置不变;

(2) 然后将第 2 个元素与第 3 个元素进行比较, 如果前者大于后者, 则交换两者的位置, 否则位置不变;

(3) 依此类推, 直到最后两个元素比较完毕为止. 这就是第一轮冒泡过程, 这个过程结束后, 最大的元素就“浮”到了最后一个位置上.

(4) 对前面 $n-1$ 个元素进行第二轮冒泡排序, 结束后, 这 $n-1$ 个元素中的最大值就被安放在了第 n-1 个位置上.

(5) 对前面的 $n-2$ 个元素进行第三轮冒泡排序.

(6) 以此类推, 当执行完第 $n-1$ 轮冒泡过程后, 排序结束.

冒泡排序的优化

如果在某轮冒泡过程中没有发生元素交换, 这说明整个序列已经排好序了, 这时就不用再进行后面的冒泡过程, 可以直接结束程序.

冒泡排序的进一步优化

假设有 100 个数的数组, 仅前面 10 个无序, 后面 90 个都已排好序且都大于前面 10 个数字, 那么在第一轮冒泡过程后, 最后发生交换的位置必定小于 10, 且这个位置之后的数据必定已经有序了, 记录下这位置, 第二次只要从数组头部遍历到这个位置就可以了.

例 8.4 冒泡排序的 MATLAB 实现 (sort_bubble.m)

8.6 快速排序

快速排序是目前最常用的排序算法之一, 它采用的是分而治之思想: 将原问题分解为若干个规模更小但结构与原问题相似的子问题, 然后递归求解这些子问题, 最后将这些子问题的解组合为原问题的解.

- 具体实现过程:

  (1) 随机选定其中一个元素作为基准数（pivot）(通常可采用第一个元素), 然后通过循环和比较运算, 将原序列分割成两部分, 使得新序列中在该基准数前面的元素都小于等于这个元素, 而

其后面的元素都大于等于这个元素. （这时基准数已经归位）

(2) 依此类推, 再对这两个分割好的子序列进行上述过程, 直到排序结束. （递归思想, 分而治之）

- 第一步的具体实现: （假定基准数的值为 a, 原始位置是 $i_1 = 1$）
  (1) 先从原序列的最右边开始, 往左找出第一个小于 a 的数, 然后将该数与基准数交换位置, 设基准数新位置为 i_2;
  (2) 从 i_1 右边的位置开始, 往右找出第一个大于 a 的数, 然后将该数与基准数交换位置, 设基准数新位置为 i_3;
  (3) 从 i_2 左边的位置开始, 往左找出第一个小于 a 的数, 然后将该数与基准数交换位置, 设基准数新位置为 i_4;
  (4) 从 i_3 右边的位置开始, 往右找出第一个大于 a 的数, 然后将该数与基准数交换位置, 设基准数新位置为 i_5;
  (5) 不断重复以上过程, 遍历整个序列.

  ✍ **注记：** 事实上, 可以不用交换, 而是先把基准数保留, 然后直接覆盖即可.

- 后面步骤:
  (1) 对基准数所在位置前面的子序列和后面的子序列, 分别重复第一步的过程
  (2) 不断重复以上过程, 通过递归实现排序.

✍ **注记：** 快速排序还有很多改进版本, 如随机选择基准数, 区间内数据较少时直接用其它的方法排序以减小递归深度等等. 有兴趣的同学可以深入研究.

例 8.5 快速排序的 MATLAB 实现 (sort_quick/quick_main.m)

8.7 归并排序

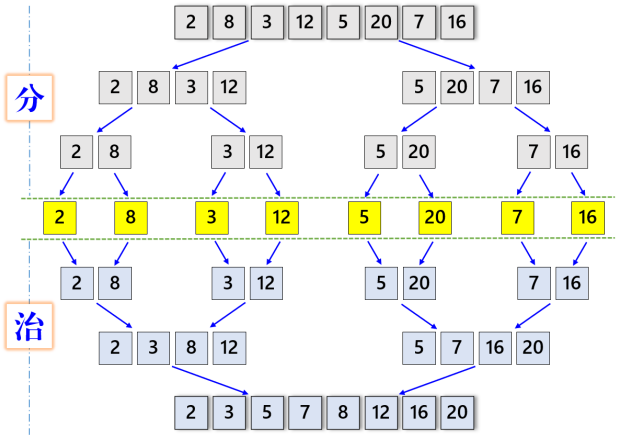
归并排序 是建立在归并操作上的一种排序算法, 是分而治之法的一个典型应用.

8.7.1 算法基本思想

- 假定序列可分成两个子序列, 并假定这两个子序列都已经排好序, 所需做的就是将这两个子序列按大小顺序合并即可 (即归并).
- 子序列的排序可通过递归方式实现.

算法的两基本操作: “分”和“治”

“分”就是将序列不断划分成两个子序列, 直到每个子序列只包含 1 个元素为止, 此时所有子序列都是有序数组, 然后进行归并操作, 即“治”.



8.7.2 算法实现过程

(1) 预先申请一个临时数组, 长度与原序列相等, 在归并时用来临时存放合并后的序列.

(2) 通过递归方式将序列不断划分成两个子序列, 直到每个子序列只包含 1 个元素为止.

(3) 从长度为 1 的子序列开始, 通过递归方式不断 **归并** 两个有序子序列.

两个有序子序列的 **归并** 实现方法:

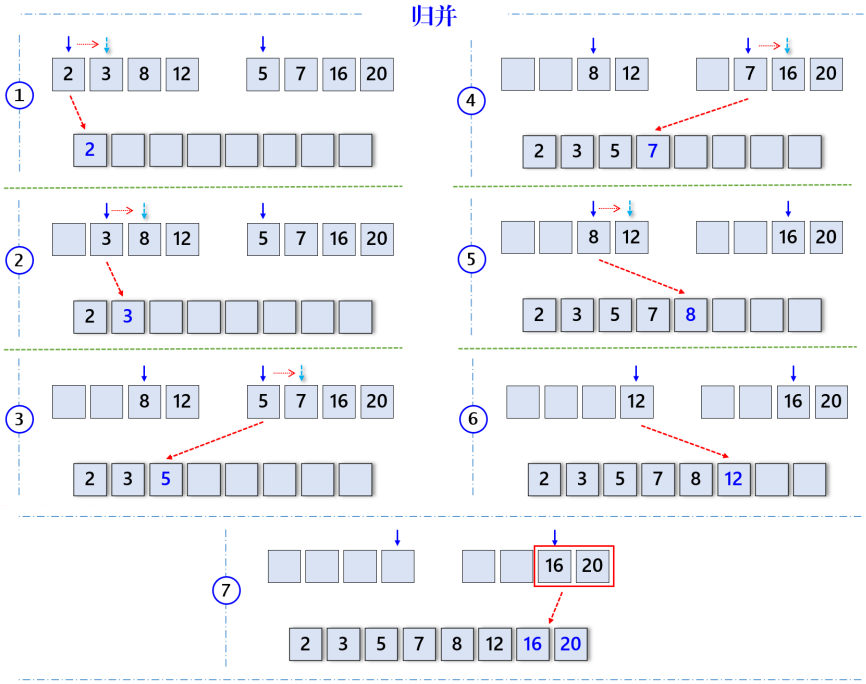
(1) 设定两个指针, 初始值分别为两个子序列起始位置;

(2) 比较两个指针所指向的元素, 将小的元素放入到合并空间 (临时数组) 中, 并移动指针到下一位置;

(3) 重复 (2) 中的步骤，直到某一指针到达序列尾; (见下图中的 ① 至 ⑥)

(4) 将另一序列剩下的所有元素直接复制到合并空间末尾; (见下图中的 ⑦)

(5) 将合并空间 (即合并后的序列) 复制到原数组.



8.8 上机练习

练习 8.1 编写函数, 实现 **插入排序**: 在主函数中生成 15 个小于 100 的随机正整数, 存放在向量 x 中, 然后调用排序函数对 x 进行排序, 并输出排序前后的 x. (程序取名 hw08_01.cpp)

```
void sort_insert(int * px, int n);
```

练习 8.2 将排序方法改为 **希尔排序**. (程序取名 hw08_02.cpp)

```
void sort_shell(int * px, int n);
```

练习 8.3 将排序方法改为优化后的 **冒泡排序**. (程序取名 hw08_03.cpp)

```
void sort_bubble(int * px, int n);
```

练习 8.4 将排序方法改为 **快速排序**. (程序取名 hw08_04.cpp)

```
void sort_quick(int * px, int left, int right);
```

练习 8.5 编写函数实现归并排序. (程序取名 hw08_05.cpp)

```
void sort_merge(int px[], int tmp[], int idx_start, int idx_end);
void merge(int px[], int tmp[], int idx_start, int idx_mid, int idx_end);
```

练习 8.6 **折半插入排序**: 插入排序主要过程是依次将新元素插入到前面已排好序的序列中. 在寻找插入点时, 我们采用的是按顺序依次进行比较. 为了减少比较次数, 我们可以采用折半查找的方法: 设待插入元素为 $a[k]$, 待插入区域为 $[low, high]$, 记 $m = (low + high)/2$, 如果 $a[k] < a[m]$, 则新的待插入区间为 $[low, m-1]$ (即令 $high = m-1$), 否则为 $[m+1, high]$ (即令 $low = m+1$). 依此类推, 直到 $low <= high$ 不成立, 此时 $high + 1$ 就是插入点. 编写函数, 实现折半插入排序, 并在主函数中以 x 为例, 调用排序函数对 x 进行排序, 并输出排序前后的 x. (程序取名 hw08_06.cpp)

```
void sort_insert_binary(int* px, int n);
```

第九讲 类与对象基础 I

本讲主要内容

- 面向对象的基本特点: 抽象、封装、继承和多态
- 类和对象的基本操作
  - ▷ 类的声明
  - ▷ 类的成员: 数据成员与函数成员
  - ▷ 对象的创建 (类的实例化)
  - ▷ 对象成员的访问
  - ▷ 成员函数的定义, 内联成员函数
- 构造函数和析构函数
  - ▷ 构造函数: 数据初始化
  - ▷ 构造函数的重载
  - ▷ 复制构造函数
  - ▷ 对象作为函数参数
  - ▷ 匿名对象
  - ▷ 析构函数

9.1 为什么面向对象

- 出发点: 更直观地描述客观世界中存在的事物 (对象) 以及它们之间的关系.
- 目的: 提高代码的可重用性, 降低软件的开发成本和维护成本, 从而大大提高程序员的生产力.
- 面向对象基本特点:
  (1) 将客观事物看作具有属性 (数据) 和功能 (函数) 的对象;
  (2) 通过抽象找出同一类对象的共同属性和功能 (或行为), 形成类;
  (3) 通过类的继承与多态实现代码重用.
- 面向对象的主要特征: 抽象、封装、继承和多态.
  - 抽象: 对具体问题/事物 (即需要研究的对象) 进行概括, 抽取出这一类对象的公共性质并加以描述的过程.
    (1) 首先关注的是问题的本质及描述, 其次是实现过程或细节;
    (2) 抽象包括数据抽象和功能抽象:
    数据抽象: 描述某类对象的属性或状态 (对象相互区别的物理量) ;
    功能抽象 (功能抽象) : 描述某类对象的共同行为或功能特征.
    (3) 抽象的实现: 类 (`class`)
  - 封装: 将抽象得到的数据和功能相结合, 形成一个有机的整体, 即将数据与操作数据的函数进

行有机结合, 形成“类”, 其中数据和函数都是类的**成员**.

✍ **注记:** 封装可以增强数据的安全性, 并简化编程. 用户不必了解具体的实现细节, 而只需要通过外部接口, 依据给定的访问权限, 来访问类的成员.

例 9.1 时钟的描述

- 数据抽象: `hour`, `minute`, `second`
- 功能抽象: `ShowTime()`, `SetTime()`
- 实现方法: **时钟类**

```
1 class Clock // 时钟类
2 {
3   public:  // 指定成员的访问权限
4     void SetTime(int h, int m, int s);
5     void ShowTime();
6
7   private: // 指定成员的访问权限
8     int hour, minute, second;
9 }; // 此处的分号不能省略!
```

- **继承**: C++ 提供了继承机制, 允许程序员在保持原有类的特性的基础上, 进行更具体、更详细的说明 (即增加新的属性和功能, 或更新原有的属性和功能).
- **多态**: 同一段程序可以作用在不同类型的对象上的能力. 在 C++ 中, 多态是通过强制多态 (如类型转换) 、重载多态 (如函数重载、运算符重载) 、类型参数化和虚函数、模板等方式来实现的.

9.2 类和对象基本操作

类 是 C++ 面向对象程序设计的核心!

- 类与函数的区别:
  (1) 函数是结构化 (过程式) 程序设计的基本模块, 用于完成特定的功能.
  (2) 类是面向对象程序设计的基本模块, 类将逻辑上相关的数据与函数封装, 是对问题的抽象描述.

✍ **注记:** 类的集成程度更高, 更适合大型复杂程序的开发.

- 类的声明: 类必须先声明后使用.

```
class 类名
{
  public: // 公有
    公有成员 (外部接口)
  private: // 私有
    私有成员
```

```
    protected: // 保护
保护型成员 }; // 注意，这里必须要有分号！
```

- 类的成员:
  (1) **数据成员**: 描述事物的属性.
  (2) **函数成员**: 描述事物的行为/功能/操作.
- 成员的访问属性 (访问权限控制)
  (1) `public` (公有属性, 通常也称外部接口) : 任何外部函数都可以访问公有属性的数据和函数.
  (2) `private` (私有属性) : 只能被本类中的函数成员访问, 任何来自外部的访问都非法.
  (3) `protected` (保护属性) : 与私有属性类似, 区别在于继承过程中的影响不同, 将在后面章节中详细解释.

  † 如果没有指定访问属性, 则缺省是 `private`.
  † 一般情况下, 建议将数据成员声明为私有属性或保护属性.
  † 一个类如果没有任何外部接口, 则无法使用.

- 在定义类时, 不同访问属性的成员可以按任意顺序出现, 修饰访问权限的关键字也可以多次出现, 但一个成员只能有一种访问属性.

```
1  // 不同访问属性的成员可以按任意顺序出现
2  class Clock
3  {
4      public:
5          void SetTime(int h, int m, int s);
6      private:
7          int hour, minute, second;
8      public:
9          void ShowTime();
10 };
```

† 一般将公有属性的成员放在最前面, 便于阅读.
† 在类中可以只对成员函数进行声明, 函数的具体实现可以在类外部定义.
† 声明一个类时, 并没有为这个类分配内存, 而只是告诉编译器这个类是什么, 即包含哪些数据, 拥有什么功能.

- 对象的创建: 声明一个类后, 便可将其作为新的数据类型来创建变量, 为了跟普通变量区别开来, 我们通常成由类创建的变量为 **对象变量**, 简称 **对象**.

```
类名 对象名
```

```
1  Clock x; // 创建对象 x
2  Clock y, z; // 创建对象 y, z
```

(1) 类与对象的关系类似于基本数据类型与普通变量之间的关系.

(2) 对象是类的实例, 即具有该类型的变量, 因此创建对象的过程也称为类的 **实例化**.

(3) 对象所占的内存空间只用于存放数据成员.

(4) 函数成员在内存中只占一份空间 (通常存放在内存的代码区), 不会在每个对象中存储副本.

(5) 同一个类的不同对象之间的主要区别: 对象名与数据值.

- 对象成员的访问: "." 操作符

```
对象名.数据成员名
对象名.函数成员名(参数列表)
```

```
1 Clock myclock; // 创建对象 myclock
2
3 myclock.ShowTime(); // 显示时间
4 myclock.SetTime(16,10,28); // 设置时间
```

(1) 类的成员函数可以访问所有数据成员;

(2) 外部函数只能访问公有成员.

- 成员函数的定义

(1) 可以在类内部声明的时候直接定义 (一般适合简短函数) , 如:

```
1  class Clock // 时钟类的声明
2  {
3      public: // 外部接口, 公有成员函数
4          void SetTime(int h, int m, int s)
5          { hour = h; minute = m; second = s; }
6          void ShowTime()
7          { cout << hour << ":" << minute << ":" << second << endl; }
8
9      private:  //私有数据成员
10         int hour, minute, second;
11 };
```

(2) 也可以在类内部声明, 然后在类外部定义 (适用复杂函数) , 如:

```
1  class Clock // 时钟类的声明
2  {
3      public: // 外部接口, 公有成员函数
4          void SetTime(int h, int m, int s);
5          void ShowTime();
6
7      private:  // 私有数据成员
8          int hour, minute, second;
9  };
10
11 void Clock::SetTime(int h, int m, int s) // 在类外部定义成员函数
```

```
{
    hour = h; minute = m; second = s;
}
void Clock::ShowTime() // 在类外部定义成员函数
{
    cout << hour << ":" << minute << ":" << second << endl;
}
```

在类外部定义成员函数时的一般形式是:

```
数据类型 类名::函数名(形参列表) // 注意要加 "类名::"
{
   函数体;
}
```

- 与普通函数的区别: 前面要加上类的名称和两个连续冒号 (作用域分辨符)

> **注记:** 简单成员函数建议直接在类内部定义, 此时函数名前面不需要加“类名::”, 而复杂函数建议在类内部声明, 然后在类外部定义.

- 目标对象/目的对象
  (1) 调用成员函数时, 需用“.”操作符指定本次调用所针对的对象, 该对象就是本次调用的“目标对象”(或“目的对象”);
  (2) 在成员函数中, 可以直接引用目标对象的所有数据成员, 而无需使用“.”操作符.
  (3) 在成员函数中, 引用其它对象的数据成员和函数成员时, 必须使用“.”操作符.
  (4) 在类的成员函数中, 可以直接访问当前类的所有对象的私有成员.
- 成员函数的形参可以带缺省值
  (1) 与普通函数一样, 成员函数的形参也可以带缺省值.
  (2) 成员函数的形参缺省值只能在类中声明或定义时设置, 不能在类外部定义成员函数时设置.

例 9.2 类与对象: 时钟类, 成员函数的形参带缺省值. (ex09_class_Clock_01.cpp)

- 内联成员函数
  (1) 成员函数可以是内联函数, 声明内联成员函数有两种方式: 隐式方式和显式方式
  - 隐式方式: 在类中直接定义的成员函数是内联成员函数.
  - 显式方式: 在类外部定义成员函数时, 使用关键字 inline.
  (2) 使用内联函数可以减少调用开销, 提高效率, 但只适合简单的函数 (短函数) .

```
class Clock
{
    public:
        void ShowTime();
    ... ...
};
```

```
7
8 inline void Clock::ShowTime() // 内联函数
9 { cout << hour << ":" << minute << ":" << second << endl; }
```

9.3 构造函数

构造函数是一类特殊的成员函数, 用于初始化对象.

- 对象的初始化: 创建对象时, 设置数据成员的值.
- 构造函数: 负责对象的初始化, 即创建对象时, 对其数据成员进行初始化.
- 对象创建的过程:
  (1) 申请内存空间, 用于存放 (非静态) 数据成员[1].
  (2) 初始化: 自动调用构造函数初始化数据成员.
- 构造函数的几点说明:
  (1) 构造函数的函数名与类的名称相同.
  (2) 构造函数没有返回值, 前面也不需要带任何返回数据类型.
  (3) 构造函数在对象创建时会被系统自动调用.
  (4) 缺省构造函数: 若用户没有定义构造函数, 则系统会自动生成构造函数, 形参和函数体都为空 (如 `Clock() { }`), 但如果用户自己定义了构造函数, 则系统不再提供缺省构造函数.
  (5) 构造函数可以是内联函数, 形参可以带缺省值, 也可以进行重载.
- 构造函数的重载: 可以通过函数重载定义**多个构造函数**, 系统将根据匹配原则自动选择相应的构造函数来初始化对象.

例 9.3 类与对象: 构造函数 (ex09_class_Clock_02.cpp)

† 在创建对象时, 如果是调用不带形参的构造函数初始化对象, 则不要加小括号;

† 如果构造函数的形参都带有缺省值, 且在初始化对象时都采用缺省值, 则此时也不要加小括号, 如: (ex09_class_Clock_02II.cpp)

```
1  class Clock
2  {
3     public:
4        Clock(int x = 10, int y = 10, int z = 10); // 形参带缺省值
5     ... ...
6  };
7
8  ... ...
9
10 int main()
11 {
12    Clock c1;  // OK
```

[1]静态成员和非静态成员将在后面介绍

```
    Clock c2(); // ERROR, 全部使用缺省值时不要加小括号!
    ... ...
}
```

9.4 复制构造函数

用已有的对象给同一个类的其他对象赋值, 系统会自动调用**复制构造函数**实现复制操作.

- **复制构造函数**: 是一类特殊的构造函数, 其作用是将已有对象的值复制给其它对象.
- 复制构造函数的一般形式:

```
class 类名
{
   public:
      类名(类名 & 对象名); // 复制构造函数的声明, 形参必须是对象的引用!
   ... ...
};

类名::类名(类名 & 对象名) // 复制构造函数的定义
{  函数体; }
```

(1) 复制构造函数的形参必须是对象的引用.
(2) 复制构造函数也可以在类内部直接定义.

- 缺省复制构造函数: 系统自动生成, 将已有对象的数据全部简单复制给指定对象, 即 **浅拷贝**
  (1) 用户可以自己定义复制构造函数, 但无论是否存在自定义的复制构造函数, 缺省复制构造函数总是存在的, 这与构造函数不同.
  (2) 若存在用户自定义的复制构造函数, 则在以下几种情况下会调用自定义复制构造函数:
  - 用一个对象去初始化另一个同类的对象 (注意是初始化, 不是赋值)
  - 若函数的形参是对象, 调用函数时实参与形参的结合 (值传递)

  (3) 无论是否存在自定义的复制构造函数, 用赋值号 "=" 对对象进行赋值时 (不是初始化, 也不是实参与形参的结合), 都是调用缺省复制构造函数, 即自定义复制构造函数不影响赋值号 (初始化除外) 的行为. 若不存在自定义的复制构造函数, 则在所有用对象赋值的地方 (包括初始化和实参与形参的结合) 都是调用缺省赋值构造函数来完成.

> **注记:** 缺省复制构造函数只能实现浅拷贝, 即将已有对象的数据成员的值简单复制给另外一个对象的相应数据成员. 有时这种复制满足不了实际需求. 比如对象 A 中有两个数据成员 x 和 px, 其中 px 是指向 x 的指针, 则将 A 复制给另外一个对象 B 时, B 中的指针 px 所得到的值也是 A 中 x 的地址, 这显然不是我们想要的结果 (事实上, 我们希望 B 中的 px 应该指向 B 中 x, 而不是 A 中 x).

- 对象作为函数参数
  (1) 对象可以作为成员函数和非成员函数的参数;

(2) 与普通变量一样, 对象实参与形参的传递方式有三种: 值传递, 引用传递, 地址传递;
(3) 为了提高程序执行效率, 一般情况下, 对象作为函数参数时, 很少使用值传递.

例 9.4 类与对象: 对象作为函数的参数 (ex09_class_Clock_03.cpp)

例 9.5 类与对象: 复制构造函数 (ex09_class_Point/PointII.cpp)

✍ **注记:** 函数调用时, 只有在进行值传递时, 复制构造函数才会被调用. 若形参是引用或指针, 则不会调用复制构造函数.

9.5 匿名对象

在大多数情况下, 创建对象时都需要指定一个对象名, 但在有些情况下, 可能需要创建一个临时对象, 只使用一次, 这时可以使用匿名对象.

- 匿名对象的声明:

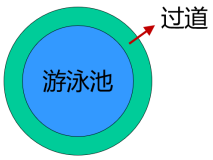
```
类名();          // 调用不带参数的构造函数 (或形参都带缺省值) 初始化匿名对象
类名(参数列表); // 调用带参数的构造函数初始化匿名对象
```

例 9.6 类与对象: 匿名对象 (ex09_class_Clock_04.cpp)

☞ 创建匿名对象时, 若使用不带参数的构造函数或形参都使用缺省值时, 一定要带小括号!

9.6 类与对象举例: 游泳池

例 9.7 一圆形游泳池如图所示, 现在需在其周围建一圆形过道, 并在其四周围上栅栏. 栅栏造价为 35 元/米, 过道造价为 20 元/平方米, 过道宽度 3 米, 游泳池半径由键盘输入. 编程计算并输出过道和栅栏的造价.
(ex09_class_Circle.cpp)



9.7 析构函数

对象在使用完后被释放时, 有时可能需要做一些处理工作, 比如释放由 new 申请的持久动态内存空间等.

- 析构函数: 负责对象被释放时的一些清理工作.
- 析构函数的一般形式:

```
~类名() { 函数体 }
```

(1) 析构函数的函数名由类名前加 "~" 组成.

(2) 析构函数没有返回值, 也不需要加返回值数据类型.
(3) 析构函数在对象生存期即将结束时被自动调用.
(4) 析构函数不接收任何参数.
(5) 若用户没有定义析构函数, 则系统会自动生成一个析构函数 (函数体为空).

9.8 上机练习

练习 9.1 `Rectangle` 类

设计一个名为 `Rectangle` 的类: 表示矩形, 这个类包括:

- 两个 `double` 型数据成员: `width` 和 `height`, 分别表示宽和高;
- 一个不带形参的构造函数: `Rectangle()`, 用于创建缺省矩形: 宽和高都为 1;
- 一个带形参的构造函数: `Rectangle(double x, double y)`, 用于创建指定宽度和高度的矩形;
- 成员函数 `void setwh(double, double)`, 用于更改矩形的宽度和高度;
- 成员函数 `double getw()`, 用于获取矩形的宽度;
- 成员函数 `double geth()`, 用于获取矩形的高度;
- 成员函数 `double getArea()`, 返回矩形的面积;
- 成员函数 `double getPerimeter()`, 返回矩形的周长.

实现这个类, 并在主函数中测试这个类: 创建两个 `Rectangle` 对象: `R1` 和 `R2`, 其中 `R1` 的宽和高分别为 4 和 40, `R2` 的宽和高分别为 3.5 和 35.9, 并在屏幕上输出 `R1` 和 `R2` 的面积和周长. (程序取名 `hw09_01.cpp`)

练习 9.2 `Complex` 类

设计一个名为 `Complex` 的类, 表示一个复数, 这个类包括:

- 两个 `double` 型数据成员: `x` 和 `y`, 分别表示实部和虚部;
- 一个带形参的构造函数: `Complex(double x1, double y1)`, 用于创建指定实部和虚部的复数;
- 成员函数 `double getx()`, 获取实部;
- 成员函数 `double gety()`, 获取虚部;
- 成员函数 `void Display()`, 在屏幕上输出一个复数, 如 `2+3i`, `4-5i`;
- 成员函数 `double Abs()`, 返回一个复数的模;
- 成员函数 `Complex Minus(Complex &)`, 返回当前复数与指定复数之差;
- 成员函数 `Complex Multiply(Complex &)`, 返回当前复数与指定复数的乘积.

实现这个类, 并在主函数中测试这个类: 创建两个复数 `z1 = 1.4 - 2.3i` 和 `z2 = -3.5 + 2.7i`, 并在屏幕上输出 `z1` 的模, `z1 - z2` 和 `z1`×`z2` 的值. (程序取名 `hw09_02.cpp`)

练习 9.3 `Integer` 类

设计一个名为 `Integer` 的类, 表示整数, 这个类包括:

- 一个 `int` 型数据成员: `value`, 分别整数的值;
- 一个带形参的构造函数: `Integer(int x)`, 用给定的整数初始化 `value`;

- 成员函数 `void Display()`, 输出 `value` 的值;
- 成员函数 `int Getvalue()`, 返回 `value` 的值;
- 成员函数 `bool Isprime()`, 判断是否为素数;
- 成员函数 `bool Isequal(int)`, 判断与给定的整数是否相等;
- 成员函数 `bool Isequal(Integer &)`, 判断是否与给定的 `Integer` 对象相等;
- 成员函数 `Integer Add(Integer &)`, 实现两个 `Integer` 对象的加法;
- 非成员函数 `Integer Add(Integer &, int)`, 实现 `Integer` 对象与整数的加法.

实现这个类, 并在主函数中测试这个类:

(1) 创建 `Integer` 对象 `x`, 其 `value` 的值为 2019, 判断其是否为素数;

(2) 要求用户输入一个整数, 然后判断 `x` 是否与这个整数相等;

(3) 创建 `Integer` 对象 `y`, 其 `value` 为 2109, 判断 `x`, `y` 是否相等;

(4) 分别计算 `x` 与 `y` 的和, `x` 与整数 119 的和, 并输出结果.

(程序取名 `hw09_03.cpp`)

第十讲 类与对象基础 II

本讲主要内容

- 类的组合
  - ▷ 什么是组合类
  - ▷ 组合类的初始化
  - ▷ 常量和引用的初始化
  - ▷ 前向引用声明
- 结构体与联合体
- 对象生存期
  - ▷ 局部变量与“全局”变量
  - ▷ 对象的生存期
  - ▷ 静态成员（数据、函数）
- 友元关系 `friend`
  - ▷ 什么是友元关系
  - ▷ 友元函数、友元类

10.1 类的组合

- 类的组合: 将已有的类的对象作为新的类的数据成员, 如:

```
1  class Point // 声明 Point 类
2  {
3     ... ...
4  };
5
6  class Line // 声明 Line 类
7  {
8     public:
9        ... ...
10    private:
11       Point P1, P2; // 内嵌对象成员
12       float S; // 普通数据成员
13 };
```

(1) 在创建类的对象时, 如果这个类的数据成员中包含其它类的对象 (称为内嵌对象成员), 则各个内嵌对象将首先被自动创建.

(2) 组合类初始化包括内嵌对象成员的初始化和普通数据成员初始化.

- 组合类构造函数的一般形式:

```
类名::类名(总参数列表) : 内嵌对象1(参数列表), 内嵌对象2(参数列表), ...
{
    函数体（普通数据成员的初始化）
}
```

(1) ": 内嵌对象 1(参数列表), 内嵌对象 2(参数列表), ..." 称为 "**初始化列表**", 作用是对内嵌对象进行初始化, 这里的参数前面不用加数据类型;
(2) 除了自身的构造函数外, 内嵌对象的构造函数也被调用;
(3) 构造函数调用顺序:
- 按内嵌对象在组合类的声明中出现的顺序依次调用内嵌对象的构造函数;
- 最后调用本类的构造函数.
(4) 总参数列表中的参数需要带数据类型（形参）, 初始化列表则不需要;
(5) 析构函数的调用顺序与构造函数相反.

例 10.1 类与对象: 组合类的初始化. (ex10_class_Line_01.cpp)

✍ **注记:**
如果用不带参数的构造函数 (或者全部使用缺省值) 初始化内嵌对象, 则不用写在初始化列表中.

```
1  class Point
2  {
3    public:
4      Point(float x1 = 0, float y1 = 0) { x = x1; y = y1; } // 构造函数: 形参都
           带缺省值
5      ... ...
6  };
7  class Line
8  {
9    public:
10     Line(float x2, float y2): B(x2, y2) { } // A 采用缺省值, 即 A(0,0)
11     ... ...
12 };
```

✍ 内嵌对象也可以在构造函数体内进行赋值, 但此时要求内嵌对象所属的类存在不带形参的构造函数或者形参全部带缺省值的构造函数.

```
1  class Point
2  {
3    public:
4      Point(float x1 = 0, float y1 = 0) { x = x1; y = y1; } // 构造函数: 形参都
```

```
        带缺省值
    ... ...
};
class Line
{
  public:
    Line(float x1, float y1, float x2, float y2)
    { A = Point(x1, y1); B = Point(x2, y2); } // 在函数体内赋值
    ... ...
};
```

- 数据成员中常量和引用的初始化:

  (1) 常量和引用的特殊性: 不能赋值, 只能初始化（因此必须初始化, 否则无意义）;

  (2) 需要在初始化列表中进行初始化, 如:

例 10.2 类与对象: 常量和引用的初始化. (ex10_class_const_ref.cpp)

✍ **注记:** 不能在类的声明中初始化任何数据成员!

✍ 普通数据成员可以在函数体内赋值（用赋值号）, 也可以在初始化列表中赋值（用小括号）

```
Myclass::Myclass(int x, int y, int z): a(x), b(y), c(z) { } // OK
Myclass::Myclass(int x, int y, int z): a(x), b(y), c = z { } // ERROR
Myclass::Myclass(int x, int y, int z): a(x), b(y) { c(z); } // ERROR
```

- **前向引用声明**: 如果在声明类 A 时需要用到类 B 的对象, 根据类必须先声明后使用的原则, 应该先声明类 B, 但是若声明 B 时也需要用到 A 的对象, 这种情况如何处理?

  (1) 若两个类之间需要互相引用对方的对象, 则需要使用**前向引用声明**, 如:

```
class B; // 前向引用声明
class A // 声明类 A
{
    public:
        void f(B b); // 声明类 A 时需要用到类 B 的对象
};
class B // 声明类 B
{
    public:
        void g(A a); // 声明 B 时也需要用到 A 的对象
};
```

✍ **注记:** 使用前向引用声明时, 只能使用被声明的符号, 而不能涉及类的任何细节

10.2 结构体与联合体

结构体与**联合体** 是两种特殊形态的类, 他们是从 C 语言继承而来, 也是为了保持与 C 语言的兼容性.

- 结构体:

```
struct 结构体名称
{
    public:
        公有成员
    private:
        私有成员
    protected:
        保护成员
};
```

(1) 结构体与类的唯一区别: 在类中, 对于未指定访问控制属性的成员, 默认为私有成员; 而在结构体中, 未指定访问控制属性的成员, 默认为公有成员.

(2) C++ 中的结构体可以包含数据成员和函数成员, 而 C 语言中的结构体只能包含数据成员.

- 联合体: 一种特殊的数据结构, 可以包含多个成员, 但却共用一个存储单元, 如:

```
union Mark
{
    char grade; // 等级制
    bool pass; // 是否通过
    int score; // 百分制
};
```

(1) 联合体的所有成员, 在同一时间至多一个有意义;

(2) 联合体中成员的默认访问属性是公有类型;

(3) 联合体一般只存放数据, 不包含函数成员.

10.3 类作用域

- 类的数据成员与成员函数中的局部变量:

(1) 数据成员可以被类中的所有函数成员访问（类似全局变量）;

(2) 成员函数中声明的变量是局部变量, 只在该函数中有效;

(3) 如果成员函数中声明了与数据成员同名的变量, 则在该函数中数据成员被屏蔽（类似于同名的局部变量和全局变量）.

例 10.3 类作用域: 类的数据成员与成员函数中的局部变量 (ex10_class_local/localII.cpp)

10.4 静态成员

- 生存期: 与普通变量一样, 对象也有动态生存期和静态生存期
  (1) 动态生存期: 对象所在的程序块执行完后, 对象就立即被释放;
  (2) 静态生存期: 生存期与程序的运行期相同, 即在整个程序执行期间一直有效.
- **静态数据成员**:
  (1) 一般情况下, 同一个类的不同对象都有自己的数据成员, 名字一样, 但各有各的值, 互不相干. 但有时希望某些数据成员为所有对象所共有, 这样可以实现数据共享.
  (2) 全局变量可以达到共享数据的目的, 但其安全性得不到保证: 任何函数都可以自由修改全局变量的值, 很有可能偶然失误, 全局变量的值被错误修改, 导致程序的失败. 因此在编程中要慎用全局变量.
  (3) 如果需要在同一个类的多个对象之间实现数据共享, 可以使用 **静态数据成员**.
- 静态数据成员的声明: 与普通变量一样, 声明时在前面加上关键字 `static`
  (1) 静态数据成员在内存中只占一份空间, 为整个类所共有, **不属于任何特定对象**;
  (2) 该类的所有对象共同使用和维护静态数据成员;
  (3) 静态数据成员可以初始化, 但必须在类的外部初始化, 如:

```
1  class Point
2  {
3     ... ...
4     private:
5        static int count; // 静态变量
6  };
7
8  int Point::count = 1; // 静态数据成员的定义和初始化
9  ... ...
```

  (4) 如果静态数据成员没有初始化, 则系统会自动赋予初值 0;
  (5) 定义了静态数据成员, 即使不创建对象, 系统也会为静态数据成员分配空间, 并可以被引用;
  (6) 静态数据成员既可以通过对象名引用, 也可以通过类名来引用, 即:

```
对象名.静态成员名      或      类名::静态成员名
```

例 10.4 静态数据成员举例. (ex10_class_static_01.cpp)

- 静态函数成员
  (1) 用关键字 `static` 修饰, 为整个类所共有, 调用方式: 类名::静态函数成员名
  (2) 静态成员函数没有目标对象, 所以不能对非静态数据成员进行缺省访问, 如:

```
// 假定静态成员函数中有以下语句:
cout << height; // 若 height 是 static, 则合法
cout << width; //若 width 是非静态数据成员, 则不合法
```

(3) 静态成员函数一般用于访问静态数据成员, 维护对象之间共享的数据

(4) 如果静态成员函数访问非静态数据成员时, 需指明对象, 如:

```
cout << p.width; // 访问对象 p 的非静态数据成员 width
```

(5) 静态成员函数声明时需加 `static`, 但定义时不需加 `static`

```
// 静态函数成员
class A
{  public:
     ... ...

     static void fun(A a);
   ... ...
};
```

注记: 实际上也允许通过对象名调用静态成员函数, 但此时使用的是类的性质, 而不是对象本身.

编程好习惯: 只用静态成员函数引用静态数据成员, 而不引用非静态数据成员, 这样思路清晰, 逻辑清楚, 不易出错.

例 10.5 静态函数成员举例. (ex10_class_static_02.cpp)

10.5 友元关系

- **友元关系**: 提供一种类与外部函数之间进行数据共享的机制.
  通俗说法: 一个类主动声明哪些类或外部函数是它的朋友, 从而给它们提供对本类成员的访问特许, 即可以访问私有成员和保护成员.
  (1) 好处: 友元提供了一种数据共享的方式, 提高了程序效率和可读性.
  (2) 坏处: 友元在一定程度上破坏了数据封装和数据隐藏机制.
- 友元包括: **友元函数**与**友元类**
- 友元类的所有函数成员都是友元函数
- **友元函数**
  (1) 用关键字 `friend` 修饰
  (2) 友元函数是非成员函数 (可以是普通函数或其它类的成员函数)
  (3) 友元函数可以通过对象名直接访问私有成员和保护成员

例 10.6 友元函数举例. (ex10_friend_Point.cpp)

- **友元类**
  (1) 用关键字 friend 修饰
  (2) 友元类的所有成员函数都是友元函数

```
class A
{
    public:
        ... ...
        friend class B; // 将 B 声明为友元类
    ... ...
};
```

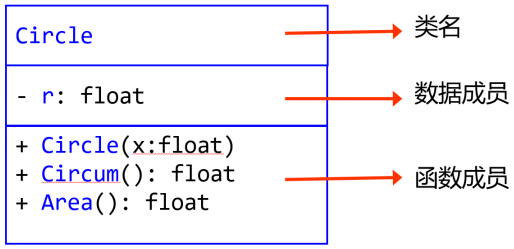
> **注记**：除非确有必要，一般不建议把整个类声明为友元类，而只将确实有需要的成员函数声明为友元函数，这样更安全.

- 关于友元关系的几点说明
  (1) 友元关系**不能传递**
  (2) 友元关系是**单向**的
  (3) 友元关系**不能被继承**（关于类的继承将在后面介绍）

面向对象程序设计的一个基本原则是封装性和信息隐蔽，而友元是对封装原则的一个小的破坏. 但它有助于数据共享，能提高程序效率. 在使用友元时，要注意它的副作用，只有在使用它能使程序精炼，并能大大提高执行效率时才用，否则可能会得不偿失.

10.6 类的 UML 描述 *

- UML: Unified Modeling Language
  (1) 面向对象建模语言，通过图形的方式描述面向对象程序设计
  (2) 在 UML 类图中，类一般由三部分组成：类名，数据成员，函数成员



- 数据成员表示方法:

```
访问属性  名称:类型 [ = 缺省值]
```

访问属性: + 表示 public, - 表示 private, # 表示 protected

- 函数成员表示方法:

```
访问属性  名称 (参数列表) [: 返回类型]
```

10.7 上机练习

练习 10.1 类的组合:

(a) 设计一个名为 Score 的类, 表示成绩, 这个类包括:

- 两个 int 型数据成员: math 和 eng, 分别表示数学成绩和英语成绩;
- 一个带两个形参的构造函数: Score(int x, int y), 用给定的分数初始化 math 和 eng;
- 成员函数 void show(), 输出数学成绩和英语成绩.

(b) 设计一个名为 Student 的类, 表示学生, 这个类包括:

- 两个数据成员: stuid(int 型, 表示学号) 和 mark (Score 对象) ;
- 一个带三个形参的构造函数: Student(int id, int x, int y), 对数据成员进行初始化;
- 成员函数 void stushow(), 输出学号和相应的成绩.

实现这两个类, 并在主函数中测试这个类: 创建一个学生: 学号为 2017007, 数学成绩为 98, 英语成绩为 85, 在屏幕上输出该生的学号和成绩. (程序取名 hw10_01.cpp)

(注: 若没有特别说明, 所有数据成员都是 private, 所有函数成员都是 public)

练习 10.2 设计一个名为 MyDate 的类, 表示日期, 这个类包括:

- 三个 int 型数据成员: year, month, day, 分别表示年、月、日;
- 一个带一个形参的构造函数, 用给出的自 1970 年 1 月 1 日 0 时流逝的秒数创建一个 MyDate 对象, 如果没有给定时间, 则缺省为当前时间:

```
MyDate(unsigned long second=time(0));
```

- 一个带三个形参的构造函数, 用给定的年月日创建一个 MyDate 对象:

```
MyDate(int year, int month, int day);
```

- 成员函数 void showDay(), 在屏幕上输出对象中的年、月、日.

实现这个类, 并在主函数中测试这个类: 创建表示当前时间的 MyDate 对象 d1 和 d2(3456201512), 然后输出它们所对应的日期. (程序取名 hw10_02.cpp)

第十一讲 类与对象基础 III

本讲主要内容

- 常对象与常成员
  - 常对象的声明
  - 常数据成员、常函数成员
  - 常引用
- 对象数组与对象指针
  - 对象数组的声明与初始化
  - 指向对象的指针、this 指针、特殊符号: ->
  - 指向成员的指针
  - 动态对象: new、delete
- 数组类: array
- 向量类: vector
- 字符串类: string

11.1 常对象与常成员

- 常对象: 将对象声明成常对象, 可以有效地保护数据. 常对象的声明如下:

```
const 类名 对象名; // const 可以放在“类名”前面
类名 const 对象名; // 也可以放在后面
```

(1) 常对象必须进行初始化（初始化列表, 不是赋值）;

(2) 常对象主要是针对数据成员, 即常对象的所有数据成员均为常量, 不能被修改.

> ✍ **注记:** 不能通过常对象调用普通成员函数, 即常对象不能作为普通成员函数的目标对象（这里的普通成员函数是相对于后面介绍的常成员函数而言, 即常对象只能作为常成员函数的目标对象）

- 常数据成员

(1) 可以将部分数据成员声明为常量;

(2) 常数据成员必须初始化（采用初始化列表方式, 不是在构造函数的函数体内赋值）, 如:

```
1 // 假定数据成员 x 是常量, y 是普通变量
2 Point::Point(int a, int b): x(a) { y = b; }
```

- 常函数成员

```
类型说明符 函数名(形参) const;
```

(1) 若一个对象是常对象, 则通过该对象只能调用常成员函数;
(2) 普通对象也可以调用常成员函数;
(3) 无论对象是否为常对象, 在常成员函数被调用期间, 目标对象都将被视为常对象.

如果某个成员函数不修改对象的数据, 则可将其声明为常函数.

例 11.1 常对象, 常数据成员, 常成员函数. (ex11_class_const.cpp)

- 常引用

```
const 类型说明符 & 引用名;
```

(1) 常引用可以绑定到常对象, 普通引用不能绑定到常对象;
(2) 常引用也可以绑定到普通对象, 但无论常引用所绑定的是常对象还是普通对象, 都不能通过常引用来修改其所绑定的对象的数据.

例 11.2 常引用. (ex11_ref_const.cpp)

用常引用和常指针作函数参数, 不仅可以节省开销, 还可以保证数据的安全.

11.2 对象数组与对象指针

- 对象数组: 与普通数组类似, 只是数组元素是对象.

(1) 一维对象数组的声明:

```
类名 数组名[n]
```

(2) 一维对象数组的引用:

```
数组名[k].成员名
```

注记: 类似地, 可以声明高维对象数组.

(3) 一维对象数组的初始化: 对每个分量都调用构造函数.

```
1 ... ..
2     Point() { x = 0; y = 0} // 构造函数
3     Point(int a, int b) { x = a; y = b; } // 构造函数
4 ... ...
5
6 int main()
```

```
{
    Point A[2] = {Point(), Point(2,3); // 一维对象数组的初始化
    ... ...
}
```

- 对象指针: 指向对象的指针, 即存放对象的地址.

  (1) 对象指针的声明:

  ```
  类名 * 对象指针名
  ```

  (2) 使用对象指针访问对象成员（两种方式）:

  ```
  对象指针名->成员名 // 对象指针所特有的使用方式, 推荐使用
  (* 对象指针名).成员名
  ```

  (3) 指向常对象的指针:

  ```
  const 类名 * 对象指针名
  ```

- this 指针: 隐含在非静态成员函数中的特殊指针, 永远指向目标对象.

  (1) this 指针是常指针;

  (2) 当局部作用域中声明了与类成员同名的标识符（如变量名）时, 可以通过 this 指针来访问目标对象的成员;

  (3) 当通过一个对象来调用成员函数 (非静态成员函数) 时, 系统会把该对象的起始地址赋给 this 指针.

  ✍ **注记:** 静态成员函数没有 this 指针（静态成员函数没有目标对象!）

例 11.3 类与对象: this 指针 (ex11_this.cpp)

- 指向非静态成员的指针: 直接指向类的成员（数据成员或函数成员）

```
类型说明符 类名::*指针名          // 指向数据成员, 与普通指针的区别: 加类名
类型说明符 (类名::*指针名)(参数) // 指向函数成员
```

(1) 指向数据成员的指针的赋值:

```
指针名=&类名::数据成员名
```

(2) 指向数据成员指针的引用:

```
对象名.*指针名
对象指针名->*指针名
```

(3) 指向函数成员指针的赋值:

```
指针名 = &类名::函数成员名
```

(4) 指向函数成员指针的引用:

```
(对象名.*指针名)(参数)
(对象指针名->*指针名)(参数)
```

- 指向静态成员的指针:
  由于对静态成员的访问不依赖于对象, 因此可以通过普通的指针来访问静态成员, 即声明时不用加类名.

11.3 动态对象

- 动态对象的创建: new

```
类名 *指针名 = new 类名()       // 调用不带参数的构造函数进行初始化
类名 *指针名 = new 类名(参数列表) // 调用带参数的构造函数进行初始化
```

- 动态对象的释放: delete

```
delete 指针名
```

✍ **注记:** 动态对象使用结束后一定要用 delete 手工释放, 否则可能会造成内存泄漏.

11.4 数组类: array

C++ 11 提供了数组类 array, 用于创建固定长度的数组对象, 是 C++ 标准库中的一个模板类, 定义在 array 头文件中. 与普通数组相比, array 具有更好的类型安全和易用性.

- array 的声明: (包含头文件 #include <array>)

```
array<Type, N> 数组名; // N 表示数组长度
```

- Type 可以是原始数据类型 (如 int, float), 也可以是已定义的类 (如 Point).

```
#include <array>
 ... ...
array<int,5> x; // 创建长度为 5 的整型数组对象
array<int,3> y = {1,2,3}; // 创建数组对象并用给定的数据初始化
```

✍ **注记:** 需要注意的是, array 创建的是对象, 不是普通数组.

- 与普通数组类似, 数组对象的元素在内存中也是连续存放的.
- 常用成员函数:

at(int k)	返回下标为 k 的分量, 带边界检查 (即越界会提示)

`size()`	返回数组的长度
`empty()`	判断数组是否为空（长度为 0）
`front()`	返回第一个分量的值
`back()`	返回最后一个分量的值
`fill(const T &)`	将所有分量都设置为指定的值
`swap(array &)`	交换目标数组对象与指定数组对象的值
`data()`	返回指向数组数据的指针
`begin() / end()`	返回数组对象的起始和结束迭代器

- 也可以用下标运算符来访问数组对象的元素:

`v[k]`	下标运算, 返回第 `k` 个分量的值, 不检查边界

- 相同长度的数组对象可以比较运算:

`a1 == a2`	比较运算, 个数和值都相等时返回真, 否则为假
`a1 <=> a2`	C++ 20 新加
`!=, <, <=, >, >=`	比较运算, C++20 中已经移除, 可通过 `<=>` 实现

例 11.4 数组类: 创建数组对象. (ex11_array.cpp)

11.5 向量类: vector

C++ 提供了向量类, 用于创建向量. 向量是对象, 其使用与一维数组类似, 但向量的长度可以根据需要自动增减, 而且提供了更加丰富的成员函数, 使用起来比普通数组更加灵活和方便.

- 向量的声明: (包含头文件 `#include <vector>`)

```
vector<Type> 向量名;
```

(1) 可创建不同数据类型的向量, 由类型说明符 `Type` 指定, 可以是基本数据类型, 如 `int`, `double`, 也可以是自定义的数据类型（包括类）, 即向量的元素也可以是某个类的对象;
(2) 需要指出的是, 这里“向量名”是对象名, 不是数组名;
(3) 由于向量是对象, 所以创建时会直接初始化（调用相应的构造函数）.

```
1 // 创建向量对象
2 #include <vector>
3   ... ...
4 vector<int> x(10); // 创建长度为 10 的整型向量, 注意这里是“( )”, 不是“[ ]”
5 vector<double> y(3); // 创建长度为 3 的双精度型向量
```

- 向量类构造函数（这里只列出部分构造函数, 更多构造函数参见 vector 类的说明文件）

```
vector<Type>();    // 不带形参的构造函数，创建一个空向量，即长度为 0
vector<Type>(int n); //创建一个长度为 n 的向量vector<Type>(int n, const Type
& a); // 长度为 n，并用 a 初始化所有分量
vector<Type>(const vector<Type> & x); // 复制构造函数，用向量初始化新向量
... ...
```

这里的 Type 可以是基本数据类型, 如 int, float, double 等, 也可以是类名, 如 Point, vector.

✍ **注记**：创建向量时, 如果是基本数据类型, 且只指定长度, 则所有分量都会被初始化为 0.

```
1 vector<int> a(100);    // 创建长度为 100 的整型向量，所有分量为 0
2 vector<float> x;       // 创建一个单精度型空向量，即长度为 0
3 vector<float> y(10, 2.1); // 创建长度为 10 的单精度型向量，初值都是 2.1
4 vector<double> z{1.1,2.2,3.3,4.4,5.5}; // 通过初始化列表初始化向量，注意不能用小括号
5 vector<Point> A = {Point(1,2),Point(3,4)}; // 创建 Point 向量并初始化，赋值号可省略
6 vector<vector<float>> X(5,vector<float>(4)); // 创建 5 x 4 的矩阵
```

☕ 如果分量是都是某个类的对象, 则创建向量时可以不指定长度（即长度为 0）, 然后通过成员函数 push_back() 逐步添加元素.

- 向量的基本操作:

v[k]	下标运算, 返回第 k 个分量的值（下标从 0 开始）, 不检查边界
v1 = v2	赋值运算（复制）
v1 == v2	比较运算, 个数和值都相等时返回真, 否则为假
v1 != v2	比较运算, 不相等时返回真, 否则为假
<, <=, >, >=	比较运算, 按字典顺序进行比较

- 常用成员函数:

at(int k)	返回下标为 k 的分量, 带边界检查（即越界会提示）
size()	返回向量的长度
clear()	清空向量中的数据, 长度变为 0
reserve(int n)	预留至少 n 个元素的存储空间
empty()	判断向量是否为空（长度为 0）
front()	返回第一个分量的值
back()	返回最后一个分量的值

`push_back(数据)`	在向量末尾插入一个数据
`pop_back()`	删除最后一个分量
`swap(vector<Type> &)`	交换目的向量与指定向量（形参）的值

✍ **注记：** 下标只能用于获取已存在的元素, 不能通过下标添加元素！

```
vector<int> x;
for(int k = 0; k < 10; k++)
   x[i] = i; // ERROR

// 正确用法
vector<int> x;
for(int k = 0; k < 10; k++)
   x.push_back(i);

// 或者
vector<int> x(10);
for(int k = 0; k < 10; k++)
   x[i] = i;
```

表 11.4. 普通数组, array 和 vector 的比较

	普通数组	`array`	`vector`
大小	编译时固定	编译时固定	动态可变
边界检查	无	`at()` 提供边界检查	`at()` 提供边界检查
内存管理	栈上分配	栈上分配	堆上分配
性能	高效	高效	较低 (动态分配)
接口	不支持 STL 标准接口	支持 STL 标准接口	支持 STL 标准接口

例 11.5 向量类: 创建向量. (ex11_vector.cpp)

例 11.6 向量类: 以其他类的对象为元素. (ex11_vector_point.cpp)

✍ **注记：** 一定要铭记: 向量名不是地址！向量是对象.

例 11.7 向量类: `vector<vector>` 矩阵 (ex11_vector_vector.cpp)

例 11.8 找出给定区间内的所有素数, 存储到一个向量中. (ex11_vector_prime.cpp)

注记

- 向量对象是连续存储的数组 (有时也称为扁平数据结构), 存放在堆中, 会适当预留空间, 如果在向量扩张过程中空间不够, 则会在内存中另外寻找一片更大的连续空间, 并将原向量内容复制过去. 因此, 如果大致知道元素的个数, 可以用 reserve 预分配足够的空间.
- 在 C++ 中, 用来存放数据的称为 **容器**, 向量是常见容器之一, 类似的有 array. 用 array 也是连续存放的, 但存放在栈中, 且长度固定.
- 对于连续存储的数据结构, CPU 缓存可以有效地加载并保持数据的连续块, 大大提高缓存命中率, 减少频繁访问主存带来的延迟, 因此适合频繁遍历或随机访问的场景.
- 其他常见容器有 list, deque, set, map 等.

11.6 字符串类: string

在 C++ 中, 字符串可以通过字符数组来实现, 也可以通过 string 类实现. 在实际使用中, string 类更灵活方便.

✍ **注记**: 为了以示区别, 我们将由字符数组定义的字符串称为 **数组字符串**.

```
#include <string> // 使用 string 类必须包含 string 头文件, 注意不是 cstring
... ...
string str; // 创建一个空字符串对象
string str1 = "Math.", str2("ECNU"); // 创建字符串对象并初始化
string str3 = str1 + str2; // str3 = "Math.ECNU"
string str4(5,'c'); // 创建长度为 5 的字符串, 分量都是字符 'c'
```

- string 类构造函数 (这里只列出部分构造函数, 更多构造函数参见 string 类的说明文件) :

```
string();  // 不带形参的构造函数
string(const string &); // 复制构造函数
string(const char * s); // 用字符串常量进行初始化
string(const string & s, unsigned int p, unsigned int n);
  // 从位置 p 开始, 取 n 个字符, 即 s[p], .., s[p+n-1]
string(const char * s, unsigned int n); // 使用 s 的前 n 个字符进行初始化
string(unsigned int n, char c); // 给定的字符重复 n 次
... ...
```

- 字符串的输入输出:

```
cin >> str;
cout << str;
```

输入也可以用 getline

```
getline(cin, str)  // 以换行符作为输入结束符
getline(cin, str, 'c')//将给定的字符 c 作为输入结束符
```

- 字符串的基本操作

str[k]	下标运算, 返回第 k 个分量
+	连接两个字符串
=	赋值运算（复制）
+=	连接赋值
==, !=, <, <=, >, >=	比较运算

✍ **注记**：比较大小按字典顺序, 从前往后逐个进行比较.

```
1 string str1 = "Hello";
2 string str2(3,'A');
3 string str3 = str1 + str2;
```

- 两个 string 对象可以相加, string 对象和数组字符串也可以相加, 但两个数组字符串不能直接相加.

```
1 string str4 = str1 + " Math"; // OK
2 string str4 = "Hello" + " Math"; // ERROR
3 string str4 = "Hello" + str1 + " Math"; // OK
```

- string 对象可以直接赋值, 但数组字符串不能!

```
1 string str1;
2 char str2[10];
3
4 str1 = "Hello"; // OK
5 str2 = "Hello"; // ERROR
```

- 常用成员函数

str.at(k)	返回第 k 个字符（会自动检测是否越界）
str.length()	字符串对象的长度（字符个数）
str.size()	同上
str.capacity()	返回为当前字符串分配的存储空间

`push_back('c')`	在末尾插入一个字符
`pop_back()`	删除最后一个字符
`str.clear()`	清除字符串中所有内容
`str.erase(k,n)`	从下标 k 开始, 连续清除 n 个字符
`str.empty()`	判断 str 是否为空
`str.~string()`	释放 str

`str.assign(str1)`	用字符串对象赋值
`str.assign(cstr)`	用数组字符串赋值
`str.assign(str1,k,n)`	用 str1 从下标 k 开始的连续 n 个字符赋值
`str.assign(str1,n)`	用 str1 前 n 个字符赋值
`str.assign(n,c)`	用 n 个字符 c 赋值

`str.append(str1)`	将字符串 str1 追加到当前字符串后面
`str.append(str1,k,n)`	追加 str1 从下标 k 开始的连续 n 个字符
`str.append(cstr,n)`	追加数组字符串 cstr 的前 n 个字符
`str.append(n,c)`	追加 n 个字符 c

`str.substr(k,n)`	返回当前字符串从下标 k 开始的连续 n 个字符
`str.substr(k)`	返回当前字符串从下标 k 开始的子串

`str.compare(str1)`	与 str1 比较 (大于为正, 等于为 0, 小于为负)
`str.compare(k,n,str1)`	与 str1 的字串 (从下标 k 开始的连续 n 个字符) 进行比较

`str.insert(k,str1)`	在下标 k 位置插入字符串 str1
`str.insert(k,n,c)`	在下标 k 位置连续插入 n 个字符 c
`str.replace(k,n,str1)`	用 str1 的内容替换从下标 k 开始的 n 个字符

`str.find(str1)`	返回 str1 在当前字符串中首次出现的位置
`str.find(str1,k)`	同上, 从下标 k 位置开始查找
`str.find(c)`	返回字符 c 在当前字符串中首次出现的位置
`str.find(c,k)`	同上, 从下标 k 位置开始查找

`str.c_str()`	将当前字符串转化为数组字符串
`str.data()`	同上

> ✍ **注记：** 更多成员函数参见 C++ Reference

例 11.9 二进制转化为十进制, 用 string 类实现. (ex11_string_bin2dec.cpp)

11.7 上机练习

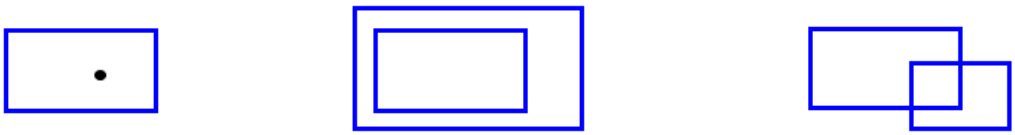
练习 11.1 设计一个名为 Rectangle2D 的类, 表示平面坐标下的一个矩形, 这个类包括:

- 四个 double 型数据成员: x, y, width, height, 分别表示矩形中心坐标、宽和高
- 一个不带形参的构造函数: Rectangle2D(), 用于创建缺省矩形: (x,y) = (0,0), width = height = 1
- 一个带形参的构造函数:

```
Rectangle2D(double x, double y, double width, double height);
```

- 成员函数 double getArea(), 返回矩形面积
- 成员函数 bool contains(double x, double y), 当给定点 (x,y) 在矩形内时返回 true, 否则返回 false, 如下页图 a)
- 成员函数 bool contains(const Rectangle2D & r), 当给定矩形在当前矩形内时返回 true, 否则返回 false, 如下页图 b)
- 成员函数 bool overlaps(const Rectangle2D & r), 当给定矩形与当前矩形有重叠时返回 true, 否则返回 false, 如下页图 c)

实现这个类, 并在主函数中测试这个类: 创建 r1(2,2,5.4,4.8), r2(4,5,10.6,3.3) 和 r3(3,5,2.2,5.5), 输出 r1 的面积, 以及 r1.contains(3,3), r1.contains(r2)和 r1.overlaps(r3) 的结果. (程序取名 hw11_01.cpp)



a) b) c)

练习 11.2 设计 Employee 类, 使用 string 对象, 这个类包括:

- 四个 string 类数据成员: name, addr, city, zip, 分别表示姓名, 街道地址, 省市, 邮编
- 一个带四个形参的构造函数, 用于初始化数据成员

```
Employee(const string & name, const string & addr,
         const string & city, const string & zip)
```

- 成员函数 void ChangeName(string &), 修改姓名
- 成员函数 void Display(), 输出所有信息 (即姓名, 地址, 省市和邮编)
- 数据成员是保护类型的, 函数成员是公有类型的

实现这个类, 并在主函数中测试这个类: 使用你自己的相关信息初始化数据成员, 并在屏幕上输出. (程序取名 hw11_02.cpp)

练习 11.3 字符异位破译, 用 string 类实现: 编写函数, 测试两个字符串是否字符异位相等, 即两个字符串中包含的字母是相同的, 但次序可以不同, 如“silent”和“listen”是字符异位相等, 但“baac”与“abcc”不是. (程序取名 hw11_03.cpp)
(提示: 先对字符串进行排序, 然后再比较)

```
bool isAnagram(const string & str1, const string & str2);
void sort(string & str);
```

练习 11.4 编写函数, 将一个十进制数转化为(用 string 对象表示)二进制数, 并在主函数中测试该函数: 将 2019 转化为二进制并在屏幕上输出. (程序取名 hw11_04.cpp)

```
string dec2bin(const long n);
```

练习 11.5 用 vector 实现多项式类. (程序取名 hw11_05.cpp)

练习 11.6 编写函数, 用 vector 类实现矩阵的求和与乘积计算. (程序取名 hw11_06.cpp)

练习 11.7 编写函数, 用 string 类实现字符串匹配的 KMP 算法. (程序取名 hw11_07.cpp)

第十二讲 运算符重载与自动类型转换

本讲主要内容

- 运算符重载
  - ▷ 为什么要“运算符重载”
  - ▷ 哪些运算符可以重载
  - ▷ 如何实现运算符重载
  - ▷ 实现方式: 成员函数与非成员函数
  - ▷ 重载 [] 与左值
- 自动类型转换
  - ▷ 为什么要“自动类型转换”
  - ▷ 怎么实现自动类型转换
  - ▷ 实现方式: 成员函数与非成员函数
  - ▷ 注意事项

12.1 为什么要重载运算符

预定义的运算符 (如 +、-、*、/、>、<、==、<<、>> 等) 只针对基本数据类型, 若要使得对象也能进行类似的运算, 则需要通过运算符重载来实现.

- 运算符重载:
  本质上就是函数重载, 即对已有的运算符添加多重含义, 使得同一个运算符可以作用于不同类型的数据, 特别是类的对象. 比如声明了 Complex 类后, 怎样实现 Complex 对象的加法?

```
1  int x1=1, x2=2, x3;
2  x3 = x1 + x2; // 普通数据类型的加法
3
4  Complex z1(1.2,3.4), z2(5.6,7.8), z3;
5  z3 = z1 + z2; // Complex 类对象的加法, 如何实现? --> 运算符重载
```

- 运算符重载基本规则:
  (1) 只能重载已有的运算符;
  (2) 重载不改变运算符的优先级和结合率;
  (3) 运算符重载不改变运算符的操作数的个数;
  (4) 重载的运算符的功能应该与已有的功能类似;
  (5) 运算符重载是为了满足新数据类型 (类与对象) 的需要, 因此要求至少有一个操作数是新类型的数据 (这里可以理解为自定义类的对象)

- 四个不能被重载的运算符:

```
.     .*     ::     ?:
```

12.2 怎么实现运算符重载

- 运算符重载的一般形式: (以成员函数方式为例)

(1) 声明:

```
类型说明符 operator运算符(形参列表);
```

(2) 定义: 可以在声明时直接定义, 也可以在类内部只声明, 然后在类外部定义, 此时需加上类名, 即

```
类型说明符 类名::operator运算符(形参列表) { 函数体; }
```

```
1 // 重载加法运算, 使得 Complex 类对象也能相加
2 Complex Complex::operator+(Complex & c2)
3 {
4     return Complex(real+c2.real, imag+c2.imag);
5 }
```

† 这里的类型说明符可以是类名或基本数据类型.

- 运算符重载的两种实现方式: 成员函数和非成员函数 (即外部函数) .

12.3 运算符重载: 成员函数方式

- 特点:

(1) 可以自由访问本类对象的 (私有) 数据成员;
(2) 运算符重载为成员函数时, 形参个数可以少一个;
(3) 若是双目运算, 则左操作数就是目标对象本身, 可使用 this 指针来调用;
(4) 若是单目运算, 则目标对象就是操作数, 不需要其它形参 (注: 后置 ++ 和后置 -- 除外)

例 12.1 运算符重载: 有理数加法运算. (ex12_overload_member_01.cpp)

- 双目运算符的重载 (成员函数方式) :

```
类型说明符 operator⊙(const 类名 & );
```

(1) 形参建议用“常引用”, 既可以实现与变量的运算, 也可以实现与常量的运算.
(2) 这里 ⊙ 表示可以重载的双目运算.
(3) 注意: 只有一个形参.

- 前置单目运算符的重载 (成员函数方式) :

```
类型说明符 operator⊙();
```

(1) 常见的前置单目运算符有: `!`, `-`, `++`, `--`

(2) 注意: 没有形参.

- 后置单目运算符 (`++`、`--`) 的重载 (成员函数方式) :

```
类型说明符 operator⊙(int);
```

(1) 带一个整型形参, 但该参数在运算中不起任何作用, 只用于区分前置和后置, 也称为 **伪参数**.

例 12.2 运算符重载: 前置和后置单目运算. (ex12_overload_member_02.cpp)

12.4 运算符重载: 非成员函数方式

- 非成员函数 (外部函数方式) :

(1) 形参个数与操作数相同;

(2) 所有操作数都通过参数传递;

(3) 一般需在相关类中将其声明为友元函数, 以便可以直接访问 (私有) 数据成员.

```
1  class Complex
2  {  ... ...
3     public:
4        friend Complex operator+(const Complex &, const Complex &);
5     ... ...
6  }
7
8  Complex operator+(const Complex & c1, const Complex & c2)
9  {
10    return complex(c1.real + c2.real, c1.imag + c2.imag);
11 }
```

- 双目运算符的重载 (非成员函数方式) :

(1) 在类内部声明为友函数:

```
friend 类型说明符 operator⊙(const 类名 &, const 类名 &);
```

(2) 定义 (必须在类外部定义) :

```
类型说明符 operator⊙(const 类名 & p1, const 类名 & p2) { 函数体; }
```

† 注意: 不是成员函数, 所以定义时不要加类名.

例 12.3 运算符重载: 有理数减法运算, 非成员函数方式. (ex12_overload_nonmember_01.cpp)

- 前置单目运算符的重载 (非成员函数方式) :

```
friend 类型说明符 operator⊙(类名 &);
```

- 后置单目运算符 (如: ++、--) 的重载 (非成员函数方式) :

```
friend 类型说明符 operator⊙(类名 &, int);
```

▷ 第二个形参是伪参数, 只用于区分前置和后置.

例 12.4 运算符重载: 前置和后置单目运算, 非成员函数方式.
(ex12_overload_nonmember_01.cpp)

成员函数 or 非成员函数

- 运算符 []、=、->、() 必须以成员函数方式重载.
  ▷ 运算符 () 通常表示强制类型转换, 如 double(x).

  ✍ **注记**: C++ 并不要求 ++ 和 -- 必须是类的成员, 但由于这些运算会改变操作数的状态, 因此建议以成员函数方式重载.

  ✍ 通常情况下, 复合运算符 (如 +=, -= 等) 也建议以成员函数方式重载.

- 运算符 <<、>> 必须以非成员函数重载
  (这两个运算符的重载涉及到输入输出流, 将在文件流中介绍)
- 其他运算符既可以用成员函数方式重载, 也可以用非成员函数重载. 建议如下:
  ▷ 单目运算符, 建议用成员函数方式重载;
  ▷ 双目运算符, 如果两个操作数的地位是平等的, 且不改变操作数的值, 则建议用非成员函数方式;
  ▷ 双目运算符, 如果两个操作数的地位不平等, 则建议用成员函数方式, 如 += 等.

☕ 算术运算符和关系运算符建议以 **非成员函数方式** 重载, 以便实现一些简单的自动类型转换.

12.5 重载赋值运算 =

- 为什么要重载 =
  用户可以自定义复制构造函数, 但赋值运算 "=" 仍然调用缺省复制构造函数。如果要修改赋值运算的工作方式, 则需重载赋值号.

```
1 class Rational
2 {
3   public:
4     ... ...
5     Rational & operator=(const Rational &p); // 重载赋值运算符
6     ... ...
7 };
```

▷ 注意: 通常返回的是对象的引用, 不是 void, 因为 C++ 允许多重赋值: a = b = c.

例 12.5 运算符 = 的重载. (ex12_overload_assignment.cpp)

12.6 左值与运算符 [] 的重载

- 为什么要重载 []
  在数组运算中, 可以通过 [] 来引用指定位置的元素. 现在对于 Rational 类, 我们希望用 r[0] 表示分子, r[1] 表示分母, 怎么实现? 我们可以这么做:

```
1 // 成员函数方式重载
2 int Rational::operator[](int idx)
3 {
4     if (idx == 0)
5         return x;
6     else
7         return y;
8 }
```

  这样, 我们就可以用 r[0] 代表分子, r[1] 代表分母了, 如:

```
1 Rational a(4,5);
2 cout << a[0] << "/" << a[1] << endl;
```

  但是, 如果我们还希望用 r[0] 和 r[1] 对分子和分母赋值呢? 比如 r[0]=3, 上面的实现方式显然是不行的. 这个问题涉及到**左值**.
- **左值**: 能出现在赋值号左边的量称为左值. 比如普通变量是左值, 常量不是左值.
- 怎样才能使得 r[0] 能出现在赋值号左边? → 返回 r[0] 的引用.

```
1 int & Rational::operator[](int idx) // 函数返回的是一个引用
2 {
3     if (idx == 0)
4         return x;
5     else
6         return y;
7 }
```

- 所有返回值需要出现在赋值号左边的重载, 返回的必须是一个引用 (即以左值方式返回) , 如复合赋值运算符 +=, -=, 等等.

> **注记:** 为了与内置版本保持一致, 前置递增或递减通常返回递增或递减后的对象的引用, 而后置运算则返回对象的原值, 不是引用.

例 12.6 运算符 [] 的重载. (ex12_overload_left_value.cpp)

12.7 自动类型转换

- 为什么要自动类型转换? → 实现对象与基本数据类型变量之间的运算.
- 怎么实现? → 将对象自动转换为基本数据类型, 或者将基本数据类型自动转换为对象.
- 基本数据类型 → 对象

```
Rational a(1,2), b;
int c = 3;
b = a + c; // 怎么实现? --> 将整数转换为有理数, 然后参与运算.
```

(1) 实现方法: 构造函数, 例如有理数与整数的加法运算.

例 12.7 自动类型转换: 基本数据类型 → 对象. (ex12_overload_conversion_01.cpp)

思考: 上例中要把 c 放在加号的右边, 怎么实现它也能出现在左边, 即 b=c+a?

- 对象 → 基本数据类型

```
Rational a(1,2);
double b=0.8, c;
c = a + b; // 怎么实现? --> 将有理数转换为双精度数, 然后参与运算.
```

(1) 实现方法: 重载类型转换函数 (必须以成员函数方式) .

(2) 声明:

```
operator 转换函数名();
```

(3) 定义 (假定在类外部定义) :

```
类名::operator 转换函数名() { 函数体 };
```

† 注意: 没有返回数据类型, 因为**转换函数名**就指定了返回数据类型.

例 12.8 自动类型转换: 对象 → 基本数据类型. (ex12_overload_conversion_02.cpp)

- 自动类型转换的注意点:

(1) 一个类可以重载类型转换函数, 实现对象到基本数据类型的转换, 也可以重载构造函数, 实现基本数据类型到对象的转化, 但两者不能并存!

(2) 若重载了类型转换函数, 则建议用非成员函数方式实现运算符重载, 并且形参使用“常引用”. 如:

```
Rational operator+(const Rational & r1, const Rational & r2)
```

12.8 上机练习

练习 12.1 设计名为 Complex 的类, 表示一个复数, 这个类包括:

- 两个私有型 `float` 数据成员: `x, y`, 分别表示实部和虚部;
- 一个不带形参的构造函数 `Complex()`, 设置缺省值 `x=y=0`;
- 一个带形参的构造函数 `Complex(float x, float y)`, 初始化实部和虚部;
- 成员函数 `void Display()`, 输出复数, 如 `2+3i`, `4-5i`;
- 以成员函数方式重载复合赋值运算 "+=";
- 以非成员函数方式重载复数的除法运算 "/".

实现这个类, 并在主函数中测试: 创建两个复数 `z1 = 2.1-5.4i` 和 `z2 = 7.5+3.2i`, 计算 `z3 = z2/z1` 和 `z3 += z1`, 并在屏幕上输出. (程序取名 `hw12_01.cpp`)

练习 12.2 以非成员函数方式重载复数的加法运算, 使之能执行下面的运算:

```
Complex a(2.1,5.7), b(7.5,8), c, d;
c = a + b;
d = b + 5.6;
e = 4.1 + a;
```

(程序取名 `hw12_02.cpp`)

练习 12.3 以成员函数方式重载有理数的比较运算, 即 `>, ==, <`, 使之能执行下面的运算:

```
Rational a(4,5), b(2,3);
cout << "a>b? " << (a>b ? "true" : "false") << endl;
cout << "a==b? " << (a==b ? "true" : "false") << endl;
cout << "a<b? " << (a<b ? "true" : "false") << endl;
```

(程序取名 `hw12_03.cpp`)

练习 12.4 设计名为 `PolyInt` 的类, 表示整系数多项式 (即系数为整数) , 这个类包括:

- 一个 `int` 型数据成员: `int n`, 表示多项式的次数;
- 一个 `int` 型指针数据成员: `int * a`, 表示多项式的系数, 即

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n,$$

  其中 `a` 指向系数向量 $[a_0, a_1, \ldots, a_n]$ 的首地址;
- 不带形参的构造函数: `PolyInt()`, 用于创建多项式 $p(x) = 0$, 即 `n = 0, a[0] = 0`;
  (提示: 在构造函数中用 `new` 为对象申请存储空间)
- 带一个形参的构造函数: `PolyInt(int n)`, 用于创建系数全部为 `0` 的 `n` 次多项式;
- 带两个形参的构造函数: `PolyInt(int n, int a[])`;
- 成员函数 `void Display()`, 按数组方式输出多项式,
  如 $p(x) = 1 + 2x + 4x^3$ 输出为 `[1,2,0,4]`;
- 以成员函数方式重载赋值 "`=`" 运算:
  `PolyInt & operator=(const PolyInt & p)`
- 以非成员函数方式重载 "+", 实现两个整系数多项式的加法运算:
  `PolyInt operator+(const PolyInt & p, const PolyInt & q)`
- 析构函数: `~PolyInt()`, 释放由 `new` 申请的存储空间.

实现这个类, 并在主函数中测试这个类: 创建整系数多项式 $p(x) = 1 + 2x + 4x^3$ 和 $q(x) = 2 + 4x^2 + 3x^3 + 5x^5$, 计算 $s(x) = p(x) + q(x)$, 并在屏幕上输出 $p(x), q(x)$ 和 $s(x)$.
(程序取名 hw12_04.cpp)

练习 12.5 改用向量类实现上面的 PolyInt, 即数据成员 a 是 vector 对象. (程序取名 hw12_05.cpp)

第十三讲 继承与派生

本讲主要内容

- 什么是类的继承/派生
- 怎么定义派生类
- 如何继承基类的成员: 继承方式与继承过程
- 派生类成员: 从基类继承的成员 + 新增成员
- 派生类成员的访问属性
- 派生类的构造函数 (初始化列表, 数据成员初始化顺序)
- 派生类成员的标识同名问题: 作用域分辨符 "::"
- 类型兼容规则: 派生类对象/基类对象
- 多重继承时重复继承问题 — 虚继承/虚基类

继承是面向对象技术的一个重要概念. 如果一个类 B 继承自另一个类 A, 就把这个 B 称为 A 的**派生类**或**子类**, 而把 A 称为 B 的**基类**或**父类**. 继承可以使得派生类具有基类的各种属性和功能, 而不需要再次编写相同的代码. 派生类在继承基类的同时, 还可以通过重新定义某些属性或改写某些方法来更新基类的原有属性和功能, 或增加新的属性和功能.

13.1 继承与派生

- 类的派生: 在已有类的基础上产生新的类的过程.
- 类的继承: 派生类继承了基类的特性, 包括数据成员 (属性) 和函数成员 (功能).
- 原有类称为**基类**或**父类**, 新类则称为**派生类**或**子类**.
- 派生类可以改造基类的特性, 也可以加入新的特性.
- 派生类也可以作为父类, 派生出新的子类. 如果 A 是 B 的基类, 而 B 是 C 的基类, 则称 B 是 C 的 **直接基类**, 而 A 是 C 的 **间接基类**.

✍ **注记:** 派生和继承提高了代码的可重用性, 有利于软件开发.

13.2 派生类的定义

```
class 派生类名 : 继承方式 基类1, 继承方式 基类2, ...
{
  派生类成员声明;
};
```

- 关于派生的几点说明:
  (1) 一个派生类可以有多个基类, 此时称为 **多重继承**;
  (2) 如果一个派生类只有一个基类, 则称为 **单继承**;
  (3) 一个父类可以派生出多个子类, 从而形成 **类族**.
- 继承方式:
  (1) 继承方式: 控制从基类继承的成员的访问属性.
  (2) 继承方式有三种: `public`、`protected`、`private`, 缺省为 `private`.
  (3) 派生类成员: 从基类继承的成员 + 新增加的成员.
  (4) 继承是可传递的: 从基类继承的属性和功能可以传递给新派生的子类.
- 类的派生过程:
  (1) 吸收基类成员: 派生类包含基类中除构造函数和析构函数外的所有非静态成员;
  (2) 改造基类成员:
    - 通过继承方式改变基类成员的访问属性;
    - 对基类成员的屏蔽: 如果派生类中新增的成员 (数据或函数) 与基类成员同名, 则缺省只能访问新增的成员 (成员函数只要函数名相同, 即使形参不一样, 也都会被屏蔽);
  (3) 添加新成员: 根据实际需要, 添加新的数据成员或函数成员.

> ✍ **注记:** 构造函数, 析构函数和静态成员不能被继承.

- 派生类从基类继承的成员的访问属性
  (1) 这里主要指派生类中新增成员和外部函数能否访问派生类中从基类继承的成员.
  (2) 继承方式不同, 访问属性不同:
  公有继承 (`public`)
    - 基类的公有和保护成员的访问属性保持不变;
    - 基类的私有成员不可直接访问.

  私有继承 (`private`)
    - 基类的公有和保护成员都成为派生类的私有成员;
    - 基类的私有成员不可直接访问.

  保护继承 (`protected`)
    - 基类的公有和保护成员都成为派生类的保护成员;
    - 基类的私有成员不可直接访问.

例 13.1 派生类: 公有继承 (`ex13_Inheritance_Point01.cpp`)

例 13.2 派生类: 保护继承 (`ex13_Inheritance_Point02.cpp`)

- 几点注记:
  (1) 从基类继承的成员函数对基类成员的访问不受影响.
  (2) 无论以何种方式继承, 基类的私有成员都不可直接访问.

(3) 私有继承后, 基类成员 (特别是公有函数) 无法在以后的派生类中直接发挥作用, 相当于终止了基类功能的继续派生. 因此, 私有继承较少使用.

(4) 保护继承与私有继承的区别: 基类成员 (特别是公有函数) 可以在以后的派生中作为保护成员继承下去.

访问控制小结

(1) 基类成员函数访问基类成员: 正常访问.
(2) 派生类成员函数访问派生类新增成员: 正常访问.
(3) 基类成员函数访问派生类新增成员: 不能访问.
(4) 派生类成员函数访问基类成员: 取决于继承方式和基类成员本身访问属性.
(5) 派生类外部函数 (非成员函数) 访问派生类所有成员: 只能访问公有成员.

- 派生类成员按访问属性可划分为下面四类:
  - 不可访问成员: 基类的私有成员.
  - 私有成员: 基类继承的部分成员 + 新增的私有成员.
  - 保护成员: 基类继承的部分成员 + 新增的保护成员.
  - 公有成员: 基类继承的部分成员 + 新增的公有成员.

例 13.3 派生类成员访问控制 (ex13_Inheritance_Person01.cpp)

13.3 派生类构造函数

派生类不能继承基类的构造函数, 因此在定义派生类的构造函数时, 需要调用基类的构造函数来初始化从基类继承的数据成员.

- 派生类对象的初始化: 包括新增数据成员的初始化和继承的数据成员的初始化.
  - 从基类继承的数据成员需通过调用基类的构造函数进行初始化;
  - 派生类的构造函数 (函数体) 负责新增数据成员 (基本数据类型, 不含内嵌对象) 的初始化.
- 派生类构造函数 (初始化参数列表)

```
派生类名(总参数列表) : 基类1(参数), ..., 基类n(参数),
                        内嵌对象1(参数), ..., 内嵌对象m(参数)
{
  新增基本数据成员的初始化 (不包括继承的基类成员和内嵌对象);
};
```

(1) 总参数列表中的参数需要带数据类型 (形参), 其他参数 (基类和内嵌对象) 不需要;
(2) 需要初始化的数据成员: 基类成员 + 内嵌对象 + 新增数据成员 (基本数据类型);
(3) 基类数据成员的初始化: 调用基类构造函数;
(4) 内嵌对象的初始化: 调用内嵌对象所在类的构造函数.

- 几点注记

- ▷ 在派生类构造函数的总参数列表中, 有一些参数是传递给基类和内嵌对象的构造函数的.
- ▷ 若基类使用不带参数的构造函数进行初始化, 则可以省略.
- ▷ 若内嵌对象使用不带参数的构造函数来初始化, 也可以省略.
- ▷ 派生类构造函数的执行顺序:
  (1) 调用基类的构造函数, 按被继承时声明的顺序执行;
  (2) 对派生类新增内嵌对象初始化, 按它们在类中声明的顺序执行;
  (3) 执行派生类构造函数体的内容.

例 13.4 派生类: 构造函数. (ex13_Inheritance_BC01.cpp)

思考: 如果 Student 中的成员函数 showStu 也取名为 show, 则该如何处理?

- 派生类构造函数若只是声明, 则不需要带初始化列表, 定义时要带初始化列表.

```
class C: public B2, public B1, public B3
{
  public:
    C(int a, int b, int c, int d); // 只是声明, 不能带初始化列表
    ... ...
};
C::C(int a, int b, int c, int d) : B1(a), memberB2(d), memberB1(c), B2(b)
{ x = a; }
```

13.4 派生类的复制构造函数

- 派生类复制构造函数的作用:
  调用基类的复制构造函数完成基类数据成员的复制, 然后再执行派生类数据成员的复制.
- 在定义派生类的复制构造函数时, 需要为基类相应的复制构造函数传递参数.

13.5 派生类的析构函数

- 派生类的析构函数只负责新增数据成员 (非对象成员) 的清理工作.
- 派生类析构函数的定义与普通析构函数一样.
- 基类成员和新增对象成员的清理工作由基类和对象成员所属类的析构函数负责.
- 析构函数的执行顺序与构造函数相反:
  (1) 执行派生类析构函数体;
  (2) 执行派生类对象成员的析构函数;
  (3) 执行基类的析构函数.

13.6 同名成员屏蔽规则

如果派生类中出现与基类同名的成员, 该如何处理?

同名成员屏蔽规则

如果存在两个或多个具有包含关系的作用域, 则外层作用域声明的标识符在内层作用域可见; 但如果在内层作用域声明了同名标识符, 则外层标识符在内层不可见.

- 派生类与基类
  (1) 类其实也是一种作用域, 在这个作用域内定义类的成员. 当存在继承关系时, 派生类的作用域嵌套在基类的作用域内, 即基类是外层, 派生类是内层.
  (2) 若在派生类中新定义了与基类同名的成员, 则缺省使用新定义的成员.
  (3) 若在派生类中声明了与基类同名的新函数, 即使函数参数表不同, 则从基类继承的同名函数的所有重载形式都会被屏蔽, **这意味着基类成员函数与派生类成员函数不会构成函数重载**.
  (4) 如何访问被屏蔽的成员: 类名 + 作用域分辨符 ::

```
类名::成员名          // 数据成员
类名::成员名(参数)    // 函数成员
```

  (5) 若派生类有多个基类, 且这些基类中有同名标识符, 则必须使用作用域分辨符来指定使用哪个基类的标识符.
  (6) 利用作用域分辨符可明确标识派生类中从基类继承的成员, 从而解决了成员同名问题.

例 13.5 类的派生: 屏蔽规则举例 (ex13_Person_02.cpp)

13.7 类型兼容规则

- 基本规则
  (1) 在需要基类对象出现的地方, 可以使用派生类 (以公有方式继承) 的对象来替代.
  (2) 通俗解释: 公有派生类实际具备了基类的所有功能, 凡是基类能解决的问题, 公有派生类都可以解决.
  (3) 替代后, 只能使用从基类继承的成员, 即派生类只能发挥基类的作用.
- 类型兼容规则中的替代包括以下情况:
  (1) 派生类的对象可以隐式转化为基类对象;
  (2) 派生类的对象可以初始化基类的引用;
  (3) 派生类的指针可以隐式转化为基类的指针.

13.8 虚继承

为什么虚继承

在多重继承时, 如果派生类的部分或全部基类是从另一个共同基类派生而来, 则在最终的派生类中会保留该间接共同基类成员的多份同名成员. 这时不仅会存在标识符同名问题, 还会占用额外的存储空间, 同时也增加了访问这些成员时的困难, 且容易出错. 事实上, 在很多情况下, 我们只需要一个这样的成员副本 (特别是函数成员) .

怎么实现虚继承

当某个类的部分或全部基类是从另一个共同基类派生而来时, 可以将继承方式设置成虚继承, 这时从不同路径继承来的共同基类的数据成员在内存中只存放一个副本, 同一个函数名也只有一个映射, 有效避免了多重副本问题.

- 虚继承的声明: 派生时在继承方式前加关键字 `virtual`

```
class 派生类名: virtual 继承方式 基类名
{
   ... ...
};
```

- 一个类可以在派生某个类时采用虚继承方式, 而在派生另一个类时不采用虚继承方式.
- 为了保证基类成员在派生类中只继承一次, 可以在该基类派生其他类时都采用虚继承方式, 否则仍然可能会出现对该基类的多重继承.
- 采用虚继承时派生类的构造函数:
  - ▷ 如果基类的构造函数带有参数, 则在**所有直接或间接继承该基类的派生类**中, 都必须在构造函数的初始化列表中包含**该基类构造函数的直接调用**;
  - ▷ 如果基类的构造函数不带参数, 则可以省略.

例 13.6 虚继承举例 (ex13_Inheritance_virtual.cpp)

✍ **注记:** 在初始化 Graduate 类的对象时, 只会通过 Graduate 的构造函数中的 `Person(str,a)` 来初始化从 Person 类继承的数据成员, `Teacher(str,a,tit)` 和 `Student(str,a,sco)` 中对 `Person` 构造函数的直接调用会被忽略.

```
1 // 如果将 Graduate 的构造函数改为下面的语句, 程序运行结果不变
2 Graduate(const string & str, int a, const string & tit, float sco, float w)
3    : Person(str,a), Teacher("Tea",2,tit), Student("Stu",3,sco), wage(w){ }
```

✍ **注记:** 对于普通继承, 派生类构造函数只能调用直接基类的构造函数, 不能调用间接基类 (如基类的基类) 的构造函数. 但在虚继承中, 虚基类由最终的派生类初始化, 因此必须调用虚基类的构造函数. (虚继承中, 如果由中间基类初始化虚基类的数据成员, 由于有多个中间基类, 因此会出现重复初始化, 可能会引起歧义)

13.9 上机练习

练习 13.1 继承与派生

(a) 编写函数 `int gcd(int a, int b)`, 利用递归 (辗转求余法) 计算最大公约数.

(b) 设计名为 `Integer` 的类, 表示整数, 这个类包括:

- 一个保护型 `int` 数据成员: `a`;

- 一个不带形参的构造函数, 用于设置默认值: `a = 0`;
- 一个带形参的构造函数: `Integer(int a)`.

(c) 设计名为 `Fraction` 的派生类, 表示最简分数, 以公有方式继承 `Integer`, 这个类包括:

- 一个保护型 `int` 数据成员: `b`, 代表分母, 继承的 `a` 代表分子;
- 一个不带形参的构造函数, 用于设置默认值: `a = 0, b = 1`;
- 一个带形参的构造函数: `Fraction(int a, int b)`; (注意分子分母要通分)
- 成员函数 `void Display()`, 以 `a/b` 形式输出分数, 如 `2/3, -3/4`;
- 以非成员函数方式重载加法运算 "`+`", 实现两个分数的加法 (注意分子分母要通分)

  `Fraction operator+(const Fraction & x, const Fraction & y)`

实现这两个类, 并在主函数中测试: 创建分数 `x = 2/3` 和 `y = -1/6`, 计算 `z = x + y`, 并在屏幕上输出 `z`. (程序取名 `hw13_01.cpp`)

练习 13.2 继承与派生: 同名成员屏蔽与类型兼容.

(a) 设计名为 `Point` 的类, 表示平面上的点, 这个类包括:

- 两个私有型 `double` 数据成员: `x, y`, 分别表示横坐标和纵坐标;
- 一个不带形参的构造函数: `Point()`, 用于创建原点 `(0,0)`;
- 一个带形参的构造函数: `Point(double x, double y)`;
- 成员函数 `double dist(const Point& p)`, 返回当前点与给定点的距离.

(b) 在 `Point` 类的基础上定义派生类 `Point3D`, 表示三维空间的一个点, 这个类包括:

- 一个新增私有型 `double` 数据成员: `z`, 表示 z-坐标;
- 一个不带形参的构造函数: `Point3D()`, 用于创建原点 `(0,0,0)`;
- 一个带形参的构造函数: `Point3D(double x, double y, double z)`;
- 成员函数 `double dist(const Point3D& p)`, 返回当前点与给定点的距离.

实现这两个类, 并在主函数中测试:

(1) 创建点 `A1(0,0)` 和 `A2(4,5.6)`, 输出它们之间的距离.

(2) 创建点 `B1(0,0,0)` 和 `B2(4,5.6,7.8)`, 输出它们之间的距离.

(程序取名 `hw13_02.cpp`)

练习 13.3 派生与继承: 虚继承.

(a) 设计名为 `Employee` 的类, 这个类包括:

- 两个保护型数据成员: `string name` 和 `int age`, 分别表示姓名和年龄;
- 一个带两个形参的构造函数 `Employee(const string & name, int age)`, 用于初始化数据成员;
- 成员函数 `void Display()`, 输出所有信息 (姓名和年龄) ;

(b) 在 `Employee` 基础上定义派生类 (以虚继承方式) `Saleman`, 这个类包括:

- 新增保护型数据成员: `int sales`, 表示销售额;
- 一个带三个形参的构造函数, 用于初始化数据成员:

  `Saleman(const string & name, int age, int sales)`
- 成员函数 `void Display()`, 输出所有信息 (姓名、年龄和销售额) ;

(c) 在 `Employee` 基础上定义派生类 (以虚继承方式) `Manager`, 这个类包括:

- 新增保护型数据成员: `int members`, 表示管理的人数;
- 一个带三个形参的构造函数, 用于初始化数据成员:
  `Manager(const string & name, int age, int members)`
- 成员函数 `void Display()`, 输出所有信息 (姓名、年龄和管理人数) ;

(d) 在 `Saleman` 和 `Manager` 基础上定义派生类 `SaleManager`, 这个类包括:

- 一个带四个形参的构造函数, 用于初始化数据成员:
  `SaleManager(const string & name, int age, int sales, int members)`
- 成员函数 `void Display()`, 输出所有信息 (姓名、年龄、管理人数和销售额) .

实现这四个类, 并在主函数中测试: 创建一个 `SaleManager`, 姓名: Zhang San, 年龄: 32, 销售额: 128, 管理人数: 8. 在屏幕上输出该销售经理的相关信息. (程序取名 `hw13_03.cpp`)

第十四讲 多态

本讲主要内容

- 虚函数: 提供了通过基类访问派生类功能的一种机制
- 虚函数实现多态: 只能借助指针或引用
- 抽象类和纯虚函数: 含有纯虚函数的类是抽象类, 抽象类不能实例化
- 模板: 模板函数和模板类, 数据类型参数化

14.1 什么是多态

- 多态 是指同样的消息被不同类型的对象接收时会导致不同的行为, 即接口的多种不同实现方式. 一个典型例子就是函数重载.

  ✍ **注记:** 多态是面向对象程序设计的关键技术之一.

- 常见的多态实现方式:
  (1) 函数重载, 运算符重载
  (2) 虚函数
  (3) 模板

14.2 虚函数

为什么虚函数

我们知道, 用派生类对象替代基类的对象后, 只能发挥基类的功能. 比如基类中定义了成员函数 show, 派生类中也有成员函数 show, 则用派生类对象替代基类对象后, 只能发挥基类的 show 功能. 但能否仍然发挥派生类的 show 功能呢? C++ 中的 虚函数 就是用来解决这个问题.

- 虚函数的声明: 只需要在成员函数声明前面增加 virtual 关键字

```
virtual 数据类型名 函数名(形参列表)
```

例 14.1 多态: 虚函数提供了一种通过基类访问派生类功能的机制.

(ex14_virtual_fun_01.cpp)

```
1 class Person // 基类
2 { ... ... };
3
4 class Student : public Person  // 派生类
5 { ... ... };
```

```
6
7  int main()
8  {
9      Person p1("Gao Dai", 20); // 基类对象
10     Student stu1("Xi Jiajia", 18, 10880108); // 派生类对象
11     Person * p; // 基类指针
12
13     p = &p1; p->show(); // 基类指针 p 指向基类对象，因此调用的是基类的 show()
14
15     p = &stu1; p->show(); // 基类指针 p 指向派生类对象，此时调用的是派生类的 show()
16 }
```

在这个例子里, p 是基类指针, 当它指向的对象是基类的对象时, 则实现的是基类的功能, 而当它指向派生类的对象时, 则能实现派生类的功能. 这表明, 基类指针可以按照基类的方式来做事, 也可以按照派生类的方式来做事, 它有多种形态, 或者说有多种表现方式, 这种现象就是多态 (Polymorphism) .

✍ **注记：** 派生类中声明虚函数时, 关键字 virtual 可以省略, 这意味着, 只要在基类中声明的虚函数, 则所有派生类 (包括派生类的派生类) 中的同名函数 (形参也相同) 都是虚函数.

- 虚函数提供了一种通过基类访问派生类 (包括直接派生和间接派生) 的功能的机制;
- 虚函数**只能借助指针或引用**才能达到多态的效果, 否则无法实现多态, 比如下面的例子就无法实现多态.

例 14.2 多态: 只能通过指针或引用才能实现多态. (ex14_virtual_fun_02.cpp)

✍ **注记：** 引用依附到某个对象上后就不能改变, 不如指针灵活, 因此在多态方面缺乏表现力, 所以实现多态时一般是用指针.

- 关于虚函数的几点说明
  - ▷ 只需在声明时加关键字 virtual, 函数定义 (在外部定义) 时不能加;
  - ▷ 如果派生类中没有对虚函数的具体定义, 则仍使用基类的虚函数;
  - ▷ 对于具有复杂继承关系的大中型程序, 多态可以增加其灵活性, 让代码更具有表现力.

例 14.3 多态: 虚函数在类内部声明, 在类外部定义. (ex14_virtual_fun_03.cpp)

例 14.4 多态: 通过虚函数, 也能实现派生类的派生类的功能 (间接派生). (ex14_virtual_fun_04.cpp)

- 构成多态的条件:

  (1) 必须存在继承关系;

  (2) 继承关系中必须有虚函数, 并且它们形成严格的屏蔽关系, 即基类与派生类中存在具有相同函数原型 (函数名和参数都要相同) 的虚函数;

  (3) 只能通过基类的指针或引用来调用虚函数.

例 14.5 多态: 虚函数必须形成严格的屏蔽关系. (ex14_virtual_fun_05.cpp)

思考: 在上例中, 如果 Point2D 中没有把 void Setpoint(float a, float b) 声明为虚函数, 则程序能否正常运行?

推迟联编/动态联编

有了虚函数, 就可能出现这种情况: 一个函数的调用并不是在编译时刻被确定的, 而是在程序运行时才被确定. 比如下面的例子, 只有当程序运行时, 根据指针 p 指向的是基类对象还是派生类对象, 才能决定调用的是基类的成员函数还是派生类的成员函数. 这种现象就称为 **推迟联编** 或 **动态联编**.

```
1 void fun(Person * p)
2 {
3     p->show(); // Person:show() or Student:show()? 程序运行是才能确定
4 }              // 同一段代码, 可以产生不同效果 → 多态
```

注记: 由于编写代码的时候并不能确定被调用的是基类的成员函数还是派生类的成员函数, 所以称为“虚”函数.

- 构造函数不能是虚函数, 但析构函数可以是虚函数, 而且有时必须声明为虚函数. 虚析构函数主要用于防止内存泄漏, 通常是在派生类中含有指针成员时才会用.

例 14.6 多态: 虚析构函数可用于防止内存泄漏. (ex14_virtual_destructor.cpp)

14.3 纯虚函数与抽象类

14.3.1 纯虚函数

- 纯虚函数的声明:

```
virtual 类型说明符 函数名(形参列表)=0; // "=0" 是纯虚函数的标志
```

- 在虚函数的末尾加“=0”, 表示此函数为纯虚函数.
- 纯虚函数没有函数体.
- 纯虚函数用来规范派生类的行为, 实际上就是所谓的“接口”, 它在基类中只声明不定义. 它的作用仅仅是告诉使用者, 我的派生类都会有这个函数 (功能), 但具体实现由各个派生类定义.
- 如果类中有纯虚函数, 则表示: 我是一个 **抽象类**, 不能实例化, 即不能创建对象.

14.3.2 抽象类

抽象类: 含有纯虚函数的类称为抽象类.

- 抽象类是一种特殊的类, 它通常是为了派生而建立的, 一般处于继承层次结构的较上层;
- 抽象类不能用来声明对象 (即不能实例化), 只能用来派生, 所以是“抽象”的;

- 如果纯虚函数在派生类中也没有具体定义, 则这个派生类还是个抽象类;
- 抽象类的主要作用是将有关的功能作为接口组织在一个继承层次结构中, 由它为派生类提供一个公共的根, 具体实现由派生类完成.

> ✍ **注记:** 定义纯虚函数的一个目的就是让基类不可实例化. 事实上, 有些基类只是用来派生, 如动物、交通工具、几何图形等, 用它来声明对象没有任何实际意义.

14.3.3 举例

例 14.7 纯虚函数与抽象类举例. (ex14_virtual_fun_pure.cpp)

14.4 模板

在 C++ 中, 数据类型也可以通过参数来传递: 在函数定义时可以不指明具体的数据类型, 当发生函数调用时, 编译器可以根据实参确定数据类型. 这就是**数据类型的参数化**.

为什么模板

设计更具通用性的函数和类, 使得它们可以作用于不同类型的数据, 使得程序更加简洁和高效.

- 模板是 C++ 中最强大的特性之一.
- 模板提供了在函数中将类型作为参数的功能.
- C++ 中的模板有 **模板函数** 和 **模板类**.

> ✍ **注记:** 值和类型是一个数据的两个基本特征, 它们在 C++ 中都可以被参数化.

14.5 模板函数

所谓模板函数, 就是一个通用函数, 它涉及的数据 (包括返回值、形参、局部变量) 的类型可以不具体指定, 而是用一个虚拟的类型来代替 (实际上是用一个标识符来占位), 等发生函数调用时再根据传入的实参来确定真正的类型.

- 模板函数的声明和定义:

```
template <typename T> // 模板头, typename 是关键字, T 是形参, 即类型参数
T fun(T x, T y)
{ ... }
```

```
template <typename T1, typename T2, typename T3> // 可以有多个类型参数
T3 fun(T1 x, T2 y)
{ ... }
```

(1) 模板头和函数名是一个整体, 为了使得代码更加清晰, 模板头通常独占一行;

(2) 定义了模板函数, 就可以用类型参数来声明和定义函数, 也就是说, 原来使用 int, float, double 等数据类型说明符的地方, 都可以用类型参数来代替;
(3) 类型参数通过尖括号 “< >” 来传递.
(4) 调用模板函数时, 用于传递类型参数的实参可以是基本数据类型, 也可以是用户自定义的类.

✍ **注记:** 类型参数原则上可以任意合法的标识符, 如 typename mytype, 但使用 T, T1,T2, ... 已经形成了一种惯例.

✍ **注记:** 关键字 typename 也可以用 class 替代, 它们没有任何区别. C++ 早期对模板的支持并不严谨, 没有引入新的关键字, 而是用 class 来指明类型参数, 但是 class 已经用在类的定义中了, 这样做显得不太友好, 所以后来引入一个新的关键字 typename, 专门用来定义类型参数. 不过至今仍然有很多代码在使用 class, 包括 C++ 标准库、一些开源程序等.

例 14.8 模板函数举例: 类型参数通过尖括号 “< >” 来传递. (ex14_template_01.cpp)

例 14.9 模板函数举例: 可以有多个类型参数. (ex14_template_02.cpp)

例 14.10 模板函数举例三: 类型参数也可以是类名. (ex14_template_03.cpp)

- 在调用模板函数时, 类型实参可以省略, 编译器会根据传入的数据自动推断其数据类型 (包括基本数据类型和用户自定义的类) .

例 14.11 模板函数举例: 类型实参可以省略, 自动判断数据类型. (ex14_template_04.cpp)

14.6 模板类

C++ 除了支持模板函数, 还支持模板类. 模板函数的类型参数可用于声明和定义函数, 而模板类的类型参数则可以用在类的声明和实现中. 模板类的目的同样是将数据类型参数化.

✍ **注记:** 前面介绍的向量类 vector 和字符串类 string 都是模板类.

- 模板类的声明:

```
template <typename T>
class 类名
{
    ... ... // 类的声明中可以直接使用类型参数
};
```

- 也可以有多个类型参数:

```
template <typename T1, typename T2, ...>
```

```
class 类名
{
... ... //类的声明中可以直接使用类型参数};
```

- 在类外部定义成员函数时，仍然需要带上模板头，而且类名后面也要带上类型参数，但不需要加 typename

```
template <typename T>
类型说明符 类名<T>::函数名(形参列表) // 类名<T>，此处 T 前面不用加 typename
{
    ... ... // 函数体中可以直接使用类型参数
};
```

  - 如果返回的是一个该类的对象，则类名后面也要带上类型参数:

```
template <typename T>
类名<T> 类名<T>::函数名(形参列表)
{
    ... ... // 函数体中可以直接使用类型参数
};
```

> ✍ **注记:** 事实上，由于此时的类是 类名<T>，因此在所有需要声明该类对象的地方都要写 类名<T> (包括声明该类的对象，函数返回值是该类的对象，函数形参是该类的对象等). 但构造函数和析构函数不需要加类型参数. 另外，在类内部也可以省略类型参数.

例 14.12 模板类举例一 (ex14_template_class_01.cpp)

> ✍ **注记:** 与模板函数不同的是，用模板类创建对象时，必须明确指明数据类型，编译器不能根据给定的数据判断其数据类型.

- 模板类的类型参数可以带缺省值.

例 14.13 模板类举例二 (ex14_template_class_02.cpp)

例 14.14 模板类举例三 (ex14_template_class_03.cpp)

- 几点注记:
  - 模板增强了 C++ 语言的灵活性，虽然不是 C++ 的首创，但是却在 C++ 中大放异彩;
  - C++ 模板有着复杂的语法，我们这里只是做了简单介绍;
  - C++ 模板非常重要，整个标准库几乎都是使用模板来开发的，STL (Standard Template Library，标准模板库) 更是经典之作.

14.7 上机练习

练习 14.1 虚函数.

(a) 设计一个名为 Real 的类, 这个类包括:

- 一个保护型 float 数据成员: x;
- 一个带形参的构造函数: Real(float x);
- 虚成员函数 float Dist(float r), 计算当前对象与给定的 r 之间的距离;

(b) 设计名为 Complex 的类, 以公有方式继承 Real, 这个类包括:

- 一个私有型 float 数据成员: y, 表示虚部, 继承的 x 表示实部;
- 一个带形参的构造函数: Complex(float x, float y)
- 虚成员函数 float Dist(float r), 计算当前对象与给定的 r 之间的距离, 即实部减去 r 后的模长;

实现以上两个类, 并在主函数中测试: 创建 Real 对象: r1 (x = 1.2) 和 Complex 对象 z1 (x = 3.14, y = -2.78), 并通过 Real 类的指针计算 r1 与 r = 2.24 之间的距离, 以及 z1 与 r = 2.24 之间的距离, 在屏幕上输出结果. (程序取名 hw14_01.cpp)

练习 14.2 纯虚函数.

(a) 设计名为 Integer 的类, 表示整数, 这个类包括:

- 一个保护型 int 数据成员: x;
- 一个不带形参的构造函数, 用于设置默认值: x = 0;
- 一个带形参的构造函数: Integer(int x);
- 纯虚成员函数 Display();

(b) 设计名为 Rational 的派生类, 表示有理数, 以公有方式继承 Integer, 这个类包括:

- 一个保护型 unsigned int 数据成员: y, 代表分母, 继承的 x 代表分子;
- 一个不带形参的构造函数, 用于设置默认值: x = 0, y = 1;
- 一个带形参的构造函数: Rational(int x, int y);
- 成员函数 Display(), 以 x/y 形式输出有理数, 如 2/3, -3/4;

(c) 设计名为 Complex 的派生类, 表示整型复数, 以公有方式继承 Integer, 这个类包括:

- 一个保护型 int 数据成员: y, 代表虚部, 继承的 x 代表实部;
- 一个不带形参的构造函数, 用于设置默认值: x = y = 0;
- 一个带形参的构造函数: Complex(int x, int y);
- 成员函数 Display(), 输出复数, 如 2-3i;

实现上面三个类, 并在主函数中测试: 创建有理数 x = 9/19 和复数 z = 3-8i, 并通过 Integer 类的指针在屏幕上输出 x 和 z. (程序取名 hw14_02.cpp)

练习 14.3 模板函数.

(a) 设计名为 Point2D 的类, 表示平面坐标下的一个点, 这个类包括:

- 两个保护型 double 数据成员: x, y, 分别表示横坐标和纵坐标;
- 一个带形参的构造函数: Point(double x, double y);
- 成员函数 double dist(const Point &), 返回当前点与给定点的距离;

(b) 设计名为 `Point3D` 的类, 表示三维空间的一个点, 这个类包括:

- 三个保护型 `double` 数据成员: `x, y, z`, 代表点的坐标;
- 一个带形参的构造函数: `Point3D(double x, double y, double z)`;
- 成员函数 `double dist(const Point3D &)`, 返回当前点与给定点的距离;

(c) 定义模板函数 `double mydist(typename T1, typename T2)`, 使其既可以计算两个 `Point2D` 对象之间的距离, 也可以计算两个 `Point3D` 对象之间的距离.

实现上面两个类和模板函数, 并在主函数中测试: 创建 `A1(1.2,3.4)`, `A2(5.6,7.8)` 和 `B1(1.2,3.4,5.6)`, `B2(9.8,7.6,5.4)`, 并调用模板函数 `mydist` 输出 `A1`, `A2` 之间的距离和 `B1`, `B2` 之间的距离. (程序取名 `hw14_03.cpp`)

练习 14.4 模板类. 用模板类实现 `Complex` 类, 表示复数, 这个类包括:

- 两个保护型数据成员: `real` 和 `imag`, 分别表示实部和虚部 (数据类型相同);
- 一个不带形参的构造函数, 用于创建复数 `0`, 即实部和虚部都为 `0`;
- 一个带形参的构造函数 `Complex(T real, T imag)`, 指定实部和虚部;
- 成员函数 `double Magnitude()`, 计算复数的模长;
- 以成员函数方式重载加法运算 `Complex<T> operator+ (const Complex<T> &)`;
- 成员函数 `void Display()`, 在屏幕上输出一个复数, 如 `2+3i`, `4-5i`.

实现这个类, 要求在类外部定义所有成员函数 (构造函数除外) , 并在主函数中测试:

(1) 定义两个单精度型复数 `z1 = 2.1+5.3i` 和 `z2 = 1.9-2.3i`, 计算 `z1 + z2` 及其模长;

(2) 定义两个整型复数 `a1 = 2+5i` 和 `a2 = 1-2i`, 计算 `a1 + a2` 及其模长;

(程序取名 `hw14_04.cpp`)

> ✍ **注记:** C++ 标准库头文件 `complex` 中定义了 `complex` 模板类及相关运算.

练习 14.5 编写程序, 使用模板实现快速排序算法, 使得其对整型、单精度、双精度等情形都适用. (程序取名 `hw14_05.cpp`)

练习 14.6 编写程序, 使用模板实现矩阵乘积的 Strassen 算法, 使得其对整型、单精度、双精度等情形都适用. (程序取名 `hw14_06.cpp`)

第十五讲 文件流与输出输入重载

本讲主要内容

- 文件流类与文件流对象
- 文件流对象与文件关联
- 文件读写: 文本文件与二进制文件
- 重载输出输入操作运算符 `<<` 和 `>>`

15.1 输入输出流

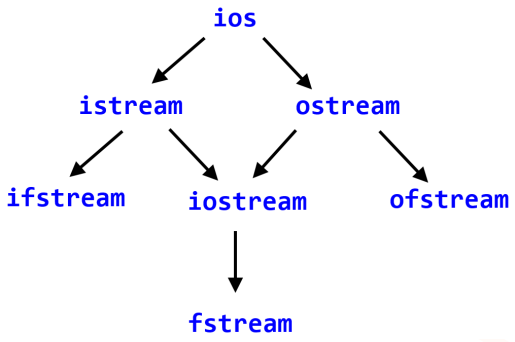
(1) 在 C++ 中, 所有的输入输出都通过“流”来描述的.
(2) 输入流: 数据流向程序, 即 input
(3) 输出流: 数据从程序中流出, 即 output
(4) 具体实现方法: 流类和流对象

15.2 文件流类与文件流对象

- C++ I/O 库中定义的流类

类名	作用	头文件
`ios`	抽象基类	`iostream`
`istream`	通用输入流和其他输入流的基类	`iostream`
`ostream`	通用输出流和其他输出流的基类	
`iostream`	通用输入输出流和其他输入输出流的基类	
`ifstream`	输入文件流类	`fstream`
`ofstream`	输出文件流类	
`fstream`	输入输出文件流类	
`istrstream`	输入字符串流类	`strstream`
`ostrstream`	输出字符串流类	
`strstream`	输入输出字符串流类	

- 各流类继承关系



- 文件流类 (需加头文件: `#include <fstream>`)

```
ofstream      向文件写入数据
ifstream      从文件读取数据
fstream       可以读写文件
```

- 创建文件流对象

```
fstream fstrm;          // 创建一个文件流对象，未绑定到任何文件
fstream fstrm(fname);   // 创建一个文件流，并绑定到文件 fname
fstream fstrm(fname, mode); // 创建文件流的同时指定文件的打开模式
```

(1) 这里的类 `fstream` 也可以是 `ifstream` 或 `ofstream`
(2) `ifstream` 对象所关联的文件只能读
(3) `ofstream` 对象所关联的文件只能写

15.3 文件的打开与关闭

- 文件流对象基本操作 (成员函数)

```
fstrm.open(fname)   // 将文件流对象 fstrm 绑定到文件 fname
fstrm.close()       // 关闭与文件流对象 fstrm 绑定的文件
fstrm.is_open()     // 测试文件是否已顺利打开 (且未关闭)
```

(1) 将文件流对象关联到其它文件时, 须先关闭已绑定的文件
(2) 文件流对象被释放时, `close` 会被自动调用

- 文件打开方式

```
ios::in     // 只读
ios::out    // 只写，若文件存在，则内容被清除
ios::app    // 追加，即每次写操作均定位到文件末尾
ios::ate    // 打开文件后立即定位到文件末尾
ios::Trunc  // 若文件存在，则清除文件中原有的内容
ios::binary // 以二进制方式进行读写
```

(1) 输入输出方式是在 ios 类中定义的
(2) 以上方式可以组合使用, 用“|” 隔开, 如 ios::out|ios::binary
(3) ios::app 通常与 ios::out 组合使用
(4) 在缺省情形下, 文件以文本方式打开
(5) ifstream 对象只能设定 in 模式, 缺省为 in
(6) ofstream 对象只能设定 out 模式, 缺省为 out
(7) fstream 对象可以设定 in 或/和 out 模式
(8) 建议使用 fstream 对象进行文件读写操作

```
ifstream ifstrm;
ofstream ofstrm;
fstream fstrm;

ifstrm.open("fname1"); // 以缺省方式打开
ofstrm.open("fname2", ios::out); // 指定打开方式
fstrm.open("fname3", ios::out|ios::app); // 指定打开方式
```

15.4 文件读写: 文本文件与二进制文件

- 文本文件操作
  (1) 文本文件的写: <<
  (2) 文本文件的读: >> 或 getline
  (3) 我们是如何使用 cin 和 cout 的, 就可以同样来使用文件流对象

```
// 将数据写入文本文件
fstream fstrm("fname.txt", ios::out);
fstrm << "Hello Math!" << endl;
fstrm << "This is an example" << endl;
fstrm.close();
```

```
// 从文本文件中读取数据
char str1[20], str2[20];
fstream fstrm("fname.txt", ios::in);
fstrm >> str1; // 缺省以空格为输入结束符
fstrm.getline(str2, 12); // 也可以用 getline 读取一整行
fstrm.close();
```

例 15.1 文件流: 文本文件的读写. (ex15_fstream_txt_01/02.cpp)

- 二进制文件操作
  (1) 对二进制文件使用 <<、>> 或 getline 是没有意义的
  (2) 写: 使用基类 ostream 的成员函数 write

(3) 读: 使用基类 istream 的成员函数 read

```
write(const char* buf, int n);
read(char* buf, int n); // buf 指向内存中一段存储空间, n 是读写字节数
```

```
1 文件流对象.write(buf,50);
2 // 将 buf 所指定的地址开始的 50 个字节的内容不加转换地写到流对象所关联的文件中.
3 文件流对象.read(buf,30);
4 // 从流对象所关联的文件中, 读入 30 个字节 (或至文件结尾) , 存放在 buf 所指向的内存
    空间内.
```

例 15.2 文件流: 二进制文件的读写. (ex15_fstream_bin.cpp)

15.5 移动或获取文件读写指针

在读写文件时, 有时希望直接跳到文件中的某处开始读写, 这就需要先将文件的读写指针定位到该处, 然后再进行读写操作.

- 文件读写位置: 是指距离文件开头多少个字节, 文件开头的位置是 0.
- 设置文件读指针的位置: 成员函数 seekg, 函数原型为

```
istream & seekg(int offset, int mode);
```

▷ 第一个参数 offset 表示字节数;

▷ 第二个参数 mode 代表文件读写指针的设置模式, 有以下三种选项:

(1) ios::beg → 将文件读指针定位到 offset 字节处, offset 等于 0 则代表文件开头. 在此情况下, offset 只能是非负数.

(2) ios::cur → 在此情况下, offset 为负数则表示将读指针从当前位置朝文件开头方向移动 offset 个字节, 为正数则表示将读指针从当前位置朝文件尾部移动 offset 个字节, 为 0 则不移动.

(3) ios::end → 将文件读指针定位到从文件结尾往前的 |offset| (offset 的绝对值) 字节处. 此时 offset 只能是 0 或者负数.

- 设置文件写指针的位置: 成员函数 seekp, 函数原型为

```
ostream & seekp(int offset, int mode);
```

▷ 参数含义与 seekg 一样.

- 获取当前读写指针的具体位置:

```
int tellg(); // 返回读指针位置
int tellp(); // 返回写指针位置
```

将文件读指针定位到文件尾部, 然后用 tellg 获取文件读指针的位置, 即可获取文件长度.

```
fstrm.seekg(0,ios::end); // 定位读指针到文件尾部
length = fstrm.tellg(); // 文件长度 = length + 1
```

15.6 重载 << 和 >>

IO 标准库分别使用 << 和 >> 执行简单的输出和输入操作, 为了使得它们也适用于新定义的类, 即也能用 << 和 >> 进行相应对象的输出和输入, 需要对这两个运算符进行重载.

```
Rational a(1,2);
cout << a << endl; // 需重载 << 才能实现
```

- 通过具体例子来说明如何重载 << 和 >>

例 15.3 文件流: 重载 << 和 >> (ex15_overload.cpp)

返回值必须是引用, 这是为了允许该运算符可以在单个表达式中多次使用, 如

```
cout << a << b << endl;
```

根据运算顺序, 该表达式等价于

```
((cout << a) << b) << endl;
```

如果 cout << a 返回的不是一个引用, 则不能作为左值, 因此语句 (cout << a) << b 就会出错!

关于输入输出运算符重载的几点说明

- 只能以 非成员函数 方式重载, 因此需要在类内部将其声明为友元函数;
- 通常情况下, 第一个形参是引用 (因为需要修改, 所以不能是常引用);
- 重载 << 时, 第二个形参通常是常引用 (绑定到需要输出的对象);
- 在重载 << 和 >> 时, 尽量减少格式化操作! 如换行等.

15.7 上机练习

练习 15.1 用文件流方式完成下面操作: 创建一个 6×6 的矩阵 A, 其元素为 $[0,1]$ 之间的双精度数.

(a) 将其按矩阵形式写入到一个文本文件 fout.txt 中;

(b) 将其写入到一个二进制文件 fout.dat 中;

(c) 从文件 fout.dat 中读取前 12 个数据 (双精度), 构成一个 2×6 的矩阵 B, 并将 B 按行输出.

(程序取名 hw15_01.cpp)

练习 15.2 重载复数类的输出运算符 "<<" 和输入运算符 ">>", 输出格式为: a+bi, 如: 3+4i, 5-6i.

(程序取名 hw15_02.cpp)

练习 15.3 设计名为 PolyInt 的类, 表示整系数多项式 (即系数为整数), 包括:

- 一个 `int` 型数据成员: `int n`, 表示多项式的次数;
- 一个 `int` 型指针数据成员: `int * a`, 表示多项式的系数, 即

$$p(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n,$$

  其中 `a` 指向系数向量 $[a_0, a_1, \ldots, a_n]$ 的首地址;
- 不带形参的构造函数: `PolyInt()`, 用于创建多项式 $p(x) = 0$, 即 `n = 0, a[0] = 0`;
  (提示: 在构造函数中用 `new` 为对象申请存储空间)
- 带一个形参的构造函数: `PolyInt(int n)`, 用于创建系数全部为 `0` 的 `n` 次多项式;
- 带两个形参的构造函数: `PolyInt(int n, int a[])`;
- 以成员函数方式重载赋值运算符 "=": `PolyInt & operator=(const PolyInt & p)`;
- 以成员函数方式重载运算符 "()", 计算多项式在给定点的值:
  `int operator()(const int x)`
  (如 $p(x) = 1 + 2x + 4x^3$, 则 $p(2) = 37$.)
- 以非成员函数方式输出运算符 "<<", 如 $p(x) = 1 + 2x + 4x^3$ 输出为 `[1,2,0,4]`,
  `friend ostream & operator<<(ostream &, const PolyInt &)`
- 使用非成员函数方式实现两个整系数多项式的乘法运算
  `PolyInt operator*(const PolyInt & p, const PolyInt & q)`
- 析构函数: `~PolyInt()`, 释放由 `new` 申请的存储空间.

实现这个类, 并在主函数中测试这个类: 创建整系数多项式 $p(x) = 1 + 2x + 4x^3$ 和 $q(x) = 2 - 4x^2 - 3x^3 + x^5$, 计算 $p(x)$ 和 $q(x)$ 的乘积 $s(x) = p(x)q(x)$, 并在屏幕上输出 $p(x), q(x), s(x)$ 以及 $p(2)$ 和 $s(2)$ (即 $p(x)$ 和 $s(x)$ 在 $x = 2$ 处的值).
(程序取名 `hw15_03.cpp`)

练习 15.4 抽奖小程序: 共有 N 人参加抽奖 (名单从 namelist.txt 中读取), 先随机抽取 $N3$ 个三等奖, 然后随机抽取 $N2$ 个二等奖, 最后随机抽取 $N1$ 个一等奖, 其中 $N3 > N2 > N1$, 且 $N3+N2+N1 < N$. 编写程序, 从名单中读入所有参加抽奖人名, 然后依次抽取 10 个三等奖, 5 个二等奖, 1 个一等奖, 并输出获奖名单. (不得参加重复抽奖, 程序取名 `hw15_04.cpp`. 提示: 可以用 `string[N]` 数组存储抽奖名单)

练习 15.5 设计名为 `Vector` 的类, 表示一维数组, 这个类包括:

- 私有数据成员 `int n`, 表示数组长度; 私有数据成员 `float * px`, 指向数组首地址;
- 一个带形参的构造函数: `Vector(int n, float x[])`;
  (提示: 在构造函数中用 `new` 为对象申请存储空间)
- 重载运算符 "<<", 实现 `Vector` 对象的输出, 如 "`[1.1, 1.2, 1.3]`";
- 析构函数: `~Vector()`, 释放构造函数中由 `new` 申请的存储空间.

实现上面的类, 并在主函数中测试: 创建 `Vector` 对象: `x=[1.1, 1.2, 1.3]`, 并在屏幕上通过 "`cout << x`" 输出. (程序取名 `hw15_05.cpp`)

第十六讲 标准模板库

本讲主要内容

- 容器
  - 顺序容器/序列容器
  - 关联容器
  - 容器适配器
- 算法: 容器所具有的功能
- 迭代器: 访问容器中数据的方法

标准的 C++ 由三个重要部分组成

(1) 核心语言: 提供 C++ 程序设计基本构件, 包括变量、数据类型、控制结构等.
(2) C++ 标准库: 提供大量的函数, 用于数学计算、字符串处理、文件操作、格式化输入输出等.
(3) 标准模板库: 提供大量的模板和方法, 用于操作数据结构等.

16.1 STL 标准模板库

C++ STL (Standard Template Library) 是一套功能强大的 C++ 模板库, 提供了大量的通用模板类和模板函数, 这些模板类和模板函数可以实现多种流行和常用的数据结构 (如向量、链表、队列等) 和算法 (如插入、排序、搜索、查找等).

- STL 三大核心组件
  - 容器 (Containers) : 用来存储数据的一种数据结构 (模板类) , 如向量, 链表
  - 算法 (Algorithms) : 对数据的各种操作 (模板函数) , 如插入, 排序, 搜索等
  - 迭代器 (iterators) : 访问容器中的数据的方法, 作用类似于指针

16.2 容器

C++ 容器包括顺序容器, 关联容器和容器适配器.

- 顺序容器 (也称序列容器, Sequential Containers) : 按顺序 (物理层面或逻辑层面) 存储数据, 常见的有数组, 链表等.

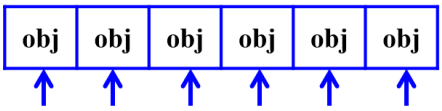
表 16.1. 常见的顺序容器

`array`	数组, 长度不能改变

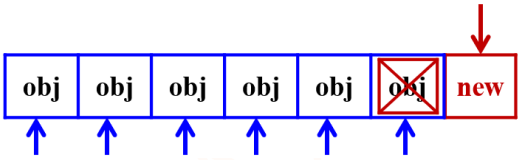
vector	可变长度的向量, 只能在最后面插入或删除数据
deque	与 vector 类似, 允许在最前面和最后面插入或删除数据
list	双向链表, 可在任意位置插入或删除数据
forward_list	与 list 类似, 但是单向的, 只能沿一个方向访问
string	字符串, 与 vector 类似, 但存储的是字符

✍ **注记：** 数组和向量的顺序存放指的是物理层面, 而链表的顺序存放则是指逻辑层面.

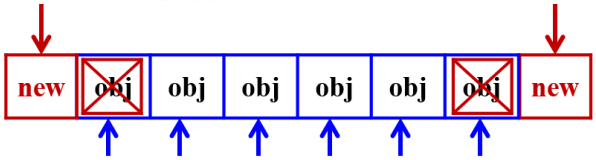
- 不同顺序容器示意图
  - array: 数组, 长度固定, 可随意访问其中的任何元素



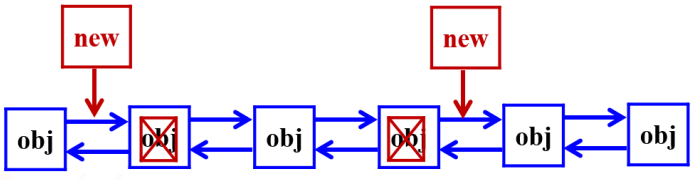
  - vector: 长度可变的数组, 但只能在最后面添加或删除数据



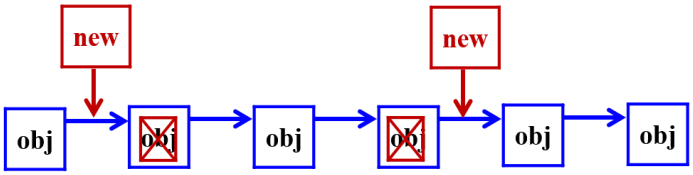
  - deque: 与 vector 类似, 但可在两头添加或删除数据



  - list: 双向链表, 可在任意位置添加或删除数据, 但只能从第一个元素或最后一个元素开始访问



  - forward_list: 单向链表, 与 list 类似, 但只能单向访问



- 关联容器 (Associative Containers) : 按排序方式存储数据, 就像词典一样, 方便搜索.

表 16.2. 常见的关联容器

set	存储已经排好序的互不相同的数据, 在插入新数据时会自动进行排列
unordered_set	与 set 类似, 但按每个数据的 Hash 值排序
map	存储“键-值”, 每个数据都有唯一的“键”与之对应, 数据按键的大小排序
unordered_map	与 map 类似, 但按“键”的 Hash 值排序
multiset	与 set 类似, 但允许存在相同的数据
unordered_multiset	与 unordered_set 类似, 但允许存在相同的数据
multimap	与 map 类似, 但不要求“键”唯一
unordered_multimap	与 unordered_map 类似, 但不要求“键”唯一

- 容器适配器 (Associative adapters) : 顺序容器和关联容器的变种, 增加一些特殊功能

表 16.3. 常见的容器适配器

stack	栈, 按后进先出 (LIFO) 方式存储数据
queue	队列, 按先进先出 (FIFO) 方式存储数据
priority_queue	队列, 但能保证最大元素总在最前面

- 容器的选择. 不通的容器有着不同的特点和性能, 适用不同的场合, 使用时可以根据不同的应用场景来选择.

例 16.1 容器举例: vector 和 list 性能比较. (ex16_vector_list_comp.cpp)

16.3 算法

算法 Algorithms: 对容器进行的各种操作, 如插入, 删除, 排序, 查找, 反转等, 通常是模板函数.

表 16.4. 常见的算法

find	查找指定的值
find_if	根据条件查找
reverse	反转

`remove_if`	根据条件删除相应的数据
`transform`	根据用户给定的方法对数据进行交换

注: 使用时要加入 `algorithm` 头文件

STL 算法的优势

- 代码简洁, 可读性强. STL 算法通过高层次的抽象, 减少代码量, 使得程序逻辑更加清晰.
- 性能优化. STL 算法通常经过高度优化, 能够充分利用数据结构、编译器优化和硬件特性等, 提供更高的性能.
- 减少错误. STL 算法通过标准化接口减少一些可能存在的错误, 如循环中的边界错误和逻辑错误等.
- 可维护性好. 简洁的 STL 算法使得代码更易于理解和维护, 特别是在团队合作中.

表 16.5. 容器常见的成员函数

`begin()`	返回开始迭代器
`end()`	返回结束迭代器
`size()`	返回实际元素个数
`capacity()`	返回当前容量
`empty()`	判断是否为空
`max_size()`	返回元素个数的最大值
`front()`	返回第一个元素的引用
`back()`	返回最后一个元素的引用
`push_back()`	在序列的尾部添加一个元素
`pop_back()`	移出序列尾部的元素
`clear()`	移出所苻的元素, 容器大小变为 0
`resize()`	改变实际元素的个数
`at()`	使用索引访问元素, 会进行边界检查
`assign()`	用新元素替换原有内容
`insert()`	在指定的位置插入一个或多个元素
`erase()`	移出一个元素或一段元素
`swap()`	交换两个容器的所有元素

`data()`	返回包含元素的内部数组的指针
`sort()`	对元素进行排序

- ▷ 这里仅列出部分算法;
- ▷ 并非所有容器都具有这些功能;
- ▷ 容器不仅使用方便, 而且效率也非常高, 可代替数组;
- ▷ 优先使用 `vector` 和 `string`.

16.4 迭代器

- 迭代器 (Iterators) , 访问容器中数据的方法, 比如指针.
  (1) 要访问容器的数据, 需要通过迭代器;
  (2) 迭代器是算法与容器之间的桥梁.

  - ▷ 实际上, 迭代器就是一个泛型指针;
  - ▷ 算法是作用在迭代器上, 而不是容器上;
  - ▷ 可以根据需要自己定义迭代器.

例 16.2 容器、算法、迭代器举例. `(ex16_iterator.cpp)`

主要参考文献

[1] Y. D. Liang 著, 刘晓光, 李忠伟, 任明明, 王刚译, C++ 程序设计, 第 3 版, 机械工业出版社, 2015.

[2] S. B. Lippman, J. Lajoie and B. E. Moo 著, 王刚, 杨巨峰译, C++ Primer, 中文版, 第 5 版, 电子工业出版社, 2013.

[3] S. Prata 著, 张海龙, 袁国忠译, C++ Primer Plus, 中文版, 第 6 版, 人民邮电出版社, 2012.

[4] B. Stroustrup 著, 王刚, 杨巨峰译, C++ 程序设计语言, 第 4 版, 2013; 机械工业出版社, 2016.

[5] B. Stroustrup, C++ 程序设计原理与实践, 第 2 版, 2014; 任明明, 王刚, 李忠伟译, 机械工业出版社, 2017; 张兴, 蔡乐, 赵林涛译, 清华大学出版社, 2024.

[6] M. A. Weiss 著, 冯舜玺译, 数据结构与算法分析: C++ 语言描述, 第四版, 电子工业出版社, 2016.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein 著, 殷建平, 徐云, 王刚等译, 算法导论, 第 3 版, 机械工业出版社, 2013.

[8] https://cppreference.cn, 参考手册, 含最新标准.