



# C++哈氏教程

第二版

cppHusky



# 序

本书为 *cppHusky* 的二〇二四年礼。

*cppHusky*



# 目录

序	i
<b>泛讲篇</b>	<b>3</b>
<b>第一章 初识 C++</b>	<b>3</b>
1.1 开始一个 C++ 程序 . . . . .	3
1.2 数据与信息 . . . . .	6
1.3 数据的定义和使用 . . . . .	7
1.4 运算符与类型 . . . . .	11
1.5 <code>sizeof</code> 与内存空间 . . . . .	13
1.6 类与对象 . . . . .	16
<b>第二章 数据的基本操作</b>	<b>19</b>
2.1 赋值语句与常量 . . . . .	19
2.2 基本数据类型 . . . . .	22
2.3 运算符 . . . . .	27
2.4 类型转换 . . . . .	32
<b>第三章 程序的流程控制</b>	<b>37</b>
3.1 简介：结构、流程与次序 . . . . .	37
3.2 选择结构 . . . . .	45
3.3 循环结构 . . . . .	51
3.4 作用域初步 . . . . .	60
3.5 实操：简易计算器的设计 . . . . .	61
<b>第四章 函数初步</b>	<b>65</b>
4.1 函数的概念 . . . . .	65
4.2 函数的定义和使用 . . . . .	67
4.3 函数重载 . . . . .	73
4.4 默认参数的设置 . . . . .	76
4.5 函数递归 . . . . .	78
4.6 函数模版简介 . . . . .	80
<b>第五章 复合类型及其使用</b>	<b>83</b>
5.1 指针 . . . . .	83
5.2 “常量指针” 与 “指向常量的指针” . . . . .	92

5.3 (左值) 引用与引用参数传递 . . . . .	94
5.4 一维数组 . . . . .	98
5.5 字符串 . . . . .	107
5.6 指针与数组的复合类型 . . . . .	112
5.7 动态内存分配 . . . . .	120
<b>第六章 自定义类型及其使用</b>	<b>125</b>
6.1 枚举常量 <code>enum</code> . . . . .	125
6.2 结构体 <code>struct</code> . . . . .	127
6.3 实操: 用结构体实现单链表 . . . . .	132
6.4 联合体 <code>union</code> . . . . .	139
6.5 类 <code>class</code> 初步 . . . . .	144
<b>第七章 代码工程</b>	<b>149</b>
7.1 跨文件编译 . . . . .	149
7.2 命名空间 . . . . .	154
7.3 作用域、存储期和链接 . . . . .	159
7.4 编码风格 . . . . .	165
<b>第八章 类与函数进阶</b>	<b>171</b>
8.1 运算符重载 . . . . .	171
8.2 成员函数与友元函数 . . . . .	178
8.3 构造与析构 . . . . .	186
8.4 成员的属性 . . . . .	195
8.5 类型转换函数 . . . . .	204
8.6 复合类型与对象 . . . . .	214
8.7 实操: 简单的 <code>string</code> 类 . . . . .	217
<b>第九章 类的继承</b>	<b>241</b>
9.1 基本概念 . . . . .	241
9.2 公开继承与受保护成员 . . . . .	243
9.3 私有继承 . . . . .	248
9.4 多级继承 . . . . .	257
<b>第十章 继承中的常见问题</b>	<b>261</b>
10.1 继承中的类型转换 . . . . .	261
10.2 虚函数与多态 . . . . .	265
10.3 抽象基类与纯虚函数 . . . . .	270
<b>精讲篇</b>	<b>279</b>
<b>附录 A C++ 运算符基本属性</b>	<b>279</b>
<b>附录 B ASCII 码表 (0 到 127)</b>	<b>281</b>

<b>附录 C 相关数学知识</b>	<b>283</b>
C.1 数制转换 . . . . .	283
C.2 布尔代数基础 . . . . .	292
<b>跋</b>	<b>293</b>



# 泛讲篇



# 第一章 初识 C++

在本章，我们将会了解一些 C++ 中最基本的知识，最常见的概念，和最简单的语法，并付诸代码来实现。

我们不要求读者搞懂每句代码的含义——那是不必要的。我们在本章中会给出一个基本的代码模版，读者只需修改其中的一小部分即可。毕竟，学习都是从模仿开始的，C++ 也不例外。

有一点值得注意。我们需要有一个编译器，来把我们的代码 (.cpp 文件或者别的什么) 变成可以执行和看到效果的程序（Windows 中的.exe 文件、Linux 或 MacOS 上的可执行文件，或者别的什么），以便检查它能否实现我们想要的功能，并在日后使用这个程序。

根据自己计算机的系统，你可以选择[Embarcadero Dev-C++ \(Windows\)](#), [Microsoft Visual C++ \(Windows\)](#), [Apple Xcode \(MacOS\)](#), [Open Watcom C++ \(Windows/Linux/DOS\)](#), [Code::Blocks \(跨系统\)](#), [MonoDevelop \(跨系统\)](#)。这些都是集成开发环境<sup>1</sup>，你只需要在下载后做很简单的配置，就可以开始写代码了。

另外还有一些代码编辑器，比如[Visual Studio Code \(Windows/Linux/MacOS\)](#), [Sublime Text \(Windows/Linux/MacOS\)](#), [Vim \(跨系统\)](#)，你需要做额外的配置（比如安装插件），把它们变得如同集成开发环境一样。

另有一些在线的代码编译工具，让你可以无需安装软件，只要在浏览器上提供代码，就能远程编译并告诉你运行结果。比如[Coliru](#)和[Wandbox](#)。不过相比于本地编译运行，这种方式的交互效果就相当差劲了，仅适合一些无输入或只有简单输入的程序。这些在线编译方式对于本章涉及的程序来说当然够用，但不要太过依赖。

## 1.1 开始一个 C++ 程序

各种编程语言的教程都喜欢用“输出 Hello World!”作为第一个编程示例，本书也不例外。

在 C++ 中，我们可以用如下的代码来输出“Hello World!”：

代码 1.1: Hello\_World.cpp

```
// 这是一个简单的C++程序，用于输出Hello World
#include <iostream> // 标准输入输出头文件
using namespace std; // 使用 std 命名空间
int main() { // 主函数，程序执行始于
    cout << "HelloWorld!"; // 输出Hello World
    return 0;
}
```

如果你使用的是集成开发环境，那么只需要用一两个选项或快捷键就可以完成编译、运行步骤，并看到程序的运行结果了。这个快捷键因软件而异，不过总得说来，可以概括成三个过程：

<sup>1</sup>集成开发环境（Integrated Development Environment, IDE），是一种辅助进行软件开发的应用。在开发工具内部就可以编写源代码，并编译打包成为可用的程序。IDE 需要的配置较少，对新手比较友好，上手容易。

1. 编译 (Compile): 编译器将你写在文件中的源代码<sup>2</sup>编译成目标代码<sup>3</sup>。
2. 链接 (Link): 将目标代码与其它代码链接。C++ 标准库提供了非常丰富的功能，而链接器负责将我们的源代码与这些功能对应的库，以及其它必需文件结合起来，形成可执行文件。
3. 运行 (Run): 可执行文件在运行时，可以进行计算、存取和用户交互等各种行为。程序可以长期处于运行状态，也可以在某个指令后结束运行。

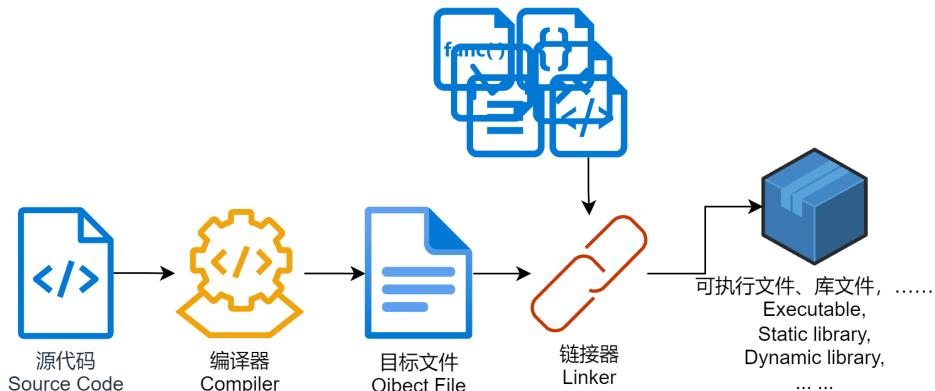


图 1.1: 从源代码到可执行文件

实际上的过程会更复杂，比如在编译之前还会进行预处理（Preprocess）。但是我们现在还不需要操心这么多，因为在按下“编译”键之后，预处理和编译都会进行，所以不妨把它统称为一个过程。

## 编译器的选择

C++ 的编译器（Compiler）非常丰富。[List of C++ Compilers - Wikipedia](#) 条目为我们提供了一个不完全的 C++ 编译器列表。

一般我们选择使用 Clang、GCC 或 Visual C++ (MSVC)，它们都是跨系统的编译器，对语言标准的支持比较好<sup>4</sup>，许多 IDE (如 Dev\_C++, Code::Blocks, Visual C++ 等) 也都默认提供这些编译器之一。另外，Coliru 和 Wandbox 等在线编译器也都支持使用 Clang 和 GCC。

同种的编译器也会有版本上的差异。很多古老的版本 (如 GCC 4.9.4) 并不支持新近的语法 (如 C++17<sup>5</sup>)。新版本的编译器往往有很多好用的特性，比如对代码有更好的优化。<sup>6</sup>

## 语言标准的选择

不同的 C++ 标准 (Standard) 对语法的支持不同。比如说，在 C++11 以后的标准中下，可以使用 `auto` 来自动确定我们要定义的变量类型。这是一种很方便的语法，但 C++11 以前的版本就不支持使用。

C++ 语言标准的更新总会带来很多新的功能，也会对以往标准中的缺陷和漏洞加以弥补，所以越新的标准往往就越强大、易用。另一方面，新标准通常都能兼容旧标准<sup>7</sup>，所以你也无需担心为了适应新标准而放弃很多旧知识，或者换用新标准后很多代码编译失败了。

<sup>2</sup>源代码 (Source Code)，是一系列人类可读的计算机语言指令，在 C++ 中一般指.cpp 文件中的代码。

<sup>3</sup>目标代码 (Object Code)，是源代码经编译器或汇编器处理后产生的代码，对人类来说可读性很差。机器不能直接执行源代码，只能执行目标代码。

<sup>4</sup>Clang 和 GCC 支持 C++17 以前标准和 C++20 的大部分标准，而 MSVC 支持 C++20 的全部标准和 C++23 的部分标准。

<sup>5</sup>C++17 是 C++ 委员会制定的一个语言标准，其最终版发布于 2017 年 12 月。这也是本书采用的主要语言标准。

<sup>6</sup>另请注意，最好选择稳定版本的编译器，而非试验性的测试版，以免出现问题带来的麻烦。

<sup>7</sup>只是通常如此。有很多反例，比如 C++17 标准移除了古已有之的 `std::auto_ptr`。

可惜并非所有编译器都能支持 C++ 新标准的特性。因此，在编写本书时，考虑到许多编译器对语言标准支持的情况，本书选择使用 C++17 标准。今后若无特殊说明，默认本书所有知识和代码遵循 C++17 标准。

### Windows 下运行的可能问题

在 Windows 系统中，很多 IDE 允许你在一个辅助窗口中运行程序（我本人比较习惯这种方式）。有些 IDE 在程序执行完毕后，会保留辅助窗口，等待用户“按任意键继续”，于是我们有足够的时间看到程序运行的结果；还有一些 IDE 在程序结束时会立刻关闭辅助窗口，我们就来不及看程序运行的结果。

为了解决这个问题，我们可能需要对代码做一些修改：

```
//... 前面的代码不变
cin.get(); // 可能需要添加的语句
cin.get(); // 甚至可能需要再加这句
return 0;
}
```

`cin.get()` 的作用是读取下一个键盘输入，所以如果你不按下键盘上的 Enter 键，程序就会一直等待下去。（只有按下 Enter，输入内容才能发送给程序，我们后面会慢慢道来）

至于用两个 `cin.get()`，这是为了应对可能的 '`\n`' 遗留被第一个 `cin.get()` 吞没。我们现在尚无必要在乎这些细枝末节，只需要知道：如果窗口没有保持打开，就加一个 `cin.get()`；如果还不行，就加两个 `cin.get()`。<sup>8</sup>

## 回到 Hello World

现在让我们回来看 Hello World 代码。这个代码可以在任何操作系统上，用任何编译器和任何语言标准来编译。程序的输出就是一个单词 `Hello World!`。

```
// 这是一个简单的C++程序，用于输出Hello World
```

程序的第一行是一个注释（Comment）。注释以两个双斜线开头，其后直到行末的内容均会被编译器忽略，所以你写什么都可以。注释是为了给人类阅读的，以便我们理解程序和代码的含义。

```
#include <iostream> // 标准输入输出头文件
```

程序的第二行是一个文件包含（File Inclusion）。通过这个语句，预处理器会对代码作预处理<sup>9</sup>，将本句命令替换成 `iostream` 库中的代码。

```
using namespace std; // 使用 std 命名空间
```

程序的第三行是一个命名空间（Namespace）使用。C++ 中，`cout` 被定义<sup>10</sup>在 `std` 命名空间中，我们要通过 `std::cout` 这样的语法才可以使用它。而有了 `using namespace std;` 之后，我们就可以直接使用 `cout`。

关于文件包含和命名空间的概念，我们会在第七章进行讲解。目前我们无需过分纠结。

```
int main(){// 主函数，程序执行始于此
    // 函数体代码
}
```

<sup>8</sup>值得留意的是，用 `cin.get()` 来阻断程序进度只是一种临时的解决方案，目的在于便于我们进行学习和演示。而在实际的开发中，人们通常不会在程序中添加这种代码。

<sup>9</sup>如前所述，源代码在编译之前需要经过预处理，例如文件包含和宏。预处理结束后，编译才会开始。

<sup>10</sup>定义（Definition），是为程序中的一个实体（如变量、函数、类等）分配内存并提供其具体的实现或值的过程。在以 C++ 为代表的静态语言中，一个实体必须定义才能使用。

程序的第四行是主函数（Main Function）的定义。主函数是程序的入口，它必须唯一。程序从主函数开始按顺序执行各句代码，除非遇到了条件或循环结构。关于函数，我们会在第四章讲解；关于结构，我们会在第三章讲解。

```
cout << "HelloWorld!"; // 输出 Hello World
```

程序的第五行是输出（Output）语句。这一句对我们来说更重要，因为我们接下来要模仿的代码主要是基于修改输出内容来实现的。

<< 左侧的 cout 是在标准库 iostream 中定义的 std 空间的对象，控制标准输出。我们无需对它作修改；<< 右侧的 "HelloWorld!" 则是我们要输出的内容。外套双引号表示它是一个字符串，我们会在第五章讲解字符串。

如果我们想要输出一个数字，我们只需要修改 << 右侧的内容即可，比如

```
cout << 3.14159; // 输出圆周率小数点后五位
```

第三节我们就会开始介绍如何使用 cout 来输出数据。

## 1.2 数据与信息

我们日日夜夜都在与数据打交道。

早上醒来，看一眼天气预报，发现今天的最低气温是-5°C，比昨天低了好多，看来要加衣服了；匆忙赶到教室，看了一眼时间，发现刚好是 7:59，庆幸自己没有迟到；中午去超市购物，发现面包促销，全场打 8 折，就买了好多回去；睡前称一下体重，发现比上个月瘦了 2 斤，心中暗暗窃喜。像这样，温度、时间、价格、体重等等，都是具象的数据（Data）。

我们还可能遇到抽象的数据。学生的学号、员工的工号、公民身份证号，这些都是“编号”，它们比那些一眼就明白含义的数据要抽象，如果不加解释，我们就很难看懂，只会觉得那是一串无规律的数字。

有些数据的形式不太常规，比如一个人的姓名，它看上去又像数据，又不像数据——我们怎么能把汉字当成数字一样的东西呢？要解决这个困惑，我们可以给所有汉字进行编码（Encoding）。比如说，“赵钱孙李”分别定为“1, 2, 3, 4”，那么我们就可以把“赵”记为“1”，把“孙”记为“3”。像这样把所有汉字都编成表，接下来我们就可以用这张表来把汉字翻译成数字了。

汉字	编码	汉字	编码	汉字	编码	汉字	编码
赵	1	钱	2	孙	3	李	4
...	...	...	...	...	...	...	...
何	21	吕	22	施	23	张	24
...	...	...	...	...	...	...	...
一	569	二	570	三	571	四	572
...	...	...	...	...	...	...	...

表 1.1: 一个简单的汉字-数字编码表

于是我们根据这个表，可以查出“张三”对应“24-571”，而“李四”对应“4-572”。这就是一个简单的编码过程。

同样的道理，我们如果拿到了“24-571”，就可以根据这张表查出“张三”；拿到了“4-572”，就可以根据这张表查出“李四”。这就是一个简单的解码（Decoding）过程。

地址也是数据，我们可以对地址进行编码。例如，中国邮政使用六位数字组成的邮政编码来划分邮件投递区域，比如说北京市是“10××××”，上海市是“20××××”。这样就可以用数据来指代我们想要表达的信息了。

还有更抽象的数据。一本电子书，一部电影，一款电子游戏，也是数据<sup>11</sup>。在计算机中，它们都有相应的文件扩展名，像是.epub, .mp4, .exe之类。这些文件都以某种编码方式储存在计算机里，扩展名不同，说明它们的编码方式不同。如果要了解这些文件中的信息，就需要用正确的打开方式（比如下载支持.epub格式的电子书阅读器）才能对内容进行解码。

## C++ 中的基本数据

说了这么多，让我们回来看看 C++ 中的数据吧。C++ 的数据可以分为两大类：基本类型（Fundamental type）和复合类型（Compound type）。本章删繁就简，只介绍三种基本的数据类型：整数、浮点数（小数），字符。

### 整数类型（Integer type）

顾名思义，整型就是专为表示整数而设计的类型。它不能表示小数。

C++ 为我们提供了很多内置的整数类型，我们可以直接使用。不同类型的区别主要在于以下两点：其一，有些可以表示负数，而有些不允许表示负数；其二，有些表示的数据范围较宽（我们可以把这个范围理解成“容量”），有些则较窄<sup>12</sup>。我们目前无需追究其中的细节，我会在第二章和后面的精讲篇中详细讨论这个问题。

### 浮点类型（Floating-point type）

浮点类型有点像“科学记数法”，一个浮点数既有表示分数的部分（Fraction），又有表示幂次的部分（Exponent bias）。它既可以表示小数，也可以表示整数，并且数据范围非常宽<sup>13</sup>。

C++ 同样提供了内置的浮点类型。不同类型的区别主要在于以下两点：其一，它们能表示的有效位数各不相同；其二，它们的幂指数范围有宽有窄。

### 字符类型（Character type）

严格说来，字符型是一种特殊的整型。不同之处在于，字符数据不是用来“计算”而是用来“编码”的。

以最原始而最通用的 ASCII 码<sup>14</sup>为例，它有若干控制字符和可显示字符，每个字符都有一个编码，如图 1.1 所示。读者无需记忆本图，只需知道它是一套编码系统即可。通过 ASCII，每个拉丁字母都能编码成一个整数，比如字符'0' 对应着编码 48，字符'A' 对应着编码 65，字符'a' 对应着编码 97。

这套字符编码只适用于现代拉丁字母。如果要表示其它语言（比如中文）的字符，就需要用更复杂的编码规则。我们会在精讲篇中细细道来。

## 1.3 数据的定义和使用

前一节我们介绍了三种数据类型：整型、浮点型和字符型。这一节让我们回到代码中来，看看如何定义一个数据，又如何使用它。

C++ 中有一些关键字，用特定的关键字就可以定义特定类型的数据，比如

<sup>11</sup>广义地讲，实体书、绘画、雕塑等等也是数据。但这些已经远远超出了我们的讨论范围。

<sup>12</sup>容量是与内存空间有关的。如果某个类型占用的字节数为 2，那么它只能表示  $256^2 = 65536$  个不同的整数。因此它无法表示十万以上的数据，我们需要用更宽的类型来表示。

<sup>13</sup>值得注意的是，浮点类型的数据都有精度限制，不能准确表示有效数字过多的数据。正因如此，浮点类型不适合用来表示和计算特别大的整数；这时应当使用整型。

<sup>14</sup>美国信息标准交换代码（American Standard Code for Information Interchange, ASCII），是一套基于拉丁字母的编码系统，主要用于显示现代英语。

b <sub>7</sub>	b <sub>6</sub>	b <sub>5</sub>	b <sub>4</sub>	b <sub>3</sub>	b <sub>2</sub>	b <sub>1</sub>	Column	0	0	0	1	0	1	1	0	1	0	1	1	1
0	0	0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p						
0	0	0	0	1	1	1	SOH	DC1	!	1	A	Q	a	q						
0	0	1	0	2	2	2	STX	DC2	"	2	B	R	b	r						
0	0	1	1	3	3	3	ETX	DC3	#	3	C	S	c	s						
0	1	0	0	4	4	4	EOT	DC4	\$	4	D	T	d	t						
0	1	0	1	5	5	5	ENQ	NAK	%	5	E	U	e	u						
0	1	1	0	6	6	6	ACK	SYN	&	6	F	V	f	v						
0	1	1	1	7	7	7	BEL	ETB	'	7	G	W	g	w						
1	0	0	0	8	8	8	BS	CAN	(	8	H	X	h	x						
1	0	0	1	9	9	9	HT	EM	)	9	I	Y	i	y						
1	0	1	0	10	10	10	LF	SUB	*	:	J	Z	j	z						
1	0	1	1	11	11	11	VT	ESC	+	:	K	[	k	{						
1	1	0	0	12	12	12	FF	FS	,	<	L	\	l	:						
1	1	0	1	13	13	13	CR	GS	—	=	M	]	m	}						
1	1	1	0	14	14	14	SO	RS	.	>	N	^	n	~						
1	1	1	1	15	15	15	SI	US	/	?	O	—	o	DEL						

图 1.2: 一个 ASCII 码表

图片来源：维基共享资源

```
#include <iostream>
using namespace std;
int main(){
    int i = -1; // 定义一个int整型数据，并初始化为-1
    double d = 3.14159; // 定义一个double浮点型数据，并初始化为3.14159
    char c = '0'; // 定义一个char字符型数据，并初始化为'0'，其ASCII码为48
    return 0;
}
```

这三个语句的结构非常相似，我们可以把它们归纳成一个统一的形式：

<类型关键字> <名字> = <值>;

接下来我们分别介绍这三个部分。

## 类型关键字（Keyword）

C++ 为不同类型提供了丰富的关键字。

对于整型来说，`int` 是最基本的关键字。我们可以通过增加前缀的方式来将 `int` 型改为其它整型。

- 改变符号性：我们可以增加 `signed`（有符号）前缀，允许它表示负数；或者增加 `unsigned`（无符号）前缀，禁止它表示负数。如果不加前缀，将默认允许表示负数。因此，如果没有禁止负数的需要，无需加前缀 `unsigned`。
- 改变“容量”：我们可以增加 `short`, `long` 或者 `long long` 前缀来改变它的容量。简便起见，如果有这三个前缀之一，可以直接省略 `int`。因此，我们可以直接用 `long long` 这样的关键字而不必用 `long long int`。

无论改变符号性还是改变容量，其目的都在于让我们的数据合乎需求。如果我们要统计某物的数量，那么这个数字肯定是非负的，于是我们可以用 `unsigned` 来限制它的取值范围；如果某物的数量可能很多，`int` 类型的容量存不下，那么我们就要考虑使用 `long long` 了。

对于浮点型来说，**double** 是最常用的关键字。另有 **float** 和 **long double**，它们的有效位数和容量也不同。简单来说，**float** 的有效位数和容量最少，**double** 居中，**long double** 最多。浮点型数据无法改变符号性，它们必须允许负数的表示。

对于字符型来说，**char** 是最常用的关键字<sup>15</sup>，它能表示 ASCII 的所有字符，但对非拉丁字母（比如中日韩文字及变体）无能为力。为了表示非拉丁字母，我们可以使用 **wchar\_t** 类型。

其实我们在实际编程时最常使用的关键字是 **int**, **double**, **char**。而其它的关键字，如 **unsigned** 或 **long double** 往往是我们有特殊需求时才会去考虑使用的。简便起见，我们在这一章只使用这三种基本类型。它们足够应付我们的需要了。

## 标识符（名字）（Identifier）

在定义数据时，我们需要给它一个名字，以便日后使用。这个名字就是一种标识符。

在 C++ 中，标识符指代的概念甚广，包含一切关键字，以及我们定义的所有名字。在遵守下述要求的前提下，我们可以起各种千奇百怪的名字：

1. 一个标识符只能由英文字母 A-Z 和 a-z、数字 0-9 以及下划线 `_` 构成。<sup>16</sup>
2. 标识符的首字母不能是数字，但可以是字母和下划线。<sup>17</sup>
3. 标识符对大小写敏感。这意味着，即便两个标识符只有大小写上的区别，它们也是不相同的。
4. C++ 中的关键字均被保护<sup>18</sup>，你不能定义一个与关键字相同的名字。
5. 标识符的长度可能受到系统或开发环境的限制，不能太长。<sup>19</sup>

比如我在上述代码中起了三个名字：`i`, `d`, `c`，这些名字都是允许的。

不只是数据名。以后我们可能需要起函数名、结构名和类名，这些命名方式全部应该按照这样的标准来起名。举几个例子：“`lhs`”, “`it_1`” 和 “`__max`” 都是可以使用的名字，而 “`12a`”, “张三” 和 “`new`” 都是不能使用的名字（`new` 是 C++ 中的关键字）。

## 初始化（Initialization）

一个数据存储一个值。整型数据存储整数值，浮点型数据存储小数值，字符型数据存储字符值（对于 **char** 型来说，它存储的值可以用对应的 ASCII 码值来表示。）如果你不为它指定一个值的话，它就会存储一个不确定的值<sup>20</sup>。使用“不确定的值”是危险的，我们要尽量避免。

初始化是一种“在定义的同时就给定值”的方法。在本例中，我们使用=加一个值的形式来实现初始化。

```
unsigned long long ull = 4294967296;
// 定义一个unsigned long long 整型数据，并初始化为 4294967296
long double ld = 3.141592653590;
// 定义一个long double 浮点型数据，并初始化为 3.141592653590
```

<sup>15</sup>C++ 还规定了两个类型：**signed char** 和 **unsigned char**。请注意，它们和 **char** 是三种互不相同的类型。这一点与 **int** 等于 **signed int** 的情况截然不同。读者尚无必要纠结其中细节，精讲篇会予以介绍。

<sup>16</sup>这是不完全的，实际上还可以用其它具有 XID\_Continue 属性的字符。本书无意在这个问题上走得太远，也不打算使用这种字符来作为标识符。感兴趣的读者可以参阅[Properties for Lexical Classes for Identifiers-Unicode Technical Reports](#)获取更多信息。

<sup>17</sup>有一类特殊的反例，参见[用户定义字面量-cppreference](#)。这种情况不在我们的讨论范围之内。

<sup>18</sup>许多代码编辑器会对关键字做特殊着色处理。本书将关键字一律以蓝色加粗来标明（偶有不宜着色者除外）。

<sup>19</sup>C++ 标准并未对标识符的长度作明确规定，实际编程中也几乎不会有长度超限的情况，所以本条不那么重要。

<sup>20</sup>严格说来，只有局部变量才会如此。全局变量如果不加指定，将会自动初始化为零。

以上初始化语法也叫作**直接初始化 (Direct Initialization)**。

初始化的语法不只这一种。在 C++11 以后的语言标准中，我更推荐使用**统一初始化 (Uniform Initialization)**<sup>21</sup>的方法：

```
<类型关键字> <名字> = {值}; // 统一初始化语法
<类型关键字> <名字> {值}; // 另一种统一初始化语法
```

统一初始化的语法更强大，在进行初始化时会进行检测，避免范围缩限等问题。

举个例子，某台电脑的 **short** 型变量只能容纳-32768 到 32767（包含两端）之间的整数<sup>22</sup>。那么这样的代码是可以通过编译并且运行的：

```
short s1 = 40000; // 定义 s1 并直接初始化为 40000
cout << s1; // 输出 s1 的值
```

这个输出的结果可能是 -25536。同时，有些编译器还可能给出如下警告信息：

```
warning: overflow in conversion from ‘int’ to ‘short int’ changes value
from ‘40000’ to ‘-25536’ [-Woverflow]
```

在这里，给一个不能容纳 40000 的数据存 40000 的数值本来就是违规操作。警告信息 (Warning) 虽然是一个提醒，但仍然不会阻止你运行这个程序并得到错误结果，更遑论一些老旧编译器甚至不会给你警告信息。统一初始化则不然，它会直接以报告错误 (Error) 的方式阻止你运行程序，你必须改正这个违规操作才行。

```
short s2 = {40000}; // 定义 s2 并统一初始化为 40000，这种方法是禁止的。
```

统一初始化还有许多其它的优点，这也是我推荐使用统一初始化的理由。

## 实操：让编译器充当计算器

有了数据之后，我们可以拿它们来做一些简单的计算了。我们希望定义一些数据，让它们进行简单的四则运算。我们在开始动工之前，需要粗略规划一下我们要做的事情。我们的大致步骤是：

1. 定义几个数据，并初始化。考虑到我们可能会计算小数，我们应该用浮点型的 **double** 类型。
2. 把我们想算的式子写到代码的输出语句中。比如说，我想计算  $(a+b*c)/d$ ，那么我应该把 `cout <<` 后面改成我想输出的这个式子。
3. 编译、运行，看下输出的结果。

现在我们完成第一步，定义数据。从我们的式子中可以看出，这里需要 4 个浮点型变量，那么我就定义 4 个浮点型变量。同时我还需要给出 4 个初始值。

```
double a = {1.5};
double b = {0.3};
double c {0.5}; // 这也是一种统一初始化语法，省略了 =
double d {4.5};
```

其实 C++ 还支持另一种简便的语法。如果要定义的量是同类型的，那么我们可以用单个类型关键字，将变量名隔开来定义。

```
double a {1.5}, b {0.3}, c {0.5}, d {4.5}; // 定义并初始化
```

<sup>21</sup>有关“统一初始化”这个词，实在让人费解。C++ 标准中没有查到过这个词，cppreference 中也没有查到这个词。但这个词却在网络中普遍存在，十分常用。笔者无法在概念问题上耗费太多时间，因此请读者留意，本书很有可能会在那里犯错。

<sup>22</sup>这个范围可能因电脑而异，如果读者要自己尝试，应以自己电脑的实际情况为准。

注意，句末用的是分号，而句中用的是逗号。这样就不用写 4 次 `double` 了，省时省力。

接下来是第二步，我们改一下输出代码，让它输出我们需要的内容。

```
cout << (a + b * c) / d; // 输出 (a+b*c)/d 的值
```

这段代码不能单独存在，现在让我们“借鉴”一下 `Hello_World.cpp` 的代码，把它改成这样：

代码 1.2: `calc.cpp`

```
// 这是一个简单的C++程序，用于简易的数值计算
#include <iostream> // 标准输入输出头文件
using namespace std; // 使用 std 命名空间
int main(){ // 主函数，程序执行始于此
    double a {1.5}, b {0.3}, c {0.5}, d {4.5}; // 定义并初始化
    cout << (a + b * c) / d; // 输出 (a+b*c)/d 的值
    return 0;
}
```

最后的输出结果是一个近似值 `0.366667`，与实际计算器算出的结果相近。

你还可以修改代码中的初始值，或者是把输出的式子改成别的式子，观察一下结果。

## 1.4 运算符与类型

在上一节的末尾，我们实现了一个简单的基于浮点数的计算器。这一节我想走得更远，增加一些数据类型和新的运算，看看会不会有什么新变化。

### 整数的模运算

模运算 (Modulo)，俗称取余，简记为 mod，就是求两个数相除得到的余数。例如， $5 \div 2 = 2 \dots 1$ ，这里的余数就是 1，所以  $5 \bmod 2 = 1$ ；再例如， $9 \div 3 = 3 \dots 0$ ，这里的余数就是 0，所以  $9 \bmod 3 = 0$ 。

一般意义上讲，只有整数才有带余除法，所以我们在计算时应该使用整型，比如 `int`。如果要计算某两个数 `a` 与 `b` 的模值，我们可以用如下的代码来实现：

```
int a {1584}, b {7}; // 定义 a, b 并初始化
cout << a % b; // 求 a mod b 的值
```

这里使用的百分号 (%) 可能会让人费解。其实在 C++ 中，% 并不是百分号，而是取模运算符，含义是左边整数除以右边整数得到的余数（模值）。在各类编程语言中，这种“记号含义与传统含义不同”的现象比比皆是，以下是 C++ 当中的部分例子（读者无需现在就掌握，但日后要留心）：

- `^` 不是幂指数的含义，而是位异或运算。<sup>23</sup>
- `<<` 和 `>>` 既不是书名号，也不是数学意义上的“远小于”和“远大于”，而是移位运算符。
- `[]` 和 `{}` 不是数学上的“中括号”和“大括号”，它们分别有另外的含义。数学意义上的括号统一使用 `()` 嵌套来实现。
- `=` 是赋值运算符，而传统意义上相等性的判断应当用 `==`。
- .....

<sup>23</sup>关于位异或和移位运算符，参见附录 C.2 布尔代数基础。

要计算  $1584 \% 7$  的值，也可以跳过定义数据这一步，直接在输出语句中使用字面量<sup>24</sup>完成。

```
cout << 1584 % 7; //直接求 1584 mod 7, 不必定义a和b
```

这种写法更加简便、直观，不需要定义数据。但也请注意，它无法支持程序层面的交互（如，用户输入两个数并计算。我们总不能要求用户来亲自修改代码。关于程序输入，我们会在第二章介绍）。

## 字面量的类型

C++ 会自动识别字面量的类型，比如前面介绍过的这个例子（编译器的报错信息以注释形式附于其后）：

```
short s2 = {40000};
//error: narrowing conversion of '40000' from 'int'
//to 'short int' [-Wnarrowing]
```

编译器的报错信息是很重要的，它能把出错的关键信息告诉我们，帮助我们有针对性地查找和解决问题。这个报错信息是“40000 在从 `int` 到 `short int` 的过程中发生了收缩转换”，意思就是“原本 40000 是 `int` 类型的，完全装得下；但是你要往 `short` 里面塞，那就装不下了”。

于是从报错信息中我们能看出，40000 这个值被编译器识别成 `int` 型。

再来个类似的例子：

```
int i {2147483648};
//error: narrowing conversion of '2147483648' from 'long long int'
//to 'int' [-Wnarrowing]
```

这个报错信息是“2147483648 在从 `long long int`<sup>25</sup> 到 `int` 的过程中发生了收缩转换”，意思就是“原本的 `long long` 类型完全装得下，但是 `int` 类型装不下”。换言之，字面量 2147483648 被编译器识别成了 `long long` 型。

总得说来，如无前缀或后缀<sup>26</sup>，代码中的浮点字面量统一识别成 `double` 型。代码中的整数字面量会按照 `int`, `long`, `long long` 的顺序，找到能容纳它的类型，作为这个字面量的类型；而字符串字面量会把 ASCII 字符统一识别成 `char` 型<sup>27</sup>。

字面量的实际知识远比我们介绍的要多。如果读者有意进一步了解相关知识，可以查阅[字面量-表达式-cppreference](#)。

## 1 和 1.0 的区别是什么？

我们在写代码时常常会遇到 1.0 这样的写法。你可能会好奇，为什么不写成 1 呢？究其原因，在于 1.0 是一个浮点字面量，而 1 则是一个整数字面量。它们的类型并不相同，一个是 `double`，另一个则是 `int`。

C++ 中还有更简洁的语法，就是用 1. 和 .1 来取代 1.0 和 0.1 这样的浮点字面量。比如说

```
cout << 1. / .1; //等价于 cout << 1.0 / 0.1;
```

<sup>24</sup>字面量 (Literal)，是一个值在代码中不具名的文本表示。比如 1584，它没有“名字”，在代码中被写出来就是 1584，它的值也是 1584。与之相对的是具名数据，它们经过定义，就有了名字。

<sup>25</sup>这点可能因电脑的环境而异，比如有些电脑的 `long` 类型就可以容纳这个数，那么报错信息应当是 `long int`；而另一些则不能容纳这个数，报错信息应当是 `long long int`。

<sup>26</sup>字面量可加前缀或后缀来指定优先使用的类型，第二章会对此进行介绍。

<sup>27</sup>关于宽字符，我们会在精讲篇中再谈。

## 整数除法问题

让我们来看一下这段代码：

```
cout << 8 / 3; // 计算 8/3 的值
```

猜一下它的运行结果。如果你手头有编译条件，可以跑一下代码看看（记得加上那一堆必需代码）。

如果你猜的结果是 2.66667 或者是类似的近似结果，那么恭喜你，这是初学者常犯的错误！

整数除法与浮点数除法有所不同。如果两数不能整除，那么结果就会被“截尾”<sup>28</sup>。如果要避免截尾，获得较精确的数据，应该使用符点型来操作。

```
cout << 8. / 3.; // 计算 8.0/3.0 的值
```

那如果我要计算两个具名整型数据的除法呢？比如在 a 和 b 都是整型的情况下计算 a/b？这时我们可以使用第二章中将会介绍的类型转换的方法。

这里有一种比较简单的类型转换方法：

```
cout << 1. * a / b; // 计算 a/b 的浮点数结果
```

但是请注意，下面的写法是不行的：

```
cout << a / b * 1.; // 计算 a/b 的浮点数结果，但事与愿违
```

关于第一种方法为什么可行，第二种为什么不可行，我们会在第二章中探讨。

总而言之，类型不同，可能会让结果出现一些意想不到的差异。因此类型是很重要的数据特征，日后编程时我们需要多加留意。

## 1.5 `sizeof` 与内存空间

我们在之前总是反复提及，整型与整型之间是不同的，浮点型与浮点型之间也是不同的。对于整型来说，有符号与无符号是一个区别，而“容量”是另一个区别。如何理解这里的“容量”呢？这就需要我们深入到“内存”中去，一探究竟。

### 什么是内存？

张三正在用电脑打单机游戏，这时突然停电了。好不容易等到来电，张三心急如焚地打开电脑，刚准备继续玩，突然发现自己忘了存档，当前游戏进度还是昨天的进度。这让他非常不爽。

那么问题来了，明明确程序中都有我们当前游戏的进度，为什么我们还要通过“存档”来把这些内容保存下来呢？这就涉及到内存和外存的区别了。

**非易失型存储器 (Non-Volatile Memory, NVM)**，俗称外存，这种存储器内的信息不会随着电脑关机、供电中断等因素而丢失。但是，外存信息的访问速度慢，可能导致程序运行卡顿。张三的游戏存档就储存在外存当中。

**易失型存储器 (Volatile Memory, VM)**，俗称内存，这种存储器内的信息会随着电脑关机、供电中断等因素消失。不过，内存信息的访问速度更快，程序运行更加流畅，所以计算机程序主要都是在内存当中运行的，偶尔才会与外存进行数据交换。在张三打游戏的过程中，游戏数据就存储于内存当中，只有通过存档才能把信息保存至外存。

正因如此，张三今天的游戏数据（储存于内存）丢失了，但昨天的存档（储存于外存）依然健在。

同样，我们写出来的程序在运行时也会在内存当中运转。我们定义的数据，也都保存在内存当中。

<sup>28</sup> 截尾 (Truncation)，广义上指对小数点后数字个数的限制。在这里指的是直接截掉小数点后的部分，比如 1.618 → 1 和 -2.718 → -2。这种方法与四舍五入和单纯上/下取整都不同，可以把它理解成“对正数向下取整，对负数向上取整”。

## 内存如何存储数据？

存储器的内部结构很像是一大群排列紧密、有序的房屋，彼此都是相同的模样。每个房屋有 8 个同款的房间，每个房间都存储着 0 或 1，看上去单调而乏味，了无新意。但就是这种看似无趣的 0 和 1 的排列组合，却构成了我们眼里丰富多彩的信息。一部时长一百分钟的高清电影，大小 7.07GB，换算过来是 76 亿个这样的房屋，由 607 亿个这样的 0 和 1 的排列组合而成。<sup>29</sup>

前文提到，我们可以用编码的方式来将多样的信息转换成 10 种阿拉伯数字的排列组合，又可以通过解码的方式将这些数据转换回去。而在存储器中，多样的信息都会被编码成 0 和 1 的排列组合。我们把每一位这样的 0 或 1 称为一个比特（Binary digit, bit）。

比特，如同生命中的细胞，化学中的原子，建成大厦的砖块和社会中的每个人，是构成信息最基本、最微小的单元。无数细胞构成了绚丽多彩的生命，无数原子构成了气象万千的世界，而无数比特则构成了我们眼中目不暇接的信息。

不过，存储器中是不允许“单细胞生物”或者“单原子分子”存在的。在存储器中，每 8 个比特构成一个字节（Byte）<sup>30</sup>。字节是度量信息的单位，也是可寻址<sup>31</sup>的最小单元。

回到我们刚才的比喻。存储器的内部结构是排列有序的房屋。每个房屋代表一个字节，它是相对独立的最小可寻址单元。而每个房屋有 8 个房间，各自存储一比特的信息。图 1.3 可以帮助我们更直观地理解这种关系。

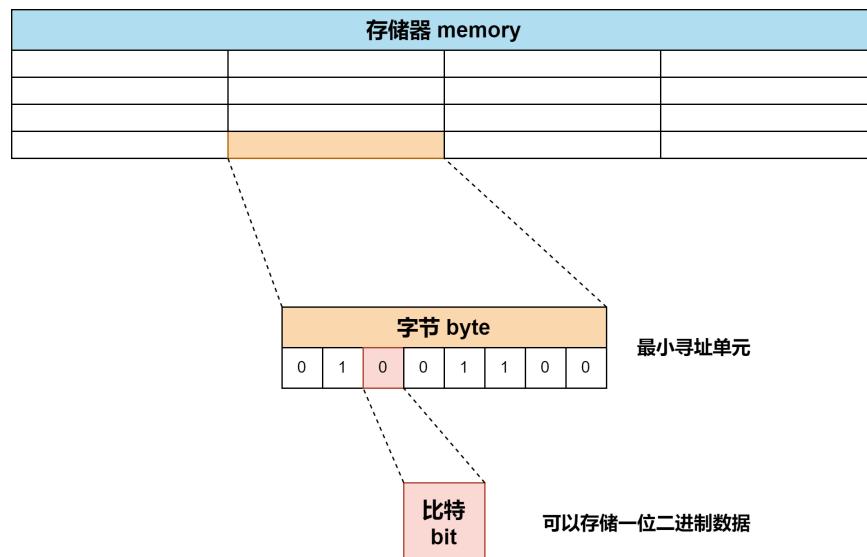


图 1.3: 存储器、字节与比特

## 信息与容量

香农<sup>32</sup>是最早把“比特”的概念引入到信息技术中的人。在论文<sup>33</sup>的第一页，他就提出：N 个比特可以表示  $2^N$  种状态。<sup>34</sup>

<sup>29</sup>严格说来，这部电影存储于外存。但内存与外存都有这样的特征，所以借来阐述并不为过。

<sup>30</sup>历史上，每个字节包含多少个比特没有特定标准，因而在一定时期是混乱的。如今的标准规定每个字节包含八个比特，但可能有极少数例外。

<sup>31</sup>关于内存地址，我们会在第五章讲解。

<sup>32</sup>克劳德·艾尔伍德·香农（Claude Elwood Shannon），美国数学家、电子工程师、计算机科学家和密码学家，被誉为“信息论之父”。

<sup>33</sup>A Mathematical Theory of Communication, 于 1948 年发表于《贝尔实验室技术期刊》。这篇论文奠定了现代信息论的基础，被《科学美国人》誉为“信息时代的《大宪章》”。

<sup>34</sup>原文：A device with two stable positions, such as a relay or a flip-flop circuit, can store one bit of information. N such devices can store N bits, since the total number of possible states is  $2^N$  and  $\log_2 2^N = N$ .

我们可以把一个数据的值看作一个状态（State）。当这个数据是 20 的时候，它的状态就是 20；当这个数据的值是 13 的时候，它的状态就发生了改变，变为 13。可以试想，如果我们有 1 个字节的数据，那么这 8 个比特一共可以表示的状态数量就是  $2^8 = 256$  个。换言之，它只能表示最多 256 种状态，也就只能编码 256 种信息。

C/C++ 中的 `char` 数据占用的内存空间是一个字节，所以它能表示 256 种信息。而 ASCII 码只有 128 种信息，所以用一个 `char` 足够表示一个 ASCII 字符。假如我们定义了一个 `char` 数据并初始化为 '`A`'，那么它在内存中的 8 个比特就有了某个特定的排布方式（假设排布方式是 `0100 0001`）。下一次，我给它另一个值，比如 '`Z`'。改变了它的值，也就改变了它的状态，于是这 8 个比特的排布方式也就改变了（假设排布方式改为了 `0101 1010`）。但是无论怎么修改它的值，这 8 个比特的信息永远都在 256 种可能性中打转，绝对不会出现第 257 种排布方式。于是我们可以说，它的“容量”就是 256。

在大多数电脑上，`int` 类型的数据占用的内存空间是 4 字节<sup>35</sup>，也即 32 比特，所以它能表示  $2^{32} = 4'294'967'296$  种状态。所有这些状态被用来给数字编码。对于 `int` 类型来说，它可以表示从  $-2^{147'483'648}$  到  $2^{147'483'647}$  的全部整数（含两端）；而 `unsigned` 类型可以表示从 0 到  $4'294'967'295$  的全部整数（含两端）。于是我们可以说，它的“容量”就是  $4'294'967'296$ 。

值得注意的是，虽然 `signed int`（亦即 `int`）和 `unsigned int`（亦即 `unsigned`）占用的内存空间大小相等，但它们能表示的数据范围是不相同的！这是一个“我们能够编码多少数据”和“我们编码了什么数据”的问题。对于 `int` 和 `unsigned` 来说，它们都只能编码  $4'294'967'296$  个数据，但因为它们编码的数据不尽相同，于是它们能表示的数据范围也就有所差异（但数据宽围的“宽度”是相等的）。我们会在精讲篇中更细致地谈论这个问题。

那么如果你想要表示比  $4'294'967'295$  还要大的数字呢？我们就要选择能容纳它的类型，比如 `long`<sup>36</sup> 或者 `long long`，甚至是 `unsigned long long`。总得说来，占据内存空间越多的类型，它能表示的数据范围也就越宽。

## `sizeof` 运算符

在不同的电脑上，同一数据类型可能占据的字节数并不相同，这就给我们造成了困扰。如何知道自己电脑上每个类型占据多少内存空间呢？我们可以用 `sizeof` 运算符来搞定这个问题。

`sizeof` 是一个单目运算符<sup>37</sup>，可以接收一个类型或数据信息，并求出其内存占用。它的语法是

```
sizeof (<类型或数据>); // 对类型求 size 必须套括号
sizeof <数据>; // 如果只对单个数据求 size，可以不套括号
```

这只是单纯的求值。如果要输出我们求出来的值，我们就需要在前面加上 `cout <<`。

```
cout << sizeof (char); // 求 char 类型的内存占用
```

这样它就会输出 `char` 类型的内存空间大小。这个值在任何电脑上都是 1。

我们还可以让它求出某个数据的内存空间占用，比如

```
cout << sizeof 2.71828; // 求双精度浮点数 2.71828 的内存占用
```

在 Coliru 上，它的输出结果是 8。

还可以输出我们定义的数据的内存空间大小，比如

```
unsigned long long ull = {10}; // 定义一个 unsigned long long 数据并初始化为 10
cout << sizeof (ull); // 求 ull 的内存占用，可以不加括号
```

在 Coliru 上，它的输出结果是 8。

<sup>35</sup>C++ 标准规定，`int` 类型的最低内存占用为 2 字节（16 比特），但现在的大多数电脑都使用 4 字节。

<sup>36</sup>在很多电脑上，`long` 类型和 `int` 类型一样占据 4 字节空间，这时就不能用 `long`。

<sup>37</sup>有关运算符的问题，我们将在第二章讲解。

## 编译时行为与运行时行为

`sizeof`也是一个运算符。不过它与我们之前见到的加减乘除和模运算不同，它不是在程序运行时进行计算的，而是早在编译时就已经计算好了的。

我们在第 1 节中讲过，从写完代码到运行程序可以粗略分为三个过程：编译、链接和运行。有些值在编译的时候就能确定下来，比如 `sizeof (a)`，无论它在内存中处于什么位置，也无论它的值是多少，它占据的内存空间的大小永远相同。（还记得吗，无论数据怎么变化，本质上只是若干个比特的排布方式发生了改变，但这个数据仍然占据同样尺寸的内存空间）

所以 `sizeof` 在编译时就已经计算出来了，运行时就无需再浪费时间去计算 `sizeof`；而四则运算和取模，它们必须要在运行时才能确定结果，所以编译时不会进行计算。

C++11 标准引入了 `constexpr` 语法，它扩展了编译时行为的范围。比如说

```
int a = {3}, b = {5};
int c = {a+b}; //a+b在运行时计算
```

假如使用 `constexpr` 呢？

```
constexpr int a {3}, b {5};
constexpr int c {a+b}; //a和b都是constexpr，所以a+b将在编译时计算
```

这说明，`constexpr` 改变了编译器处理数据的方式，使得一些数据可以在编译时就被计算出来。

本章只对编译时行为、运行时行为和 `constexpr` 作简要介绍，在精讲篇中我们还会深入讲解。

## 1.6 类与对象

C++ 语言脱胎于 C 语言。斯特劳斯特鲁普<sup>38</sup>最初选择了 C 语言，首先在 C 的基础上发展出了 C with Classes。后来 C++ 不断扩容并增加新功能，直至 C++98 标准发布，C++ 语言的基本功能已经成形。之后的 C++11 又是一个划时代的更新，不过这些就离我现在想要谈的事情比较远了。

迈耶斯<sup>39</sup>在他的著作《Effective C++》中，将 C++ 语言视为四种“子语言”的统合体<sup>40</sup>：

- **C**: 这是脱胎于 C 语言的部分，它继承和发展了 C 语言中的部分功能。所以当有人问起“如果没学过 C 语言，是不是要先学 C 再学 C++”的时候，我们的回答都是“不必如此”！在 C++ 的起步阶段中你学习的内容已经基本涵盖 C 语言了。
- **面向对象的 C**: 这里有了比 C 语言中 `struct` 丰富得多的功能，它也是 C with Classes 最初区别于 C 的特征。
- **泛型 C**: 这里提供了泛型编程和元编程的支持，使 C++ 成为一门更为强大的语言。它们是 C++ 与 C with Classes 的重要区别之一。
- **标准模版库 (STL)**: STL 由斯捷潘诺夫<sup>41</sup>设计实现。相比于单纯的泛型编程，它有一套相对独立的逻辑。STL 更多地支持对“现成”数据结构和算法的使用，其优点是方便、高效、简洁。

这些内容都会在本书的泛讲篇或精讲篇有所涉及。但本书并非进阶教程，读者若有意深入挖掘，还应选择合适的进阶教材阅读。

<sup>38</sup>比雅尼·斯特劳斯特鲁普 (Bjarne Stroustrup)，丹麦计算机科学家。他发明并发展了 C++ 语言，以其对 C++ 语言的卓著贡献而享誉世界。如果读者有意了解关于他和 C++ 的更多信息，可以访问[Stroustrup 的个人主页](#)。

<sup>39</sup>斯科特·道格拉斯·迈耶斯 (Scott Douglas Meyers)，美国作家、软件顾问和 C++ 语言程序设计专家，以其所著 *Effective C++* 系列书而闻名。

<sup>40</sup>这只是迈耶斯的观点，笔者认为其合理而采纳至此。不同的专家可能对此有不同的理解。

<sup>41</sup>亚历山大·亚历山德罗维奇·斯捷潘诺夫 (Alexander Alexandrovich Stepanov)，俄罗斯裔美国计算机科学家，以其提倡泛型编程与设计实现了 C++ 语言中的标准模版库而闻名。

从 C++ 的发展史来看，面向对象编程（Object-oriented programming, OOP）是 C++ 相对于 C 跨出的第一步。面向对象的概念广泛用于 C++ 的语法中，以至于我们往往习焉不察。

我本来不打算在第一章就介绍面向对象的概念，但我很快就发现那很难。如果不了解什么是面向对象，也不清楚相关的基本概念，后面几章的知识依然都能学（别忘了，C 语言可没有面向对象，那些 C 有关的语法也都不依托这方面的知识）；然而，这会在“C”部分与“面向对象的 C”部分的过渡处造成很大的障碍（尤其对于初学者来说）。正如迈耶斯所说，它们几乎是两个子语言，为此你需要一定的适应期，并且回顾很多以往的知识，才能真正理解它。

出于上述考虑，我决定在第一章就介绍面向对象，并且在后面的章节中也会时常提起它（甚至把它和一部分 C 内容串讲）。我希望通过这种方式，读者能够真正把“一门语言”学成“一门语言”，而不是“四门语言”——如果可以的话，这也是我对本书的一点期许。

## 什么是类？什么是对象？

类与对象是一对概念，不得不并举。

**类（Class）**是一个抽象的概念，它规定它的所有对象都有一些属性和功能。而**对象（Object）**<sup>42</sup>则是一个个具体的实现。举例来说，“哈士奇”是一个抽象的概念（类），它规定了其对象的一些属性，比如“身高”、“体重”，还有其对象的一些功能，比如“拆家”。而“我家的哈士奇”和“张三家的哈士奇”是两个实体（对象），它们衍生自“哈士奇”这个概念。

同一个类的不同对象都有这些属性，但属性的具体值是因对象而异的。我家的哈士奇的体重是 27 公斤，而张三家哈士奇的体重是 20 公斤。属性不同，会导致它们在运行一些功能时有不同的表现。比如，有个沙发的承重是 25 公斤，那么我家哈士奇拆沙发的效率当然就要好于张三家的哈士奇。

我们经常容易把类与对象的关系和另外两组关系搞混。所以不妨让我们来做一下辨析：

- “狗”与“哈士奇”的关系不是类与对象关系。它们两个都是抽象的概念，只不过“哈士奇”在“狗”的基础上进一步细化了属性和功能。比如说，“拆家”不是所有狗的特征，它不是“狗”的功能；而“哈士奇”在“狗”这个概念的基础上增加了“拆家”的功能。然而说到底，“哈士奇”依然是一个概念，而非实体。<sup>43</sup>
- “猫”与“我家哈士奇”的关系不是类与对象关系。“猫”的确是一个类，“我家哈士奇”也的确是一个对象，但它们完全八竿子打不着。我家哈士奇根本就不是猫。
- “张三”与“张三的心脏”的关系不是类与对象关系。“张三”确实是由各个器官构成的，然而张三是一个实体，它们之间的关系应当是整体与部分，而不是类与对象。
- “人”与“张三的心脏”的关系也不是类与对象关系。我们不能说“张三的心脏”就是一个人。
- “人”与“张三”的关系是类与对象关系。张三是一个人。

可以看出，类与对象的关系可以用“a 是一个 A”的语句来检验和阐述，比如：秦始皇是一个皇帝；上海是一座城市；银河系是一个“有生命存在的星系”。

## C++ 中的类与对象

面向对象的概念并非 C++ 独有；不过本书只介绍 C++ 中的面向对象。

在 C++ 中，类型（Type）和类（Class）很多时候是等价概念。<sup>44</sup>假设我定义了一个 `int` 型数据 `i`，那么它们两者之间的关系是不是类与对象呢？

<sup>42</sup>“对象”这个译名容易引起困惑；相比之下，英文原名 object 和台湾译名“物件”就显得易懂些。

<sup>43</sup>在第九章中我们会讲到，这是两个类的继承关系。

<sup>44</sup>在泛型编程中，我们一般将其视为等价概念，使用 `typename` 和 `class` 关键字的效果是相同的。不过，在另一些语境下，`class` 是比 `type` 更为广泛的概念。

我们可以用“a 是一个 A”的语句来检验这个关系。就像我们反复强调的那样，i 是一个 int 型的数据，于是“i 是一个 int”的判断成立，它们是类与对象的关系。

我们可以在同一个类之下定义很多对象。

```
double a {1.1}, b {2.2}, c {3.3}; // 在 double 类之下定义三个对象 a,b,c
```

这里的 **double** 就是一个抽象的概念，而 a, b, c 就是这一概念下创造出来的实体。

C++ 中还有一个更好的例子，就是我们一直在用但是还没有讲的 cout。这是一个 ostream 类的外部对象<sup>45</sup>。ostream 类重载了 << 运算符<sup>46</sup>，使得我们可以方便地输出很多基本类型的对象的值。

ostream 只是一个类，它提供了诸多输出方面的功能支持，但是倘若没有一个实体，这些功能都是空中楼阁，无从实现。为此，C++ 在 std 命名空间中定义了 cout 作为标准输出对象，所以我们才可以使用这个对象来实现基本类型值的输出。

```
namespace std{
    extern std::ostream cout;
};
```

这个对象定义在头文件 iostream 中，所以我们必须要在每个完整的程序代码之前加上

```
#include <iostream> // 包含头文件 iostream
```

方可使用它。

C++ 还定义了标准输入 cin，也在头文件 iostream 当中。我们会在下一节中使用它。

---

<sup>45</sup>关于外部对象等概念，我们将在第七章中介绍。

<sup>46</sup>关于运算符重载，我们将在第八章中介绍。

# 第二章 数据的基本操作

在第一章，我们通过一些简单的代码，对 C++ 语言有了最基本的认识。我们还了解了数据、信息、内存和面向对象的相关常识。在后面的章节中，我们将更多关注于具体的知识。

本章我们先来介绍数据。按照分类标准的不同，我们可以把 C++ 语言中的数据分成不同的类别：

- 如果按照数据的类型来分类，数据可以分为整型数据、浮点型数据等等。
- 如果按照是否具名来分类，数据可以分为不具名量<sup>1</sup> 和变量。前者没有名字，后者有名字。
- 如果按照是否可以改变来分类，数据可以分为常量和可变量。前者不可改变，后者可以改变。
- 如果按照是否可取地址来分类，数据可以分为左值和右值。<sup>2</sup>
- .....

光是分类还不够。我们还要学会如何更好地使用数据。在第一章中我们通过简单的代码实现了计算器的功能，但如果我们仅仅止步于此，那还远远没有达到 C++ 能够支持的水平。在这一章我们会学习更多有关运算符的知识，以此扩展代码的可能性。

数据类型的差异可能会为我们造成一些困扰。在实际编程时，我们可能需要在不同数据类型之间作转换。本章我们也会谈及数据类型转换的基础语法。

C++ 的世界非常丰富，但一切都要从数据开始。

## 2.1 赋值语句与常量

在此之前我们定义了许多数据，但是只进行了初始化，并未在定义之后修改过它们的值。实际上我们之前定义过的那些数据（它们也被称为变量（Variable））都可以通过赋值、输入、运算等方式来修改其值。这里先来讲一下赋值语句，然后是常量；输入和运算将在之后讲解。

### 基本赋值语句

最基本的赋值语句的结构是这样的：

```
<变量名> = <值>; //用字面量为其赋值  
<变量名> = <表达式>; //用其它数据或表达式为其赋值
```

如果你愿意，也可以将初始化看作一个赋值过程<sup>3</sup>：但我绝不会这样讲。

在基本赋值语句中出现的 = 叫做赋值运算符（Assignment operator），它的含义是：将右边的数的值赋给左边，从而改变左边的值。

<sup>1</sup>我们先前提及的“字面量”正是不具名量的一种。

<sup>2</sup>有关左值和右值的问题，我们在精讲篇中讲解。

<sup>3</sup>对于内置类型，我们看不出细节来；但对于自定义类型来说，初始化和赋值是不同的。初始化时会调用构造函数，而赋值时会调用赋值运算符。我们在第八章中会细讲。

赋值运算符的用途很广泛，C++ 中的许多数值改变都要通过赋值来实现。以下是一些常见的例子：

```
int n = {3}; // 定义n并初始化为3
n = n * n; // 这是在做什么？
cout << n; // 输出会是什么样的？
```

初学者可能会对  $n = n \times n$  这样的表达感到困惑。但是不要忘记，这里是编程，而不是数学！

$n=n*n$  这个语句可以由赋值运算符分为两部分，左边是  $n$ ，右边是  $n*n$ 。按照赋值运算符的含义，它的功能是将右边的数的值 ( $n*n$ ) 赋给左边，从而改变左边数 ( $n$ ) 的值。也就是说，它的含义是在把  $n^2$  的值赋给  $n$ ，于是输出就变成了 9。

赋值运算符和数学上的等号<sup>4</sup>不同的地方还在于，它是不满足交换律的。比如说， $a=b$  的含义是把  $b$  的值赋给  $a$ ，而  $b=a$  的含义是把  $a$  的值赋给  $b$ 。再比如，如果我们把刚才的代码改成这样，它是不能通过编译的：

```
int n = {3}; // 定义n并初始化为3
n * n = n;
//error: lvalue required as left operand of assignment
cout << n;
```

我已将编译器的报错信息以注释的方式添加到代码中。它的意思是：赋值运算符的左操作数需要是一个左值。我们不需要管什么是“左值”，我们只需要知道，编译器并不允许我们把  $n$  的值赋给  $n*n$ 。

## 常量

任何一个变量（广义地讲，对象）在被定义之后都可以修改，这固然很方便，但也可能会遭遇一些问题。

圆周率的近似值是 3.14，我们希望定义一个变量  $\text{Pi}$ ，它不会被我们自己，或是别的人随便改动；围棋棋盘上有  $19*19$  个可放子的交叉点，这是恒定不变的，我们也希望不会有人乱改。

但是传统意义上的变量无法禁止修改。C 语言使用宏<sup>5</sup>来定义一个标识符，并通过预处理将其转换为字面量（字面量无法被修改）来实现禁止修改的效果。而在 C++ 中，我们有更好的方法：使用 **const** 限定符，将变量限定为常量（Constant）。<sup>6</sup>

定义一个常量的语法是

```
const <类型> <名字> = <初始化>; //const可以在类型关键字前
<类型> const <名字> = <初始化>; //const在类型关键字后，效果相同
```

这两种方式均正确，读者可以选择自己喜欢的方式来定义。

我们可以定义一个 **double** 型的常量  $\text{Pi}$  来表示圆周率的近似值，再定义一个 **int** 型的常量  $\text{Grid}$  来表示围棋棋盘的边长。那么我们可以这样写：

```
const double Pi = {3.14}; // 定义一个double常量Pi
const int Grid {19}; // 定义一个int常量Grid
```

如果你试图在之后的代码中修改常量，那么编译器将报告这样的错误：

```
Pi = 3.1; //试图修改常量Pi的值
//error: assignment of read-only variable 'Pi'
```

<sup>4</sup>数学上的等号在 C++ 中可以用相等性运算符  $==$  来表示。我们会在后面讲解。

<sup>5</sup>宏（Macro），是一种批量转换特定内容的方法。它的功能很像我们常用的“查找替换”。

<sup>6</sup>广义上讲，“常量”是“变量”的一部分。根据[Variable-Wikipedia](#)的阐释，凡在内存中占据存储空间的具名量都是变量。这点似与数学上的“变量”概念不同。本书于处理常量变量之关系处也很为难，故约定：在需要分清常量和变量的场合下，将它们视为不重合的概念；而在一般场合下，统称其为变量。当然，若以“对象”统称之，便没有歧义了。

这说明 `Pi` 是一个只读的数据，我们不能对它赋值来进行修改。

在 C++ 中，常量是一种一经定义就无法修改的量。这意味着，我们只能在定义它的同时就给它初始化，而不能把定义和初始化的步骤分开。<sup>7</sup> 变量则不同，我们可以不进行初始化，或者是在定义之后进行赋值。

```
int i; // 定义变量但不初始化，此时i的值未定
i = 2; // 赋值，这样i的值就确定下来了
const int i_c; // 定义常量但不初始化
//error: uninitialized 'const i_c' [-fpermissive]
```

这个报错信息的含义是：`i_c` 作为一个常量，没有被初始化。所以这是不允许的。

## 常量与字面量的区别

说到这里，你也许会感到困惑：如果我们用 `const` 常量就是为了防止数据被修改的话，那为什么不全部使用字面量呢？这里就来介绍一下常量相比于字面量的优点。

首先，常量可以用标识符来阐述其含义。比如用 `Grid`，这就比单纯放一个字面量 19 要容易理解得多，也便于后期我们回顾代码。

而且，常量容易统一修改。这里的“修改”不是上文中提到的那种赋值式的修改，而是直接修改定义语句中用来初始化的值。

我们在一开始设计围棋程序的时候可能没有想到，围棋棋盘的大小还可以有  $13 \times 13$  和  $9 \times 9$  两种简化版。如果咄咄逼人的甲方要求你改成 13 或者 9 的边长，而你又在你的上千行代码中写下了数百个 19，你可能会崩溃的。

而如果我们都用常量呢？我们可以不用写数百个 19，而是把它们都用 `Grid` 代替。这样一来，如果甲方要求你改变棋盘的大小，你就可以通过改变初始化值的方式来解决问题了。

```
const int Grid {13}; // 现在把19改成13
```

这样就很省事了。

常量还有一个优点，它可以使用变量来初始化。比如说，我现在换了个思路，不想再修改源代码了。我通过一个中间变量来接收输入，然后把这个输入的值作为棋盘大小，用来给常量初始化。这个过程对于初学者来说可能有点复杂，我把代码列在这里：

```
int g; // 定义一个临时变量g。因为通过输入来赋值，我们不必初始化
cin >> g; // 通过输入来改变g。输入多少，g就变为多少
const int Grid = {g}; // 用g的值来为Grid初始化，g的值取决于输入
```

这是一个相当有创造性的做法！在这里，`Grid` 不是预先决定的常量，而是可以通过变量来控制的常量。它仍然符合一经定义就无法修改的限制，同时又有了一定的灵活性。这种做法在将来的函数参数传递中非常常用，后面我们还会继续这个话题。

## 枚举常量简介

**枚举 (Enumeration)** 在 C++ 中的使用不像常量那么频繁，但它也很好用。在有些场合中，定义整型常量比较麻烦，而定义枚举很简单，这时我们可以用枚举来简化我们的代码。在另一些场合中，我们可以用枚举来方便地为一些标识符制定常量编码，并限制可能的取值范围。

具名的枚举可以视作一个自定义类型，而常量只能依托于已有的类型（无论是内置类型还是自定义类型），比如 `int`, `double` 之类。

定义一个枚举要用到 `enum` 关键字。定义枚举的最简单语法是：

---

<sup>7</sup>某种意义上，这也体现出初始化与赋值的区别。

```
enum { <标识符1>, <标识符2>, <标识符3>, ... }; //无枚举名
enum <枚举名> { <标识符1>, <标识符2>, ... }; //有枚举名
```

无枚举名的语法一般不用于自定义类型，而是纯粹当作整型常量。我们可以指定不同的标识符的值。

```
enum { MiniGrid = 9, SmallGrid = 13, NormalGrid = 19 };
//定义无枚举名的枚举，相当于整型常量
```

这样我们就可以像常量一样使用它们了。

有枚举名的语法当然也可以当作整型常量来看待，不过它支持的功能就更丰富了。

```
enum Color { Red, Blue, Green}; //定义有枚举名的枚举
Color sky_color = {Blue}; //定义Color类数据sky_color，其值为Blue
Color grass_color = {Green}; //定义Color类数据grass_color，其值为Green
```

这里的 `sky_color` 和 `grass_color` 就相当于 `Color` 类型的两个数据，或者也可以说是 `Color` 类的两个对象。

这里只对 `enum` 作简单介绍，而不要求读者掌握。我们会在第六章讲解关于枚举的更多细节。

## 2.2 基本数据类型

### 整型

在第一章我们已经谈过，整型就是只能表示整数的数据类型。

常用的整型变量一般有 `short`, `int`, `long`, `long long` 及其相应的 `unsigned` 版本<sup>8</sup>。某一类型在不同电脑上占用的内存空间大小可能是不同的，而内存空间的多少又和数据表示范围紧密相关，所以我们不妨把类型和数据范围分开讨论。

2 字节的数据类型可以表示  $(2^8)^2 = 65'536$  种状态，也就是可以存储 65'536 种值。有符号类型编码了 0~32'767 和 -32'768~-1 的数字（换言之，从 -32'768 到 32'767）；而无符号类型就不需要编码负数了，节省出来的容量可以编码更多正数，所以无符号类型编码了 0~65'535 的数字。

4 字节的数据类型可以表示  $(2^8)^4 = 4'294'967'296$  种状态，可以存储 4'294'967'296 种值。有符号类型编码了 -2'147'483'648~2'147'483'647 的数字；无符号类型编码了 0~4'294'967'295 的数字。

8 字节的类型可以表示  $(2^8)^8 \approx 1.84 \times 10^{19}$  种状态。有符号类型编码了 -2<sup>63</sup>~2<sup>63</sup>-1 的数字；无符号类型编码了 0~2<sup>64</sup>-1 的数字。

于是乎，我们只需要通过 `sizeof` 算出它的内存空间大小，再根据它是有符号类型还是无符号类型，就可以判断出它的数据范围是什么了。

C++ 还有另一种更简便的方法，那就是直接使用 `limits` 头文件中定义的 `numeric_limits` 来输出某一类型的数据上下限。`numeric_limits` 给出的结果正是本电脑的类型数据范围。

以下是实现这个功能的完整示例代码：

代码 2.1: Integer\_Limits.cpp

```
// 这是一个简单的C++程序，用于输出一些整型数值的上/下限值
#include <iostream> //标准输入输出头文件
#include <limits> //标准算术类型属性头文件，numeric_limits 定义于此
using namespace std; //使用 std 命名空间
int main(){ //主函数，程序执行始于此
```

<sup>8</sup>`long long` 及 `unsigned long long` 在 C++11 标准以后可用。本书默认使用 C++17 标准，故不再反复提及此问题。

```

cout << numeric_limits<short>::lowest(); // 输出 short 类型的下限值
cout << endl; // 输出换行，下同
cout << numeric_limits<short>::max(); // 输出 short 类型的上限值
cout << endl;
cout << numeric_limits<unsigned>::lowest(); // 输出 unsigned 类型的下限值
cout << endl;
cout << numeric_limits<unsigned>::max(); // 输出 unsigned 类型的上限值
return 0;
}

```

在 Coliru 上，该程序代码的运行结果如下：

---

```

-32768
32767
0
4294967295

```

---

这里需要介绍几点：

首先，`numeric_limits` 是一个类模版<sup>9</sup>，它要接收一个“类”作为模版参数。比如说，在这个例子中，我们分别使用了 `short` 和 `unsigned` 作为模版参数，然后用 `lowest()` 和 `max()` 成员函数<sup>10</sup> 来求出我们想要知道的值。如果你愿意，也可以把这段代码中的类型改成其它任何一个基本数据类型，然后观察输出，比如 `double` 或 `char`。

其次是“输出换行”的问题。当我们的代码中有多个输出语句的时候，程序会在运行时按顺序执行这些输出语句。但是程序不会自己添加分隔符<sup>11</sup>，它会这样输出（以刚才程序为例）：

---

```
-327683276704294967295
```

---

这不便于我们阅读和理解，所以我们应当人为地把两个输出用分隔符隔开。而 `cout` 就是一个换行符，输出它的效果相当于另起一行。我们当然可以使用其它的换行符，比如我们把输出换行的语句改成输出逗号，

```
cout << ','; // 这相当于输出一个字符字面量 ',', 于是程序会输出一个逗号作分隔符
```

那么输出应当是这样的：

---

```
-32768,32767,0,4294967295
```

---

我们现在还没到讲解输入输出控制的时候，但这方面的知识我们又不得不了解，所以在这里先开个头，以后我们还会频繁遇到的。

最后还有一个问题：假如我的运算超过了整型的表示范围，会发生什么？比如对于有符号型 2 字节的变量来说， $32767+1$  会得到什么？答案是  $-32768$ ！这是一种整数溢出（Integer Overflow）现象，详细的原理我们之后再谈；总之我们需要记住，在写代码的过程中需要对可能出现的数据进行预判，尽量避免出现这种运算结果“装不下”导致的情况。

<sup>9</sup>关于类模版的有关知识，我们将在第十一章中讲解。

<sup>10</sup>关于函数的基本知识，我们将在第四章中讲解；而关于类的成员函数，我们将在第八章讲解。

<sup>11</sup>分隔符（Delimiter），指的是将一系列内容按同样的格式分开的符号，它有点像标点符号，但又不尽然。分隔符既方便人类阅读，又让计算机能够分辨不同的数据。比如说，我们用空格将一系列词语分开，这时空格就是它们的分隔符。

## 浮点型

浮点型数据的编码方式比整型数据要复杂些，我们在精讲篇中再作详细探讨。

我们一般使用 `float`, `double` 和 `long double` 来存储浮点型变量。一般来说，`float` 型占用 4 字节内存空间，`double` 型占用 8 字节内存空间，而 `long double` 型占用 16 字节内存空间。我们同样可以用 `numeric_limits` 来求出它的上/下限。这里我就不再演示代码了，想要尝试的读者可以自行修改代码 2.1 并运行之。

浮点数据和整型数据有一个重要的差异在于，很多时候浮点数表示的都是“近似值”。例如，在 17 位小数的显示精度下，`double` 型的 0.1 和 0.2 相加的结果是 0.3000000000000004；而 0.3 则显示为 0.2999999999999999。

那这个结果对还是不对呢？在精度要求不高的情况下它是对的，因为当你四舍五入到 16 位小数的时候它就相等了。而 `cout` 默认显示至多 6 位有效数字，所以这样的误差是微乎其微的。<sup>12</sup>

还有另一个问题。假如我们需要很高精度的计算，比如要求圆周率取到小数点后 20 位，那么有效数字位数仅有 15~17 位的 `double` 型是不能满足我们的要求的。这时有经验的程序员当然会首选 `long double` 了。

```
const long double Pi = {3.14159265358979323846};  
// 定义一个长双精度浮点型常量 Pi，精确到小数点后 20 位
```

看起来没有丝毫问题，对吗？但是问题恰恰就出在字面量上！

前面我们就提过，编译器会将所有浮点字面量统一视为 `double` 类型的字面量。因此，字面量 3.14159265358979323846 本身就是一个 `double` 型数据，它早在编译时就因为不能容纳如此高的精度而发生了精度上的损失。那么用损失后的只有 15~17 位精度的数字去给 `long double` 数据初始化也是枉然！

但我们也有应对方法：指定字面量的类型。在 C++ 中，我们可以用后缀的方式来指定整型或浮点型字面量的类型。当我们在一个浮点数后加 `f` 后缀时，就指明了它是 `float` 单精度浮点数，比如 `1.f`, `.57f`, `2.718f`；而当我们在一个浮点数后加 `l` 后缀时，就指明了它是 `long double` 长双精度浮点数，比如 `1.l`, `.57l`，以及 `3.14159265358979323846l`，这样就不致有 20 位以内的精度损失了<sup>13</sup>。

除了使用后缀以外，我们还有一些其它的字面量表示语法。如果你刚才已经试过输出浮点数的上/下限，那么你可能会看到这种格式的输出结果：`1.79769e+308`。这是什么意思呢？

实际上，浮点数表示的范围跨度很大，比如 `float` 可以表示  $-3.402'823'4 \times 10^{38}$ ~ $3.402'823'4 \times 10^{38}$  范围的值。这么大的值不适合用一般方法表示，所以在计算机中我们有一种类似科学记数法的形式。

它的规则很简单。一个整数或浮点数都可以在后面加上 `e`（或 `E`，均可）和指数（可正可负）的形式。例如：

`2.99792458e+8`（小写字母，带正号）；  
`6.02214076e23`（正号可省略）；  
`1.602176634E-19`（大写字母也可以）；  
`.000662607015E-30`（不一定非要在 1 到 10 之间嘛）；  
`25.81280745E+3`（形式很灵活，这样写也可以）；  
`3141.59265358979323846264338327950288e-31`（末尾的 `l` 说明它是长双精度浮点数，注意：其位置在指数之后）

当然，浮点的字面量表示还有更多花样，不过这里就不再介绍了。

整型数据的字面量同样有很多后缀可以用来指定类型，比如 `ull` 后缀规定了 `unsigned long long` 类型。但这并不是刚需，因为大多数情况下编译器都能自动为整型字面量匹配合适的类型。

<sup>12</sup> 这里还涉及浮点数相等性的判断问题。正因浮点数多为近似值，所以直接使用相等性运算符极易出现失误。实际操作中我们多用两数相减，通过差值的绝对值是否足够小（比如，限定小于  $10^{-5}$ ）来判断它们是否相等。

<sup>13</sup> 诚如上文所言，浮点数的微小精度损失普遍存在，但只要在我们的承受范围内就可以。

## 字符型

在第一章，我们就介绍过 `char` 字符类型和相应的 ASCII 编码。我们也提及，字符类型其实是一种特殊的整型。它与整型的编码方式有相同之处，而与浮点型的编码方式则是天差地别。

字符型在输出时，主要是为了表示，而非为了计算。所以在用 `cout <<` 输出字符型数据时，输出的并不是它的 ASCII 码值，而是这个字符本身，比如

```
cout << '0'; // 输出字符型字面量 '0'
```

虽然 '`0`' 的 ASCII 值是 48，但程序的输出实际上是 `0`。<sup>14</sup>

如果我们要输出它的 ASCII 码值呢？我们可以使用后文马上就要讲到的类型转换方法，将 `char` 类型数据转换成 `int` 类型来输出，或者是通过减去 ASCII 码为 0 的字符 '`\0`' 来实现——因为两个 `char` 类型之差的结果为 `int` 类型。

```
cout << '0' - '\0'; // 输出相当于 48-0，也就是 48
```

你可能有注意到，这个 ASCII 码为 0 的字符长得好像有点奇怪。在 C++ 中还有许多类似的字符，像 '`\t`'、'`\n`' 之类，它们统称为转义字符（Escape sequence）。转义字符的真实含义不是字面上的含义，它表达了特定的控制字符。比如 '`\n`' 指的是换行符，如果我们输出一个 '`\n`'，就相当于为输出换了一行。那么我们也可以用它来替代 `endl`<sup>15</sup>。

```
cout << numeric_limits<char>::lowest(); // 输出 char 类型的下限值
cout << '\n'; // 输出换行符，效果相当于 cout << endl;
cout << numeric_limits<char>::max(); // 输出 char 类型的上限值
```

不同电脑环境下 `char` 类型的范围有两种可能：一种是 -128~127，另一种是 0~255。无论哪种可能，都涵盖了 ASCII 码的基本范围 0~127。

那么剩下的范围呢，它们有什么作用？这不仅取决于你的系统，还取决于你使用的语言。如果你使用的是 Windows 中文版，你可以测试下这段代码：

```
// 适用于 GBK 编码
cout << char(196) << char(227) << char(186) << char(195);
```

它的输出应该是“你好”。你还可以在 Coliru 上测试下这段代码，这个结果不受你使用的系统和语言的影响：

```
// 适用于 UTF-8 编码
cout << char(228) << char(189) << char(160)
    << char(229) << char(165) << char(189);
```

它的输出也应该是“你好”。

实际上这里涉及到更深奥的可变宽度编码<sup>16</sup>技术。在 Windows 简体中文的环境中常用 GBK 编码，这是一种变宽编码，其中 ASCII 字符占一个字节，汉字占两个字节（比如第一段代码）；在 Windows 繁体中文的环境中早期常用 Big5 编码，这是一种定宽编码，所有字符均占两个字节；而许多系统现在都支持 Unicode，其中以 UTF-8 实现方式为多，所有字符占一至四个字节不等，而 CJK 文字<sup>17</sup>常用者为三字节（比如第二段代码），罕见字为四字节。在泛讲篇中我们不会深入讲这些知识，也不会普遍使用，我们留到精讲篇再来谈。

<sup>14</sup> 在代码中，'`0`' 和 `0` 类型不同。但在控制界面上，无论是人类的肉眼还是计算机程序，都无法分辨它究竟是字符还是数字——甚至说，它们就是同一个东西！我们在第三章会对此有所涉及。

<sup>15</sup> 实际上这两种换行方法有更细微的区别，即何时清空缓冲区的问题。不过它不会影响我们的输出结果，我们不必细究。

<sup>16</sup> 可变宽度编码（Variable-width encoding）是一种字符编码方案，它对不同的字符用不同长度的编码来表示。相对地，ASCII 是一种定宽编码，它的每个字符都用统一的单个字节来表示。

<sup>17</sup> CJK 文字是中文、日文和韩文的统称，包括全数汉字及变体、假名和谚文等。

## 字符串简介

字符串并非基本类型，但是它很重要，我们就在那里一并介绍。

还记得我们的[代码 1.1](#)吗？在这个代码中，我们输出的 "`HelloWorld!`" 就是一个字符串字面量。顾名思义，字符串字面量就是一串字符构成的字面量。每个字符串都有一个我们看不到的结束符作为结尾，它就是 '`\0`'。

我们可以用 `sizeof` 来取它的内存大小。这个字符串一共有 10 个大小写字母、1 个标点和 1 个空格<sup>18</sup>，外加结束符 '`\0`'，全部是 ASCII 字符，所以它的内存占用应当是 13 字节。那么我们用 `sizeof` 算算，验证一下我们的猜想。

```
cout << sizeof ("HelloWorld!"); // 也可以不加括号
```

输出果然是 13。

需要留心，字符是用单引号套起来的单个字符（转义字符是个例外，它允许你使用反斜线加一个字符）<sup>19</sup>；而字符串是用双引号套起来的任意多个字符。转义字符同样可以用在字符串中，比如

```
cout << "ABCD\nabcd";
```

这段代码的输出应为

---

```
ABCD abcd
```

---

这里的 `\n` 就起到了换行符的作用。如果要用 `sizeof` 来计算，它的内存占用应当是 10 字节。

## 布尔型

布尔型是一种特殊的整型，它一般用来表示一种逻辑判断的信息。举例来说，“`2>3`”是一个逻辑判断，它的值为“假”(`False`)；而 “`"HelloWorld!"`” 的内存占用为 13 字节”也是一个逻辑判断，它的值为“真”(`True`)。

布尔型数据用关键字 `bool` 来定义，它只有两个可能的取值：`true` 和 `false`。在编码时，它们分别被编为 `1` 和 `0`。

按理说，我们只需要一个比特就能表示一个布尔变量。然而，内存当中的比特并不是独立的寻址单元，所以一个布尔型变量必须要用一整个字节来存储。<sup>20</sup>读者可以用 `sizeof` 来验证它。

布尔型数据在输出时默认以整型数据的格式输出，于是 `false` 会被输出为 `0`，而 `true` 会被输出为 `1`。

```
bool judgement {2>3}; // 定义 bool 型变量 judgement，其值为 false
cout << judgement; // 因为 judgement 的值为 false，所以输出 0
```

我们可以通过一些语法设置来强制 `cout` 输出 `true` 或 `false`。

```
cout.setf(ios_base::boolalpha); // 设置格式化标志
cout<<true; // 这样一来，它就会输出 true 而非 1
```

我们现在先不讨论这些细节，在第十三章中会谈及。

<sup>18</sup>在本书中，空格字符以 `_` 表示，以便于阅读。（当然，这可能会对读者使用 `Ctrl+C/V` 造成一定的麻烦）

<sup>19</sup>实际上 C++ 中允许使用单引号套起来的多字符字面量（Multicharacter literal），但我认为没有必要在本书讲它。感兴趣的读者可以阅读[What do single quotes do in C++ when used on multiple characters?-Stack Overflow](#)。

<sup>20</sup>标准模版库中的 `vector<bool>` 采用了优化方法，使得每个字节可以存储 8 个布尔数据，这就提升了存储空间的利用率。

## 2.3 运算符

本节我们将会接触到一个非常有趣的知识：运算符。许多教材在讲运算符的时候讲解不够透彻，虽然不太影响基础层面的知识掌握，但对以后的学习，尤其是运算符的重载，是比较不利的。我在本节会用函数的视角来剖析运算符，希望能为读者带来清晰的认识和比较透彻的理解。

### 运算符的操作数与返回值

来看一个简单的例子：

```
int num {13 * 5 + 9}; // 定义num并初始化为13*5+9
```

我们可以把这句代码分成两步操作：第一步是把  $13*5+9$  求出来，第二步是初始化。第一步的求值过程还可以进一步拆分，先求乘法，再求加法。那么我的问题是：乘法操作把哪两个数乘起来了？加法操作又把哪两个数加起来了？初始化操作又把哪个数的值给了 `num`？

让我们来梳理一下程序的操作过程，这样就会对这些问题有清晰的认知了：

1. 首先，乘法操作把 13 和 5 乘起来，并求得 65，这是毫无疑问的。
2. 其次，加法操作把  $13*5$ ，也就是 65，和 9 加起来，并求得 74。这时候加号左边的  $13*5$  被加号看作一个整体，它操作的不是 5 而是  $13*5$ 。
3. 最后，初始化操作把  $13*5+9$ ，也就是 74 的值初始化给 `num`，这个过程没有再求出什么。这时候初始化所用的  $13*5+9$  被初始化语句看作一个整体，它操作的不是单个数而是整个表达式。

从这个过程中，我们可以看出这样一个特点：有些操作过程会直接用我们给定的字面量（也可以是变量）来计算；而有些操作过程需要用到前面某一步算出来的值，并把它当作一个整体来运算。有些操作过程还会产生一个结果，后面的操作可以利用这个结果。

现在我开始甩专有名词：我们把一个运算符用来计算的数据（包括表达式整体）叫作它的操作数（Operand），而一个运算符计算出来得到的结果（一个表达式整体）叫作它的返回值（Return value）。操作数和返回值都是对于某个运算符而言的，比如  $13*5$  既是乘法运算的返回值，又是加法运算的操作数。初始化的过程其实不涉及某个运算符<sup>21</sup>，所以我们这里不再探讨。

我们在关注操作数和返回值的时候必须留意它们的类型，因为类型不同可能会带来天差地别的结果。我们在前面已经提过  $8/3$  和  $8./3$  的区别了，这是因为：当除号的操作数都是 `int` 型时，返回值就是 `int` 型；而只要有一个操作数是 `double` 型，即便另一个操作数是 `int` 型，返回值也将是 `double` 型<sup>22</sup>。

两个 `char` 类型数据相减，得到的结果是 `int` 类型的，因此我们上一节中讲到的 '`'0'`-'`'\0'`' 的语法能够求出 '`'0'`' 的 ASCII 码值。

赋值语句也有返回值，它返回的是左操作数的一个引用<sup>23</sup>。在 C++ 中我们偶尔会遇到连续赋值的语法，如下例：

```
int a, b, c; // 连续定义a,b,c，但不初始化，此时它们的值均不确定
a = b = c = 0; // 连续赋值
```

鉴于我们还没有讲运算符结合性，我先用一个容易理解的带括号版本来介绍：

```
a = (b = (c = 0)); // 连续赋值，等价于a=b=c=0;
```

<sup>21</sup>如果涉及自定义类型的初始化，它其实是一种构造函数。

<sup>22</sup>实际上这个过程更复杂，它先将 `int` 型经过类型转换变为 `double` 型，再进行两个 `double` 数据的计算。

<sup>23</sup>我们会在第五章介绍引用。即便读者尚不知道何为引用，也不会对读者理解此节造成困难。

圆括号 (Parentheses<sup>美</sup>/Round brackets<sup>英</sup>) 改变了计算的顺序<sup>24</sup>, 它将把括号内的部分当作一个整体来对待。于是这段代码执行的操作就是

1. `c=0` 中的赋值运算符将左操作数 `c` 的值变为右操作数 `0`。最后返回 `c` 的引用。
2. `b=(c=0)` 中的赋值运算符将左操作数 `b` 的值变为 `(c=0)` 的返回值, 即 `c`。因为此时 `c` 的值为 `0`, 所以 `b` 的值变为 `0`。最后返回 `b` 的引用。
3. `a=(b=(c=0))` 中的赋值运算符将左操作数 `a` 的值变为 `(b=(c=0))` 的返回值, 即 `b`。因为此时 `b` 的值为 `0`, 所以 `a` 的值变为 `0`。最后返回 `a` 的引用。

在上述过程中, 每个赋值运算符都接收了两个操作数, 并给出了一个返回值。当然, `a=(b=(c=0))` 的返回值就不需要使用了, 因为我的目的已经达成。如果你愿意, 还要以加上一个 `cout<<` 来输出 `a` 的值。

```
cout << (a = (b = (c = 0))); // 输出 a=(b=(c=0)) 的值, 应为 0
```

再举一例。`<<` 在左操作数是整型数据的时候, 意思是左移位运算符。而 C++ 中重载了这个运算符<sup>25</sup>, 当其左操作数为 `ostream` 类的对象时, 实现输出功能, 最后返回左操作数的引用。C++ 中我们最常用的 `ostream` 类的对象就是 `cout`, 所以我们可以用它和 `<<` 操作符实现连续输出。

```
const double a {3.6e3}, b {6.02e-23}; // 定义两个 double 常量并初始化
cout << a << '\n' << b; // 输出 a, 再输出换行符, 再输出 b
```

我们按照相同的思路来解释一下这段代码的行为:

1. `cout<<a` 中的左移运算符的左操作数是 `cout`, 所以标准输出将把右操作数 `a` 的值输出到相应设备上。最后返回 `cout` 的引用。
2. `cout<<a<<'\n'` 中的第二个左移运算符的左操作数是 `cout<<a` 的返回值, 即 `cout`, 所以标准输出将把右操作数 `'\n'` 这个转义字符输出到相应高备上 (起到换行作用)。最后返回 `cout` 的引用。
3. `cout<<a<<'\n'<<b` 中的第三个左移运算符的左操作数是 `cout<<a<<'\n'` 的返回值, 即 `cout`, 所以标准输出将把右操作数 `b` 的值输出到相应设备上。最后返回 `cout` 的引用 (这个返回值我们不再需要用到)。

`>>` 叫做右移位运算符。C++ 中重载了这个运算符, 当其左操作数为 `istream` 类的对象时, 实现输入功能, 最后返回左操作数的引用。C++ 中我们最常用的 `istream` 类的对象就是 `cin`, 所以我们可以用它和 `>>` 运算符实现连续输入。

```
int i, j, k; // 定义 int 数据 i,j,k。不能定义为常量, 否则将发生错误
cin >> i >> j >> k; // 先后为 i,j,k 连续输入
cout << i << endl << j << endl << k; // 连续输出 i,j,k 并以换行为间隔
```

读者可以仿照上例, 对本例中连续输入的语法进行解读。

## 运算符的优先级

在上小学时我们就学过“先乘除, 后加减”这类的口诀。这其实就体现出一个先后顺序的问题, 亦即“先算什么, 后算什么”。C++ 中有数十种运算符, 除了我们熟知的四则运算以外, 还有赋值运算符、模运算符和移位运算符等等。如果我们不人为规定一个“先操作什么, 后操作什么”的顺序的话, 有些代码的理解处理将会变得很麻烦。我们举个例子:

<sup>24</sup>再次提醒, 方括号`[]`和花括号`{}`它们有各自的作用。我们不能用它们来达到改变运算顺序的目的, 只能用圆括号`()`。

<sup>25</sup>对于重载后的运算符, 我们仍然称其为左移运算符, 但它的功能已经发生了实际上的变化。

```
num = 6 * 7; //假设num是一个已经定义了的int型变量
```

如果先操作乘法运算符，再操作赋值运算符的话，应当是这样的过程： $6*7$  的返回值作为赋值运算符的右操作数，经由赋值运算符把它赋给 `num`。赋值运算符的返回值不需要再使用。最终 `num` 变为 42。

假如先操作赋值运算符，再操作乘法运算符呢？（虽然对于人来说有点费解，但电脑不会怀疑，只会默默执行）它先将 6 赋值给 `num`，然后将赋值运算符的返回值，即 `num`，与 7 相乘。乘法运算的返回值没有被使用。最终 `num` 变为 6。

再比如，`a+b*c` 究竟是先算加法还是先算乘法，结果很有可能是不同的。因而我们需要人为规定，某个运算符要优先于某个运算符来计算——这就是优先级（precedence）的问题。

在初学阶段，我建议读者把“优先计算”理解成“套括号”。C++ 规定乘法运算符的优先级比加法要高，那么我们就为乘法运算符和乘法运算符所操作的数套上括号，变成 `a+(b*c)`。套上括号，就意味着 `(b*c)` 是一个整体，它的返回值将会作为加法运算的右操作数。

C++ 中的赋值运算符优先级很低，比四则运算都要低。那么按照我们的思路，如果加法，乘法和赋值运算符同时出现，我们应该先给乘法套括号，然后是加法，最后才是赋值。如图 2.1 所示。

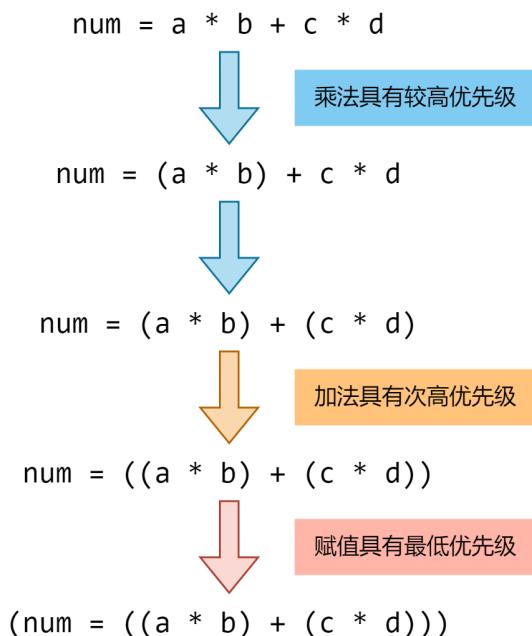


图 2.1: 代码 `num=a*b+c*d;` 的套括号解释

再看这段代码：

```
cout << 3 + 4; //输出3+4的值
```

在 C++ 中加法运算符的优先级高于移位运算符，所以它就可以解释为 `cout<<(3+4)`，就会输出 7。

比较运算符，包括大于号 `>`，小于号 `<`，相等号 `==26`，不等号 `!=` 大于或等于 `>=`，小于或等于 `<=`，它们会比较左右操作数的关系，然后以 `bool` 数据的形式给出一个逻辑判断。正如前文所说，我们可以用 `cout` 来输出一个布尔型数据（只是默认以数值的方式输出），比如这样：

```
cout << 3 != 4; //输出3!=4的结果，预期是ture，输出是1
//error: no match for 'operator!=' (operand types are
//'std::basic_ostream<char>' and 'int')
```

<sup>26</sup>在 C++ 中，单个等号 `=` 是赋值运算符，而 `==` 是一个比较运算符。

但是这段代码在编译期就出现了问题，而报错信息也显得十分奇怪：“没有匹配的运算符 !=（操作数类型分别是 `std::basic_ostream<char>` 和 `int`）。”我们甚至都搞不懂为什么会出现这样的报错信息！

但是解决方法也很简单，只需为 `3!=4` 套上括号：

```
cout << (3 != 4); // 输出 3!=4 的结果，编译正常，并输出 1
```

为什么是这样呢？原因在于，`<<` 的优先级要高于这一系列的比较运算符。所以如果按照 `cout<<3!=4` 的写法的话，`cout<<3` 会先套上括号，于是变成了 `(cout<<3)!=4`。显然，`cout`（也就是 `cout<<3` 的返回值）和 `4` 比较是没有意义的，C++ 也没有定义这种语法，于是才有了我们之前遇到的问题，即“没有匹配的运算符”。正因如此，我们才需要把 `3!=4` 套上括号，以此来表示它要作为一个整体来计算。

而假如我们需要为一个 `bool` 变量来赋值的话，那就不会有这样的问题了，因为比较运算符的优先级要高于赋值运算符。

```
bool judgement; // 定义 bool 型变量 judgement，不初始化
judgement = 2 * 5 > 9; // judgement 应当被赋值为 1
```

在这个赋值语句中，乘法运算符的优先级最高，大于号次之，赋值运算符最低，于是它的正确表达就是 `judgement=((2*5)>9);`。

C++ 中的运算符实在太多，优先级规则也相当复杂，这里无法尽数讲解，读者可以参考[附录 A](#) 的内容。在后面的章节中我们会接触到更多运算符，到那时我们会再细致讲解。

## 运算符的结合性

那么同一优先级的运算符又该按照什么顺序来计算呢？这就涉及到结合性的问题了。

举一个简单的例子：

```
cout << 8 / 4 / 2; // 结果会是什么？
```

我们知道 `/` 的优先级比 `<<` 要高，所以套括号的顺序一定是先给除法运算符套括号。但是这里有两个除法运算符，究竟是套成 `((8/4)/2)` 还是 `(8/(4/2))` 呢？这就要涉及到结合性的问题了。

C++ 中规定了每个运算符的结合性（Associativity）。对于同一优先级<sup>27</sup>来说，如果它的优先级是从左向右（Left-to-right），那么套括号的顺序就是从左向右；如果它的优先级是从右向左（Right-to-left），那么套括号的顺序就是从右向左。

除法运算符所在的优先级，其结合性是从左向右的，所以在乘法、除法和取模混合运算时，它就会从左向右套括号，而直观表现上就是“从左向右算”。赋值运算符所在的优先级，其结合性是从右向左的，所以在连续赋值的时候，它就会从右向左套括号，而直观表现上就是“从右向左算”。

所以连续除法中 `8/4/2` 就应该解释成 `(8/4)/2` 而非 `8/(4/2)`；而连续赋值中 `a=b=0` 就应该解释成 `a=(b=0)` 而非 `(a=b)=0`。如图 2.2 所示。

综上所述，运算符的优先级决定了不同种运算符套括号的顺序，而结合性决定了同种运算符套括号的顺序。于是编译器可以通过这套规则，给每个运算符都“套起括号”，从而让每个运算符都知道它的操作数是什么，而不至于出现歧义。

当然我也需要提醒读者，“为所有运算符都套括号”只是我们初学编程时的权宜之计；实际编程中，只有当默认的优先级与结合性不能满足我们的要求时，我们才通过加括号的方式改变运算逻辑（比如，在 `(a+b)*c` 中加括号是为了先算加法后算乘法）。当我们把这些知识掌握熟练了之后，我们自然就清楚在什么时候有必要加括号，而那些可有可无的括号就不会再用了。

<sup>27</sup> 相同的运算符一定是同一优先级。当然，同一个优先级可能包含若干种运算符，比如乘法、除法和模运算这三种运算符就属于同一优先级。

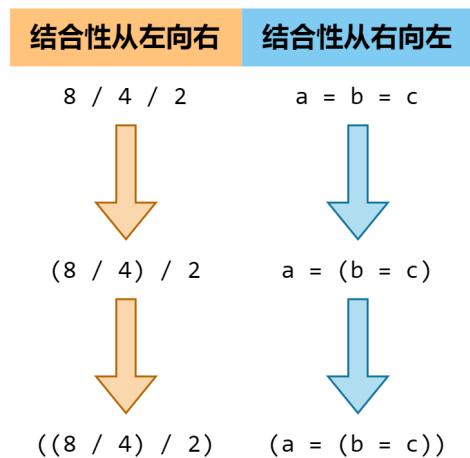


图 2.2: 除法运算符和赋值运算符的结合性

运算的语义

我们在实际编程的时候经常会犯一些语法（Syntax）上的错误，包括但不限于：

- 给常量赋值。这是因为常量不能赋值，编译器会禁止这种行为。
  - 用字面量进行统一初始化时发生了收缩转换。比如某电脑的 `short` 类型占 2 字节，但使用了 `short s {40000};` 这样的语法。
  - 写错了变量名，比如我定义的是 `num` 结果不小心写成 `nun`，然后编译器就会报错，理由是“`nun`”未定义。
  - 写出 `cout<<a` 这种表达式。可能你想输出 `a<b` 的结果但是忘记套括号了；也可能你想先后输出 `a` 和 `b` 但是不小心把 `<<` 写成了 `<`。但无论如何，编译器都会以“没有匹配的运算符”为由报错。

这些语法上的错误都很容易被编译器查出来，然后告诉你。你可能一眼就知道怎么回事，也可能花了好久才终于找到了问题所在，总是这些语法错误都是会在编译期由编译器告诉你的。

但是还有一类语义（Semantics）上的错误，包括但不限于：

- 不小心把 `a==3` 写成了 `a=3`。前一个是进行相等性判断，而后一个是赋值。这种失误常见于新手，但对于老手来说也在所难免。
  - 受到数学表达的影响，在 C++ 代码中写出诸如 `2<x<10` 这样的表达式。
  - 想要计算 `8/3*1.` 的浮点结果，但是算出来的却是整型的，原因不明。

语义错误的特点是，编译器不会认为你的代码是“错误”的。它会认为你写的代码都有你自己的目的。只要不违反 C++ 标准对于语法的规定，你就可以想怎么写就怎么写。但是很多时候，因为各种原因，“我们真正写出来的代码”并没有符合“我们想要实现的功能”，这时我们就会说，你的代码在语法上正确，但在语义上错误。这种语义上的错误是很可怕的，因为很多编译器不会给你提醒，你也不会立刻知道哪里出错，只有在你运行程序并发现结果不合预期之后，才会去找寻问题的来源。

`2 < x < 10` 这句代码就属于一种语义错误，因为 C++ 没有三元比较运算符<sup>28</sup>，我们不能用这样的方式来进行比较。但是它在语法上正确，所以编译器会作如下分析：因为 `<` 的结合性是从左向右，那么 `2 < x < 10` 可以被解释成 `(2 < x) < 10`。

`28C++` 中唯一的三元运算符是条件运算符，它可以有三个操作数。我们之后再谈。

那么这样的解释方式能否达到我们的目的呢？我们分析一下就知道了：无论  $2 < x$  的返回值是 **true** 还是 **false**，它都小于  $10^{29}$ ，所以  $2 < x < 10$  得到的返回值永远是 **true**！

那么怎样写才能达到我们的目的呢？这里要用到逻辑运算符“**&&**”（逻辑与）。逻辑与运算符用于计算两个布尔数据，仅当左右操作数均为 **true** 时，返回值才为 **true**；否则，返回值为 **false**。

那么我们拆解一下数学表达式  $2 < x < 10$ ，它的含义可以阐述为“ $2 < x$ ”并且“ $x < 10$ ”，必须两个要求都满足。于是我们可以通过逻辑与运算符来把它们拼合起来，写成  $(2 < x) \&\& (x < 10)$ 。

又因为 **&&** 运算符的优先级低于 **<**，所以我们可以不用套手动括号了，直接写成  $2 < x \&\& x < 10$ ，这才是语义上正确的写法。

至于  $8/3*1$ . 为什么在语义上是错误的，我们放到下节的类型转换部分探讨。

## 2.4 类型转换

C++ 提供了丰富的基本类型，包括各种整型、浮点型、字符型和布尔型等<sup>30</sup>。它们用途各异，表示范围也有差别。类型区分明确有很多好处，其中一点就是，许多运算符可以在操作数类型不同的情况下，表示不同的含义，得到不同的结果。将来我们会学习函数重载和运算符重载，这种方法大大增加了代码的可扩展性，并让我们可以通过自定义重载的方法针对不同类型实现多种多样的功能。前文提过，`<<` 在左操作数为 `ostream` 类时，实现输出功能；而在左操作数为整型时，实现左移位计算功能，这就是对类型多样性意义的最佳诠释。

但是，不同类型之间的隔阂也让人望而却步。假设我要输入两个整数并求它们的商和模值，而我们这么写代码：

```
int a, b; // 定义两个 int 型变量，无须初始化
cin >> a >> b; // 输入 a 和 b 的值
cout << a / b << endl << a % b; // 输出 a/b 和 a%b
```

那么就会出现经典问题，整型除法的返回值自动截尾，我们得到的是不精确的结果。那么如果我们把代码改成这样呢？

```
double a, b; // 定义两个 double 型变量，这样 double 除法得到的结果就精确了
cin >> a >> b; // 输入 a 和 b 的值
cout << a / b << endl << a % b; // 输出 a/b 和 a%b
//error: invalid operands of types 'double' and 'double'
//to binary 'operator%'
```

这样代码就会报错，因为“用 **double** 型数据与 **double** 型数据相取模是不允许的”。

类型限制太死板，难免就会在这种需要灵活性的地方出现问题，而 C++ 解决这个问题的方式就是类型转换（Type cast/Type conversion）。

### 隐式类型转换

我们知道 '**0**' 的 ASCII 码值是 48，但是它是 **char** 型，而 C++ 中的 `cout <<` 在面对 **char** 型时会直接输出这个字符，而不是它的 ASCII 码。如果我们要知道它的 ASCII 码，要怎么办呢？

我们在此之前已经介绍过，可以用 '**0**'-'**\0**' 的方式来实现输出整数值的功能。然而实际上，C++ 并没有为 **char** 和 **char** 类型定义减法运算。那么它为什么不仅能通过编译，还能计算出正确的结果呢？这就是隐式类型转换（Implicit type conversion）的功劳了。

<sup>29</sup>**bool** 型数据和 **int** 型数据比较的时候会发生类型转换，这点我们会在之后提及。

<sup>30</sup>这里所列举的都属于算术类型。除此之外，基本类型中还有 **void** 和 **nullptr\_t** 类型，但这里不讲。

在 C++ 中，算术运算符<sup>31</sup>不能接受低于 `int` 级别的操作数。在处理这些操作数的时候，将会先把它们转换成 `int`（或 `unsigned`）类型，然后再进行计算。于是 '`0`'-'`\0`' 的操作数分别被转换成 48 和 0，然后它们再经由减号计算出结果 48。

再来看这段代码：

```
int a, b; // 定义 int 型整数 a 和 b
cout << 1. * a / b; // 试图利用隐式类型转换计算 1.0*a/b 的结果
```

我们使用上一节讲过的运算符的知识来解读一下输出语句。乘除法的优先级高于左移运算符，而乘除法的优先级相等，其结合性是从左到右，于是它可以这样套括号来理解：`cout<<((1.*a)/b)`。

接下来开始分析类型。在乘法运算符遇到一个整型和一个浮点型的时候，整型会被转换成浮点型。于是在 `1.*a` 这个过程中会发生一个 `int` 到 `double` 的隐式类型转换，然后是两个 `double` 型的数据相乘，结果顺理成章地就是 `double` 型；

接下来是这个 `double` 型的 `1.*a` 作为整体与 `int` 型的 `b` 相除，于是 `int` 型的 `b` 也需要被转换成 `double` 型数据再参与运算。而 `double` 型数据的除法就不致于有截尾级别的精度损失了。整个过程如图 2.3(a) 所示。

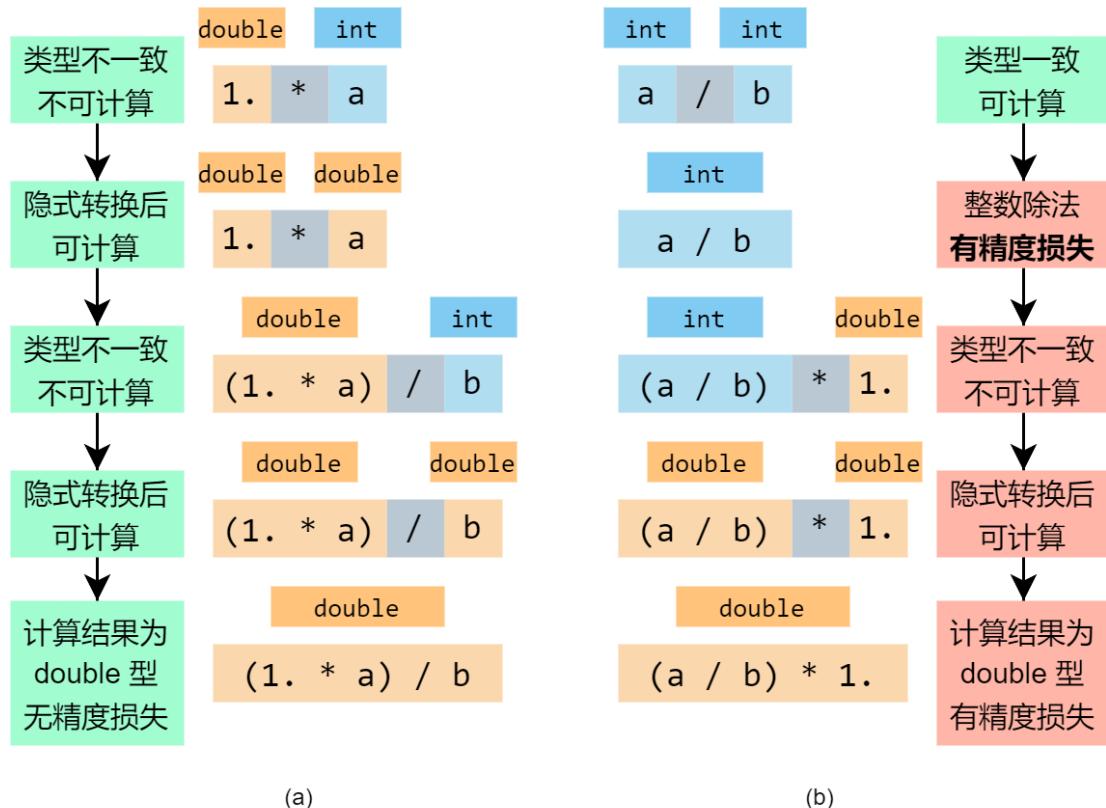


图 2.3: `1.*a/b` 与 `a/b*1.` 的隐式类型转换过程图解

那么我写成这样行不行呢？

```
cout << a / b * 1.;
```

我们同样分析一下这个语句，它应当被理解为 `cout<<((a/b)*1.)`。当除法运算符遇到两个整型数据 `a` 和 `b` 时，它会按照整型的除法来计算，这个过程中就存在精度损失！

这之后，虽然 `a/b` 作为一个整体被转换成了 `double` 型，但是之前丢掉的一部分信息再也找不回来了，于是即便最后算得 `double` 型的结果，也无济于事。整个过程如图 2.3(b) 所示。

<sup>31</sup> 算术运算符包括正负号、四则运算符和取模，以及位运算；不包括自增和自减。

隐式类型转换的语法普遍存在于我们的代码中。凡是在运算符（或函数）需要某一个类型而我们提供的操作数（或实参）是另一个类型时，隐式类型转换都有可能发生。上一节中提到的语义错误代码 `2<x<10` 就是如此，`2<x` 先求出了一个 `bool` 型的结果，然后这个结果隐式转换成 `int` 型，再和 `10` 进行比较。在类型转换的过程中，如果 `2<x` 是 `false`，它就会被转换为 `0`，反之被转换为 `1`。

在赋值语句中也常常出现隐式类型转换，比如

```
double d {3.14159}; // 定义 double 型变量 d 并初始化为 3.14159
int i; // 定义 int 型变量 i, 无初始化
i = d; // 将 d 的值赋给 i, 这时会发生从 double 向 int 的类型转换, 损失小数部分
```

值得注意的是，类型转换并不意味着原变量的类型或值发生了改变！类型转换只是创建了对应类型的临时变量<sup>32</sup>；在类型转换后 `d` 依旧是 `double` 类型，其值也依旧为 `3.14159`。

整数提升（Integral promotion）和浮点提升（Floating-point promotion）是两类特殊的隐式类型转换，它们的特点是精度完全没有损失。浮点提升是比较简单的，`float` 类型可以无精度损失地转化为 `double` 类型，就像是“升级”了一般。同样地，`char` 类型可以无精度损失地转化为 `int` 类型，也是一种整数提升。<sup>33</sup>

布尔转换（Boolean Conversions）是隐式类型转换当中比较特殊，而又十分常用的一类转换。如图 2.4 所示，它的转换规则很简单：如果别的基本数据类型要转换为 `bool` 类型的话，`0` 会转换成 `false`，而其它的所有数都会被转换成 `true`；如果 `bool` 类型要转换成其它的基本数据类型的话，`false` 会转换成 `0`，而 `true` 会转换成 `1`。

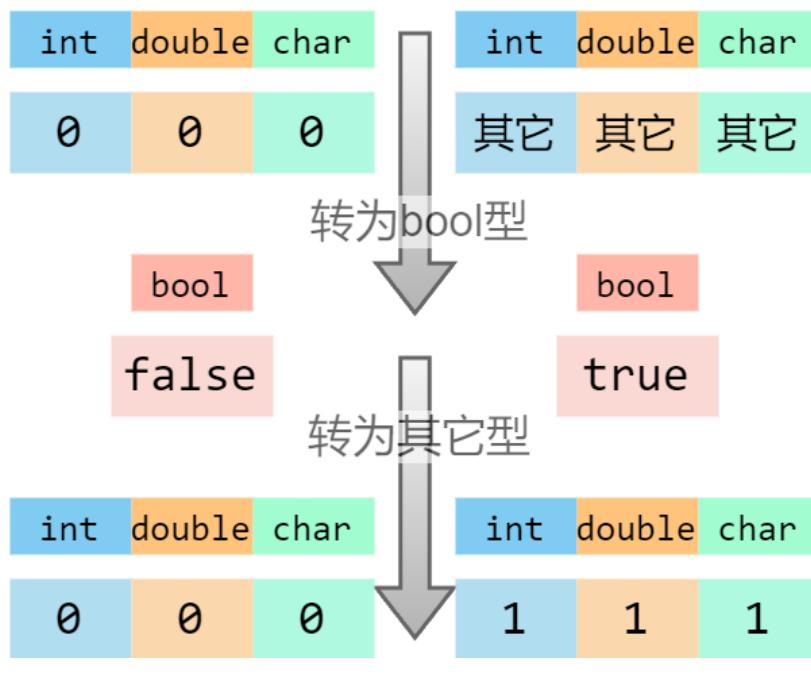


图 2.4: 布尔转换规则

## 显式类型转换

隐式类型转换固然方便，但并不是任何时候都能满足我们的需要；有些时候甚至会背离我们的需要，从而造成很多不便。对于这种情况，显式类型转换（Explicit type conversion）就比较有用了。

C 风格的显式类型转换语法是

<sup>32</sup>“临时变量”这个概念对读者来说或许有点陌生。无妨，我们会在之后慢慢熟悉它的。

<sup>33</sup>但是，并非所有的无精度损失的类型转换都是整数提升，这里没有必要细究，只把它们都当作隐式类型转换即可。

```
(<类型> <数据>)
```

这样就能得到目标类型的一个临时变量，我们可以拿它去计算了。

比如说，如果我要把整数运算 `a/b` 变为浮点运算，那么我可以这样写：

```
cout << (double) a / (double) b; // 显式类型转换
```

实际上我们只要写一个显式类型转换就可以了，另一个操作数自然会被隐式类型转换过去，不用麻烦我们再写一遍了。

如果我们嫌弃 `double` 类型的精度不够高，解决方法就是显式地转换为 `long double` 类型！

```
cout << (long double) a / b; // 显式类型转换；b 会被隐式转换为 long double
```

在 C++ 中，模运算不支持浮点型到整型的隐式类型转换，比如本节开头的例子里就体现了这一点。在保证两个操作数为整数的前提下（实际上不是整数也可以，不过两个小数本来就不可以取模，强行取模又有什么意思呢），我们也可以使用显式类型转换。

C++ 中引入了两类新的显式类型转换风格：一类是构造函数风格的类型转换；另一类是包含 `static_cast` 在内的四种特定类型转换语法。构造函数风格的类型转换语法为

```
<类型> (<数据>)
```

比如说上述的除法可以写作 `double(a)/double(b)`<sup>34</sup>。而四种特定类型转换语法中我们只讲 `static_cast`，其余的留到精讲篇。它的语法是

```
static_cast<[类型]> ([数据])
```

在这里我为了避开括号对 `<>`，转而使用括号对 `[]` 来表达，请读者留意。

C 风格类型转换（和构造函数风格类型转换）最大的好处就是方便。它是一种比较“粗暴”的类型转换方式<sup>35</sup>，写起来又很简单，所以新手程序员更青睐这种写法。但是在面对复杂类型转换，尤其是涉及指针等复合类型的时候，这样的转换很容易出错。因而，出于类型安全的考量，指定类型转换方式要更稳妥些。我也建议读者按照自己的需求，尽量使用 `static_cast` 等特定的转换语法，避免 C 风格那种大包大揽的转换。

比如说，如果我们要用 `static_cast` 来实现 `double` 型转 `int` 型并进行取模运算，我们可以这样写：

```
cout << static_cast<int> (a) % static_cast<int> (b);
// 将a和b都类型转换为 int，然后就可以使用模运算了
```

这里需要多加注意，无论你使用哪种风格的类型转换语法，在这里必须将 `a` 和 `b` 都显式类型转换为 `int`，因为在模运算中没有整数和浮点数之间的隐式类型转换规则。

而假如你想要把 `double` 型转换为 `int` 型来进行四则运算的话，你就要注意把每个数据都转化成 `int` 型，不然……

```
double a, b; // 定义浮点变量a,b
cin >> a >> b; // 输入a和b
cout << static_cast<int> (a) * b; // 试图进行显式类型转换
```

在这里，乘法运算符在面临一个 `int` 操作数和 `double` 操作数的时候，它会怎么做呢？如图 2.5 所示，它将把 `int` 通过隐式类型转换变为 `double` 型！这样一来我们就白干了。

总而言之，类型转换是一个非常复杂的话题。限于我们目前接触到的类型尚少，这里只作简单概括。将来我们接触复合类型和自定义类型的时候，还会对此作针对性讲解的。

<sup>34</sup>注意：它不适合类型关键字中间有空格的类型，比如 `unsigned char` 和 `long double`。

<sup>35</sup>它是 `const_cast`, `static_cast` 和 `reinterpret_cast` 三种转换方法的统合体。

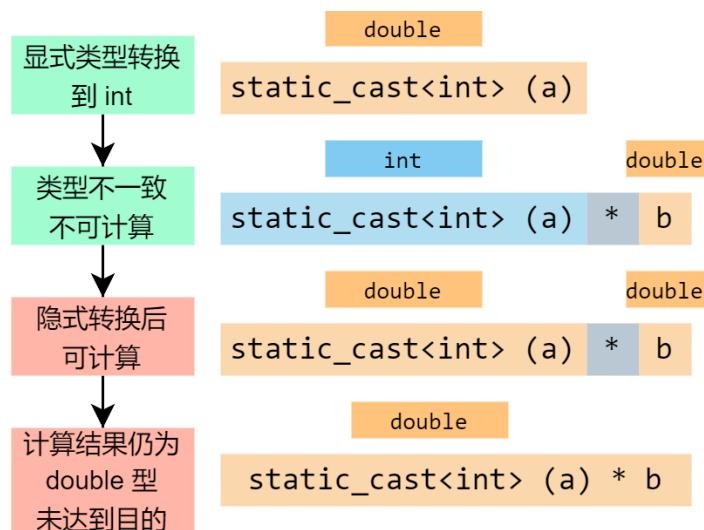


图 2.5: 显式类型转换, 但是又被隐式类型转换改回去了

# 第三章 程序的流程控制

在第二章我们讲解了数据类型、运算符，以及类型转换的相关知识。对于一些初学者容易犯的错误，我们也尽量加以解释。有了这些知识之后，读者就已经具备完成一个简单的程序的能力了。

但是仅有这些还不够。回想一下我们的[计算程序](#)，我们为了求出  $(a+b*c)/d$  的值，先定义了四个变量，再写了一个表达式，通过输出来求得算式的值。如果用户希望自定义各个数据的值呢？只能改代码。

“改代码”是很可怕的。我们不能苛求每个使用计算器的用户都有看懂并修改代码的本领。所以我们可能需要条件结构（或曰选择结构），根据用户按下的按键（加减乘除，或者归零，或者别的什么）来判断应当该使用何种运算符，然后把这些规则写到程序中。

一个计算器可以进行多次计算。它不会每算完一个数就自动关机，等你再次开（代码 1.2）机才能进行下一次计算。但是我们的程序每次算完都会终止，如何让它能够多次计算呢？我们可能需要循环结构，让这个程序周而复始地运行同一段代码，直到有一个信号让它停下来。

对于初学者来说，这些听上去可能有点天方夜谭，完全无从下手。没关系，本章我们就来讲解如何设计和控制程序的流程，以便写出功能更加复杂的代码——一个简单的计算器程序当然也不在话下了。

## 3.1 简介：结构、流程与次序

### 编译器如何处理代码？

在 2.3 节中，我们提到过“语法”和“语义”的区别。语法正确与否，事关代码能否通过编译；而语义正确与否，则事关程序能否实现我们想要的功能。在编译期，预处理器会进行一系列预处理，例如把 `iostream` 头文件的内容复制到我们的代码中，以便编译器进行编译。有些值也是在编译期进行计算的，比如 `sizeof`。这类处理和计算过程早在编译结束之前就已经操作完毕，我们将其统称为**编译时行为（Compile-time behaviour）**。

```
int a {3};  
cout << sizeof a << endl; // 输出 a 的内存占用  
cout << sizeof (double); // 输出 double 的内存占用
```

之所以这种操作在编译时就可以算出来，是因为编译器在编译期就已经可以确定它的值。还记得吗，无论 `a` 的值如何变化，它占用的内存空间总是那些，不会变多和变少。换言之，无论 `a` 的值是多少，`sizeof a` 总是一个在编译时就可以确定下来的常量。那么在编译时就把它计算出来，当然更划算了。

另一些操作，我们在编译时不能确定它的值，所以需要在程序运行时，具体情况具体分析。比如 `x+y`（假如 `x` 和 `y` 是两个变量），这个运算结果当然依赖于具体变量的值，无法预测，所以要在编译时求解。我们将其统称为**运行时行为（Runtime behaviour）**。以下所列代码均为运行时行为。

```
double d; // 定义 double 型变量 d，程序为其分配内存空间  
cin >> d; // 输入 d 的值  
d = d * d; // 为 d 赋值
```

```
cout<<d; // 输出d的值
```

读者可能好奇，代码 1.2 中的操作有哪些是编译时行为，哪些是运行时行为呢？事实上，只有 `include` 和 `using` 语句是编译时行为；而 `double` 变量的定义、表达式的计算和输出，全部是运行时行为。

编译器没有人那么智能。它判断这个表达式能否在编译时操作的依据是“它是否在编译时就能确定下来”。由变量构成的表达式当然不满足这样的条件（因为变量可以在任何时候发生改变），于是编译器一刀切，认为只要有变量参与，这个表达式就不能在编译期求得。

那么常量呢？看似可以，但它也不总是可行的！我们在 2.1 节中介绍过，常量是可以用变量来初始化的。

```
int g; // 定义一个临时变量g
cin >> g; // 通过输入来改变g的值
const int Grid = {g}; // 用g的值来为常量Grid初始化
```

也就是说，常量的值也未必能在编译期就确定下来。一般来说，编译器会对实际情况进行判断，如果一个常量能在编译期求出，这就是编译时行为；否则就放在运行期求出，这是运行时行为。

另外就是，字面量构成的表达式也可以在编译期求得。比如说

```
int a {15+8/2-3}; // 定义变量a并初始化为14
```

请注意，“ $15+8/2-3$  的计算”和“定义数据并初始化”是两个过程！计算过程中只涉及字面量，所以编译器可以直接求得，这是编译时行为；而变量的定义和初始化是无法在编译期就完成的，这是运行时行为。

C++11 以后的标准允许我们定义常量表达式（Constant expression），来拓展编译时行为的可能性。我们可以把它理解成一种新型的字面量<sup>1</sup>。

```
constexpr double Pi {3.14159}; // 定义常量表达式Pi，现在Pi是一个字面量啦
constexpr double Pi2 {Pi * Pi}; // 定义常量表达式Pi2，现在Pi2也是一个字面量
```

仅由字面量（广义地说，常量表达式）构成的表达式可以在编译期求出，比如这里的 `Pi*Pi`。但假如中间环节出现了变量，或者是不能在编译时确定下来的常量，那么就会出现问题。

```
const double Pi {3.14159}; // 定义常量Pi，它不是常量表达式，不能视为字面量
constexpr double Pi2 {Pi * Pi}; // 尝试用Pi来定义常量表达式Pi2
//error: the value of 'Pi' is not usable in a constant expression
```

编译器报错，显示信息为“`Pi` 的值不能用于常量表达式中”。这说明我们必须用常量表达式构成的表达式来定义常量表达式。

我们此前介绍过 `numeric_limits<int>::max()` 这类的写法，它也是常量表达式。编译器能够直接算出这个值，并把它当作字面量来看待，就不需要运行时再去计算了。

## 控制台如何处理输入/输出

源代码经过编译和链接等步骤形成的文件多种多样。有些是库文件（Library），它们可以作为目标文件（Object file）与其它文件链接，我们暂不讨论；有些是可执行文件（Executable），我们在前两章中有三个完整的.cpp 代码，可以编译成可执行文件（如果你使用 Windows，那么这类文件的扩展名就是.exe），并给我们输出执行结果。当然还有其它的形式，比如多媒体，我就不一一列举了。

可执行文件也多种多样。有些是图形用户界面（Graphical user interface, GUI）的，有些是文本用户界面（Text-based user interface, TUI），还有些则是命令行界面（Command-line interface, CLI）的。

<sup>1</sup>事实上，常量表达式是字面量的超集。但用字面量的思路来理解常量表达式要容易得多。

早期的电脑为用户提供的都是命令行界面，用户和电脑之间通过纯文本来进行交互。这种方式不直观，也很难用，用户通过键盘来输入纯文本，电脑通过显示器来输出纯文本。这就给计算机学习造成很高的门槛。

文本用户界面主要基于文本，但相比命令行界面来说，在操作上要轻松一些。用户可以使用鼠标和键盘来控制输入。图 3.1 是一个基于文本用户界面的控制台程序。



图 3.1: Alpine, 一个基于文本用户界面的电子邮件客户端

图片来源：维基共享资源

图形用户界面则为用户提供了丰富的输入方式，输出方式也从纯文本扩展到多媒体等各种类型。它在呈现上也更为直观。一个浏览器就是一个典型的图形用户界面程序。笔者还记得自己开始用电脑的时候，很少使用键盘，基本上就是拿着鼠标在点点点。

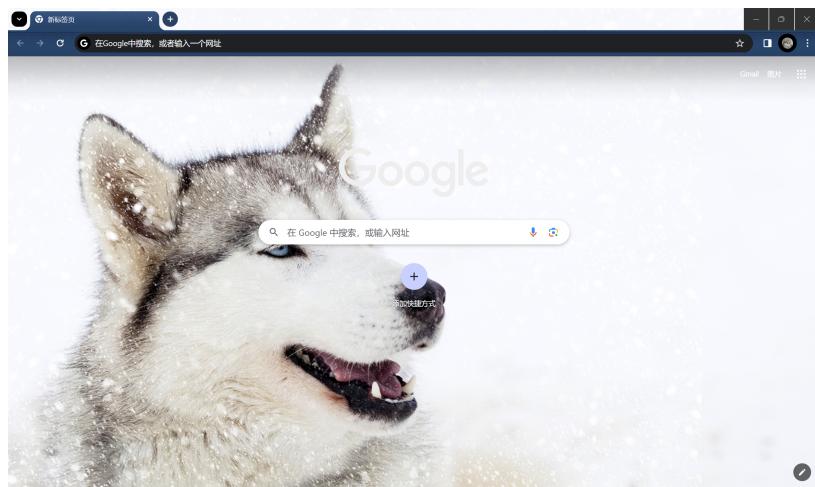


图 3.2: Chrome, 一个典型的图形用户界面程序

C++ 功能丰富，无论是命令行界面还是文本用户界面，我们都可以用代码的方式来实现。而本书所介绍的内容全部适用于命令行界面的编程。我们写出来的程序也全部都是控制台程序（Console Application）。

我们与控制台程序的交互往往依靠键盘来实现输入，靠屏幕来实现输出<sup>2</sup>。C++ 使用流（Stream）来控制输入输出。我在键盘上按下一系列按键，最后按下回车，我的输入信息就以流的方式传递给程序，这是输入；程序得到了我的输入，再经过运算，求得我想要的结果，也以流的方式显示到控制台程序的命令行界面上，这是输出。

<sup>2</sup>并不尽然，比如可以通过文件操作，或者字符串等方式来实现输入、输出。我们会在第十三章和精讲篇中进行解释。

## 输入

想像一下我们使用命令行界面时的情形：如图 3.3 所示，有一个光标（Cursor）在界面上不停闪烁，等待我们输入。当我们按下一个按键，比如 d，光标的位置上就会出现一个 d，同时光标后移一位。如果我们按下退格键，光标就会前移一位，并删除上一个输入字符。如果用方向键前移光标的位置并再次输入内容，还可能把之前的输入覆盖掉<sup>3</sup>。



图 3.3: 命令行界面输入场景示例

值得注意的是回车键。在我们按下回车键之前，我们输入的内容并未实际发送给计算机，我们可以修改这些内容；而一旦按下了回车键，本行的输入内容将会发送给程序<sup>4</sup>，我们也没机会再修改了。

那么当我们按下回车键之后，计算机将会接收到什么输入内容呢？我们可以用这个程序来检测：

代码 3.1: Input\_to\_ASCII.cpp

```
// 这是一个简单的程序，用来将用户所有的输入转为ASCII码
// 读者无需掌握本代码的全部细节，后续会讲到
#include <iostream> // 标准输入输出头文件
using namespace std; // 使用命名空间 std
int main() { // 主函数，程序执行始于此
    char c; // 定义一个char变量c用以接收输入
    while (cin.good()) { // while 循环，当 cin 状态正常时继续循环
        cout << c;
    }
}
```

<sup>3</sup>只在覆盖模式下有效。计算机文字处理器都有输入两种模式：插入模式和覆盖模式。对于有些系统来说，命令行界面默认使用的是插入模式。

<sup>4</sup>这才是真正意义上完成了一次输入。

```

    c = cin.get(); //通过cin.get()接收输入，并存入c
    std::cout << static_cast<int>(c) << std::endl;
    //强制类型转换为int，故以ASCII码值形式显示输入内容
}
}

```

读者可以自行尝试这个程序。如果你单独按下一个回车键，程序会给出输出 10。这说明，回车键也是一种输入！当你按下回车键的时候，既有“发送输入内容”的作用，又有“输入一个回车字符”作用。这点我们现在还不必深究，读者只需有一个印象即可。

## 输出

如果说输入是我们在命令行中敲下若干字符的过程的话，那么输入就是程序在命令行中敲下若干字符的过程。<sup>5</sup>想象一下，在输出过程中有一个输出光标，程序每输出一个字符，光标就会后移一位；输出换行符时，程序就会换行。

以后我们会谈及更多细节，这里就不再赘述。

## 程序的结构

科拉多·博姆<sup>6</sup>和朱塞佩·雅可比尼<sup>7</sup>于 1966 年在论文<sup>8</sup>中提出了结构化程序理论（Structured program theorem）。结构化程序理论证明了，任何计算过程都可以用以下三种结构及其组合来表示：

- 一个接一个地执行一系列操作，这叫做顺序（Sequence）。
- 根据一个判断条件<sup>9</sup>，选择执行两个操作中的一个，这叫做选择（Selection）。
- 只要满足某个判断条件，就重复执行某个操作，直至条件不满足，这叫做循环（Iteration/Loop）。

图 3.4 是对结构化程序理论的直观解释。

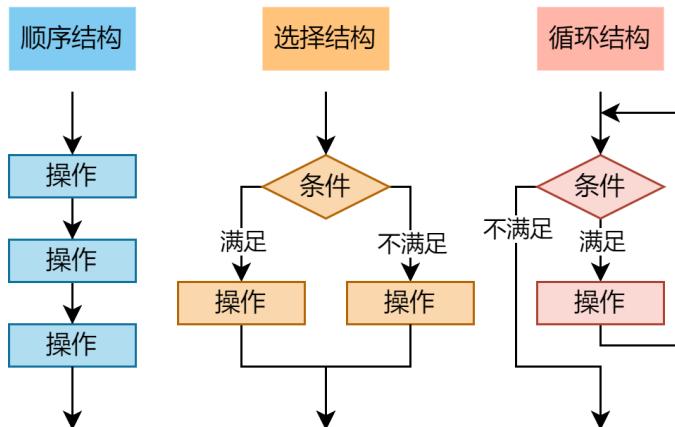


图 3.4: 顺序、选择和循环结构

<sup>5</sup>这当然是不严谨的。一般来说输出流有缓冲区机制，会先把输出内容存入缓冲区，再一次性将缓冲区内容输出，以提高效率。程序输出的逻辑与人工输入并不相同！这里只为方便初学者理解而如是描述。

<sup>6</sup>科拉多·博姆 (Corrado Böhm)，意大利计算机科学家，以其对结构化程序理论和λ演算等的贡献而闻名。

<sup>7</sup>朱塞佩·雅可比尼 (Giuseppe Jacopini)，意大利数学家和计算机科学家，结构化程序理论的提出者之一。

<sup>8</sup>Flow Diagrams, Turing Machines And Languages With Only Two Formation Rules, 1966 年发表于《ACM 通讯》。

<sup>9</sup>一般是一个 **bool** 变量，或者可以隐式类型转换为 **bool** 类型的数据。当然，**switch** 语句是个例外。

对于顺序结构，我们已经很熟悉。我们之前已经尝试了好几个程序的代码。在这些程序的主函数中，代码的顺序就对应着程序执行的顺序。所以一个变量要先定义，后使用。

```
int a {1234}; //先定义
cout << a; //后使用
```

如果把这两句代码的顺序颠倒，编译器就会报错。

```
cout << a; //先使用
//error: 'a' was not declared in this scope
int a {1234}; //后定义
```

报错信息的意思是，“a 尚未定义”。

所以说程序运行过程中还有一个不得不关注的次序问题，我们马上就来讲解一些有关的例子。

## 语句的运算次序

C++ 中有一对有趣的运算符：自增运算符 `++` 和自减运算符 `--`。

自增运算符作用于各种类型的变量<sup>10</sup>，可以使它的值增加 1。自减运算符反之，可以使它的值减小 1<sup>11</sup>。如果我们反复操作自增或自减运算符，可以使数据增加或减小任意值。这种性质使其在循环结构中很好用，我们之后就会介绍。

自增运算符的语法有两种，自减运算符的语法可以仿照写出来，我就不赘述了。

```
<变量> ++; //后缀自增运算
++ <变量>; //前缀自增运算
```

它们实现的功能都是为 `<变量>` 加一，但区别在于“返回值”。前缀自增的返回值是增加之后的值；而后缀自增的返回值是增加之前的值。让我们用一个很简单的例子来说明这个问题吧：

```
int a {3}; //定义 a=3
cout << a++ << '\u'; //输出 a++, 观察返回值
cout << a << endl; //输出 a, 观察 a 的值
cout << ++a << '\u'; //输出 ++a, 观察返回值
cout << a; //输出 a, 观察 a 的值
```

程序的运行结果如下：

---

```
3 4
5 5
```

---

程序给了两行输出，第一行分别是 `a++` 的返回值和 `a` 的值。可以看出，`a` 在自增之后给到的返回值依然是原来的（增加之前）3；而这时 `a` 的值其实是 4。第二行分别是 `++a` 的返回值和 `a` 的值。可以看出，`a` 在自增之后给到的返回值就是增加之后的值了。

自减运算符的道理是相同的，读者可以把上述代码中的 `++` 全部改成 `--`，先猜一下它的运行结果，然后再编译运行一下，检验你的猜想。不出意外的话，程序的运行结果应当是：

---

```
3 2
```

---

<sup>10</sup>包括整型、浮点型和字符型，乃至指针等类型。但 `bool` 型是个例外，C++17 标准起已明文禁止对 `bool` 进行自增和自减运算。

<sup>11</sup>但不要忘记每个类型都有它的范围和精度限制。对 `int` 表示范围的最大值自增将会发生溢出；对 `1e50` 这种过大的数据自增/自减已经超出了其精度承受范围。

## 1 1

一些人会把前缀自增/自减和后缀自增/自减的区别解释为“前缀写法是先加后用，后缀写法是先用后加”。这种理解方式很直观，先加后用就是先增加，再提供返回值；先用后加就是先提供返回值，再增加。

但是我在那里也要提醒读者，这种理解方式未必是正确的！对于内置类型的自增/自减运算来说，不同编译器可能给出不同的运算方式。如果读者能够理解汇编<sup>12</sup>代码，可以比较一下 x86 msvc v19.38 和 x86-64 gcc 13.2 如何编译这段代码：

```
int a {3}, b {a ++};
```

答案是，x86 msvc v19.38 将后缀自增处理为“先用后加”，先返回这个变量本身，再进行自增运算；而 x86-64 gcc 13.2 将后缀自增处理为“先加后用”，先进行自增运算，再提供返回值，而且此处的返回值不是这个变量本身，而是一个存储了原先值的临时变量。

Compiler	Assembly Code (Approximate)
x86 msvc v19.38	<pre> 1   b\$ = -8 2   _a\$ = -4 3   PROC _main 4       push    ebp 5       mov     esp, ebp 6       sub     esp, 8 7       mov     DWORD PTR _a\$[ebp], 3 8       mov     eax, DWORD PTR _a\$[ebp] 9       mov     DWORD PTR _b\$[ebp], eax 10      mov    ecx, DWORD PTR _a\$[ebp] 11      add    ecx, 1 12      mov     DWORD PTR _a\$[ebp], ecx 13      xor    eax, eax 14      mov     esp, ebp 15      pop    ebp 16      ret 17   ENDP _main </pre>
x86-64 gcc 13.2	<pre> 1   main: 2       push    rbp 3       mov     rbp, rsp 4       mov     DWORD PTR [rbp-4], 3 5       mov     eax, DWORD PTR [rbp-4] 6       lea    edx, [rax+1] 7       mov     DWORD PTR [rbp-4], edx 8       mov     DWORD PTR [rbp-8], eax 9       mov     eax, 0 10      pop    rbp 11      ret </pre>

图 3.5: 后缀自增运算在不同编译器下的汇编代码

左图为 x86 msvc v19.38；右图为 x86-64 gcc 13.2

编译支持：[Compiler Explorer](#)

读者尚无必要去纠结这些细节，只需从这个例子中体会到，很多时候，我们或别人“总结”出来的规律只是一种管中窥豹，仅在一定范围内是正确的；一旦离开这个范围，这些规律非但可能不适用，还有可能会为我们准确理解事物逻辑造成更大的障碍。所以读者必须具备这样的意识，对于各种第三方——包括本书在内——总结出来的“规律”保持警觉，并最好通过亲手操作的方式来验证之。

回到自增/自减运算。考虑这样一个赋值语句：`b=a++`。自增/自减运算符的优先级比赋值运算符的优先级高，所以它可以解释为 `b=(a++)`。我们可以进一步把这个过程拆解成三个操作：赋值，`a` 增加一，以及 `a++` 提供返回值。那么问题来了，哪个操作在先，哪个操作在后呢？这就是次序（Order）问题了。

首先我们不难理解，“`a++ 提供返回值`”这个操作肯定要早于“赋值”操作。否则赋值语句就不知道该赋什么值。既然要算出返回值，那么赋值运算符右边的所有操作都应在赋值之前做完，于是“`a 增加一`”这个操作肯定也早于“赋值”操作。

接下来问题来了，“`a 增加一`”和“`a++ 提供返回值`”这两个操作谁先谁后呢？刚才我们就看到了，不同编译器会给出不同的操作次序。但是无论以何种次序以何种次序计算，编译器总会给出正确的结果。<sup>13</sup>

<sup>12</sup>汇编语言（Assembly language），是一类用于各种可编程器件的低级语言。汇编语言不是一个单一的语言，它是一系列语言的统称。汇编语言的指令集严重依赖于具体的器件，不同的器件可能采用不同的指令集，因而汇编代码的移植性很差。

<sup>13</sup>我们也称之为“不定序”。一般来说，这种不定序的情况不会影响到我们的日常编程，因为结果总是一样的。但是也请注意防范“未定义行为”的发生，我们会在后面讲到。

但是并非所有这类操作都能给出“正确的结果”——或者说，压根就没有正确的结果！一个最经典的例子就是

```
int a {0};
cout << a++ + a++;
```

这段代码在不同编译器下的结果是不同的，而且也没有标准答案！其原因在于，加法运算符并不确定左右各个操作的顺序。

如果要追究本源，这就要看 C++ 标准是如何规定的了。C++17 起的标准明确规定，= 右端的所有值计算和副作用发生<sup>14</sup>都要早于左端的任何值计算和副作用发生。而 C++ 标准仍未规定加法运算符两端操作的运算次序，所以不同编译器就有不同的理解。

### 逻辑运算符

在 C++ 标准中，有三个逻辑运算符：逻辑非 (!)、逻辑与 (&&) 和逻辑或 (||)。它们在选择和循环结构中很常用，所以在这里不得不提。我们在前面已经分析了很多运算符，这里只要按照相同的思路来分析就可以了。先来分析逻辑与运算符。

a && b		b	
		false	true
a	false	false	false
	true	false	true

a    b		b	
		false	true
a	false	false	true
	true	true	true

a	! a
false	true
true	false

图 3.6: 逻辑与、逻辑或、逻辑非运算符的真值表

逻辑与运算符只能接收 `bool` 类型的操作数；非 `bool` 型的操作数都会通过[布尔转换](#)来变成 `bool` 型。

逻辑与运算符的返回值也是 `bool` 类型，只有当两个操作数的值都为 `true` 时，它的返回值才是 `true`，否则就是 `false`。

逻辑或运算符和逻辑与运算符很相像，唯一的区别是返回值，只有当两个操作数的值都为 `false` 时，它的返回值才是 `false`，否则就是 `true`。

简单说来，逻辑与运算符要求两个操作数都为真，而逻辑或运算符要求两个操作数有至少一个为真<sup>15</sup>。

逻辑非运算符比较特殊，它只有一个操作数<sup>16</sup>，而且只能以前缀的形式表达。`!a` 是正确的，但 `a!` 就不对了。

逻辑非的功能不复杂：如果操作数的值是 `true`，返回值就是 `false`；如果操作数的值是 `false`，返回值就是 `true`。但是要注意：逻辑非运算符不能直接改变操作数的值，这和自增/自减运算符是不

<sup>14</sup>值计算和副作用将在第四章讲解。在此之前我们可以把它们当作所有操作的统称。

<sup>15</sup>在不同语境下，自然语言中的“或”可能指的是“互斥或”或者“可兼或”。而编程语言中的“异或”与“或”是有很明确的规则的，不可混淆。

<sup>16</sup>像逻辑非、自增/自减运算符这种仅有一个操作数的运算符，我们有时称它们为一元运算符或单目运算符。

一样的。比如 `!a`, 完了之后 `a` 的值是不变的。

关于这三个运算符的优先级和结合性, 读者可以自行查阅附录 A 的表, 我这里就不谈了。图 3.6 是这三个运算符的真值表, 读者可以对照此表, 加深对这三种运算符的印象。

至于运算次序, C++ 对于逻辑与/逻辑或运算符的运算次序有明文规定, 运算符左边的操作必须早于右边的操作完成。

### 短路求值

逻辑与/逻辑或运算符有一个短路求值 (Short-circuit evaluation) 的特性。对于 `&&` 运算符来说, 如果左操作数的值算完了, 发现是 `false`, 那么无论右操作数的值是什么, 整个表达式的返回值都只可能是 `false` 了。于是程序为了节省时间, 就会跳过右操作数的计算。

```
constexpr int init {0}; // 定义常量表达式init, 值为1
int a {init}; //用init来初始化变量a
a++ && a++; //用&&连接两个a++, 预期会使a自增两次
cout << a - init; //a当前值减去初始值, 预期输出2
```

这个程序的最终输出为: 1。这可能有点令人吃惊, 但仔细分析一下, 你就会明白其中原因了。

在 `a++ && a++` 语句中, `&&` 的左操作数会优先计算 (C++ 标准明文规定), 这里 `a++` 后缀自增, 使 `a` 变为 1, 而返回值为 0。

根据布尔转换规则, 0 将转换为 `false`, 这说明无论右操作数为何, 整个计算的结果必然是 0, 于是右操作数的计算过程被跳过, 到输出的时候, `a` 还是 1, 减去 `init` 当然就是最终结果 1 了。

读者可以把 `init` 改为 1 或者是其它数, 输出结果将变为 2。

关于 `||` 的短路求值是同理的, 如果左操作数的值是 `true`, 就会跳过右操作数的计算过程。这不是本章的重点, 所以我就不再赘述了。

## 3.2 选择结构

在此之前我们遇到的代码都是顺序结构的, 程序一个接一个地执行一系列操作。仅用顺序结构是不能完成复杂工作的, 比如根据一定的条件, 作出不同的反应。这时我们就需要用到选择结构了。

`if-else` 结构是最常用的选择结构, 而 `switch-case` 结构在处理特定的条件判断时, 会更加高效。本节我们就来介绍这两种结构。

### if-else 结构

`if-else` 结构的基本格式是这样的:

```
if (<条件>) { //if是必需的
    <若干操作>;
}
else { //如果不需要else部分, 可以不写
    <若干操作>;
}
```

其中的 `<条件>` 要么是一个 `bool` 数据, 要么是可以隐式类型转换为 `bool` 数据的表达式。

这个结构其实不复杂, 我说一遍你就懂了: 如果 `<条件>` 为 `true`, 那么程序就会执行 `if` 块内的代码, 跳过 `else` 块内的代码; 如果 `<条件>` 为 `false`, 那么程序就会跳过 `if` 块内的代码, 执行 `else` 块内的代码。当然, 如果只有 `if` 块而无 `else` 块, 规则就更简单了。

初学者一般都会见过这样的问题：输入一个正整数，要求程序判断它是奇数还是偶数，并输出相应的内容。我们就用 **if-else** 结构来设计这样一个程序。在设计之初，我们需要想清楚一个问题：如何判断一个正整数的奇偶？

我们希望有一个表达式，它要么是 **bool** 类型的，要么可以通过布尔转换成为 **bool** 类型，并且能提供“某正整数是奇数还是偶数”这个信息。这里我们选择取模运算，因为一个正整数在对 2 取模运算后得到的值只能是 0 或 1——如果这个数是奇数，那么得到 1；如果是偶数，那么得到 0。

那么有了模运算的结果之后，如何把它表达成一个“条件”呢？这里我们可以使用两种方法：其一是直接把它作为条件，这个运算的结果会隐式转换为 **bool** 类型，奇数意味着 **true**，偶数意味着 **false**；第二种方法是用相等性运算符  $\text{==}^{17}$ ，我可以用计算结果和 1 比较，等于 1 意味着 **true**；不等于 1 就意味着 **false**。

好，到这里我们已经想清楚了关键部分，那么接下来我们梳理一下我们的程序需要做什么：

1. 定义整型变量（可以使用 **int**，或者考虑到输入一定是正数而使用 **unsigned**），以便后续输入。
2. 输入这个数。
3. 再定义一个数，用来保存这个数对 2 取模的值。
4. 写一个选择结构（**if-else**），根据上一步的值是否等于 1 来采取不同的操作。（这里采用第二种方法）

有了基本的设计思路之后，我们可以开始写代码了。

首先，定义整型变量。我们可以任意起名，但最好起一些有实际含义的名字，这样我们一眼就能知道它是什么，代表什么。比如说，我们就给它起名叫 **num** 吧。

```
int num; // 也可以用 unsigned
```

接下来是输入部分，很简单，用 **cin>>** 就可以了。

```
cin >> num; // 输入 num 的值
```

然后我们定义一个新的变量，可以叫 **mod**，来保存模值。因为这个模值不需要改变，我们可以选择将其定义为 **const** 常量，来防止误改动。

```
const int mod {num % 2}; // num 模除 2 的值；也可以不用 const
```

最后是整个程序最关键的部分：选择结构。我们以 **mod==1** 作为条件，在这个 **if** 块内让程序输出“奇数”，而在 **else** 块内让程序输出“偶数”。

```
if (mod == 1) { // 如果 mod 为 1 满足，mod==1 返回值就是 true，执行此段
    cout << "奇数"; // 输出 "奇数"
}
else {
    cout << "偶数"; // 输出 "偶数"
}
```

这个程序的流程图如图 3.7 所示。可以看出，前面部分是顺序结构，后面部分是选择结构。

最后 Ctrl+V 一下文件包含等等乱七八糟的代码，就是这样：

#### 代码 3.2: odd\_or\_even.cpp

```
// 这段代码可以判断输入的正整数是奇数还是偶数
#include <iostream>
```

<sup>17</sup>前面介绍过，当相等性运算符的左右操作数相等时，返回值为 **true**；不相等时，返回值为 **false**。

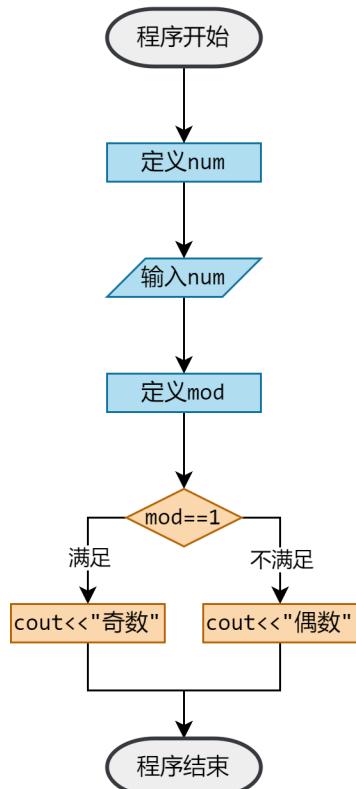


图 3.7: 奇偶判断程序流程图

```

using namespace std;
int main() { // 主函数，代码顺序执行
    int num; // 也可以用 unsigned
    cin >> num; // 输入 num 变量
    const int mod {num % 2}; // num 模除 2 的值；也可以不用 const
    if (mod == 1) { // 选择结构
        cout << "奇数";
    }
    else {
        cout << "偶数";
    }
    return 0;
}
  
```

读者可以自行测试一下，例如输入 1，程序就会给出输出 奇数；输入 2，程序就会给出输出 偶数。

如果甲方有什么其它的奇怪要求，比如“如果是偶数就输出，如果是奇数就不输出”，这时我们就只需要为偶数部分写一个 `if` 块即可，没有必要再写 `else` 块，代码可以这样写：

```

if (mod == 0) { // 注意，判断条件是 mod 是否为 0
    cout << "偶数"; // mod 为 0 说明它是偶数
} // 奇数时什么也不做，那就不需要写 else 了
  
```

## 多路分支的选择结构

刚才接触到的选择结构是两路分支的，我们可以用一个简单的 **if-else** 结构来实现这样的功能。但是有些时候我们需要实现更复杂的多路分支。现在我们需要写一个程序，输入单个字符，并判断它是“大写字母”“小写字母”“数字”还是“其它”，那么我们要怎么做呢？

首先，我们要知道我们应该用什么样的“判断条件”。如何判断一个字符（起个名字，`ch`）是大写字母呢？或许我们可以连用逻辑或来构成一个表达式

```
if (ch == 'A' || ch == 'B' || ch == 'C' || <此处省略23个>)
```

但这样写毕竟还是太麻烦了。我们可以用下面的简单方法来实现：

```
if (ch >= 'A' && ch <= 'Z')
```

还记得我们之前反复提及的 ASCII 码值吗？'A' 到 'Z' 在 ASCII 码表里是连续排列的，如果 `ch` 是 26 个大写字母之一，那么它一定在大于等于 'A' 且小于等于 'Z' 的范围内；相反，如果它不是 26 个大写字母之一，那么它一定不在这个范围内。同样的道理，我们可以用 `ch>='a'&&ch<='z'` 来判断它是不是小写字母，用 `ch>='0'&&ch<='9'` 来判断它是不是数字。

关于条件，我们想清楚了，那么如何实现多路分支呢？我们可以看一下这段代码的实现方式：

```
if (ch >= 'A' && ch <= 'Z') { // 判断是否为大写字母
    cout << "大写字母";
}
else if (ch >= 'a' && ch <= 'z') { // 判断是否为小写字母
    cout << "小写字母";
}
else if (ch >= '0' && ch <= '9') { // 判断是否为数字
    cout << "数字";
}
else { // 其它情况
    cout << "其它";
}
```

有些参考资料将其称为 **if - else if - else** 结构，以 **if** 加条件为开头，中间全用 **else if** 加条件的形式，结尾用 **else**。

但是以我个人的观点来看，实际上并没有 **else if** 这个关键字。它是 **else** 嵌套了 **if** 的结构，本质上只是对 **if-else** 结构的一种特殊应用。这段代码与下面的代码并无任何区别<sup>18</sup>：

```
if (ch >= 'A' && ch <= 'Z') {
    cout << "大写字母";
}
else
    if (ch >= 'a' && ch <= 'z') {
        cout << "小写字母";
    }
    else
        if (ch >= '0' && ch <= '9') {
            cout << "数字";
        }
```

<sup>18</sup>与 Python 等一些语言不同，C++ 对代码的缩进、换行和标识符间的空格几乎没有限制，所以缩进的修改、换行的使用，乃至特定情况下是否套花括号 {} 都不会对编译造成影响。

```

else {
    cout << "其它";
}

```

这样就是 **if-else** 方式理解了。所以 **else if** 只不过是一种更紧凑的写法而已，不能称为一个“关键字”。它的 **else** 部分属于上一个 **if-else** 结构，而 **if** 部分属于下一个 **if-else** 结构。图 3. 描述了这样一种结构是如何搭建起来的。

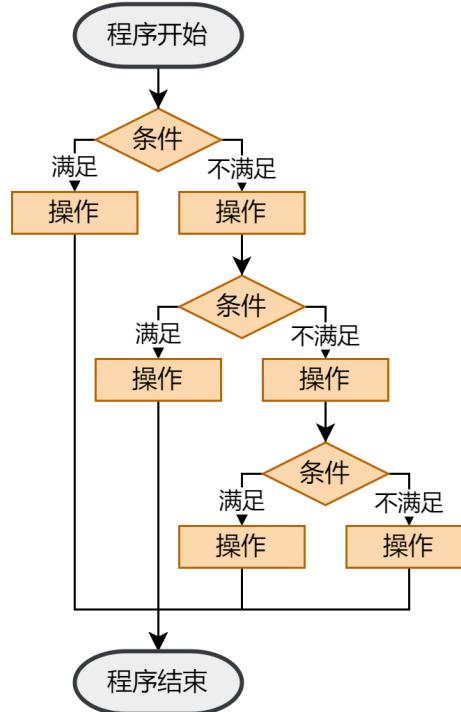


图 3.8: **if-else** 嵌套实现多路分支

## switch-case 结构

**switch-case** 结构是一种在特定情况下可以方便地实现多路选择的选择结构。它的基本语法是

```

switch (<整型数据>) {
    case <字面量>: // 判断分支
        <操作>
        break; // break 并非必需，但请注意后果
    case <字面量>: // 判断分支
        <操作>
        break;
    default: // 默认情况
        <操作>
        break; // break 并非必需，尤其对于最末的 case 而言
}

```

**switchcase** 结构与 **if** 结构不同，它的判断是通过“匹配”进行的。

**switch** 块中的每个 **case** 都是一个标签，能对某个位置进行标记。**switch** 跟随的整型数据将会去匹配每个 **case** 标签后的字面量<sup>19</sup>。一旦匹配到了合适的字面量，程序就会从本句开始顺序执行。

<sup>19</sup> 其实不光是字面量，所有常量表达式都是可以用的。

现在我们要输入一个 1~6 之间的正整数，并输出它的英文 (one, two, ...); 如果输入不是 1~6 之间的正整数，就什么也不做。我们可以按照下面这个使用 **switch-case** 结构的代码来实现这个功能。

```
int num; // 定义待输入变量，注意必须用整型、字符型或布尔型
cin >> num; // 输入 num 的值
switch (num) {
    case 6: // case 可以从 1 到 6，也可以从 6 到 1，也可以打乱顺序，对应即可
        cout << "six" << endl; // 如果 num 匹配 6，就从这里开始顺序执行
        break; // 用 break 来退出 switch 块
    case 5:
        cout << "five" << endl; // 如果 num 匹配 5，就从这里开始顺序执行
        break;
    case 4:
        cout << "four" << endl;
        break;
    case 3:
        cout << "three" << endl;
        break;
    case 2:
        cout << "two" << endl;
        break;
    case 1: // 最末的 case 可以不加 break
        cout << "one" << endl;
}
}
```

可以看出，这样写要比用 **else if** 然后加一众 `num==` 的方式要简洁一些。读者可以自行测试这段代码，看看效果。

**switch-case** 结构有一个特性，叫做穿透效应（Fallthrough behavior）。因为 **case** 只是一个标签，不能起到退出 **switch** 结构的作用，所以如果在写每个 **case** 之后不加以 **break** 的话，程序将会继续运行下一个 **case**、再下一个 **case**，直到遇到一个退出语句或者运行到 **switch** 的末尾为止。

举个例子来说，如果上面的代码不加 **break** 的话，程序运行的结果将是这样的<sup>20</sup>:

---

```
4
four
three
two
one
```

---

而加上了 **break** 之后就会得到这样的运行结果:

---

```
4
four
```

---

我们还可以做合并标签的操作，这个效果相当于“如果 <整型数据> 匹配了 <字面量1> 或者 <字面量2>，就从这里开始执行”。

---

<sup>20</sup>本书默认：键盘输入的内容用黑体标识，程序输出的内容不用黑体。换行符不单独显示，行末空格和回车也忽略掉。如有特殊情况，会特别说明。

举个例子，某个百分制的成绩根据分数划定等级，90~100 分定为 A, 80~89 分定为 B, 70~79 分定为 C, 60~69 分定为 D, 0~59 分定为 E。我们可以用 **switch-case** 结构来实现输入成绩，输出等级的操作。

首先思考一下我们该用什么思路。如果直接按照 100 分制来写 **case** 的话，那么我们要为 **score**（代表分数的整型变量）变量写 100 个 **case**，这个实在太麻烦了。即便考虑到可以用 **default** 来表示 60 分以下的情况，我们也要为 60 及以上的分数写 41 个 **case**。这还不如用 **if-else** 呢，好歹这种方式可以表示范围。

为了简化我们的代码，我们可以取个巧：用 **score/10** 来匹配数据（还记得吗，整型变量的除法是自动截尾的）。这样我们就只需为 6, 7, 8, 9 和 10 来写 **case** 了。

```
switch (score / 10) { //用 score/10 来匹配数据
    case 10:
    case 9: //合并 9 和 10 两个 case, 如果匹配 9 或 10 都从这里开始顺序运行
        cout << "A" << endl; //输出 A
        break; //不要忘记 break, 否则还会顺序执行后面的代码
    case 8:
        cout << "B" << endl;
        break;
    case 6: //打乱 case 的顺序? 完全没问题, 编译器不会误解的
        cout << "D" << endl;
        break;
    case 7:
        cout << "C" << endl;
        break;
    default: //如果以上均不匹配, 就应该是 E 等了吧
        cout << "E" << endl;
        break; //最末 case 可不加 break, 加了也行
}
```

### 3.3 循环结构

在上一节，我们通过选择结构来实现一些“判断”的功能。但是我们的程序还有一些美中不足：它们只能单次运行。每次我打开这个程序，输入完毕再得到一次输出，然后程序就终止了。如果我要多次输入，我就要反复打开程序——这就太麻烦了。所以我希望有一个程序，能够接收任意多次的输入，并在我给定某个条件后终止。循环结构就可以起到这样“反复执行”的作用。

**for** 循环和 **while** 循环都是常用的循环结构，而 **do-while** 结构用得比较少。另有一些循环算法，比如 **for\_each**，我们留到精讲篇再介绍。

#### **while** 结构

**while** 结构的基本格式是这样的：

```
while (<条件>) {
    <若干操作>;
}
```

它与 **if** 结构非常相似，区别仅仅在于，**if** 结构中的内容只会运行一次；而 **while** 的内容可以反复运行（循环）。**while** 结构的规则是：如果 <条件> 为真，那么执行 **while** 块内的若干操作；执行完毕后如果 <条件> 依然为真，那么再次执行若干操作。

那么我们就用 **while** 结构来修改一下我们的奇偶数判断程序，使它可以无限次接收输入——那意思就是需要无限循环。所以最简单的方法就是用 **true** 当作条件了，这样它就永远不会执行下去。

```
int num; // 定义 num
while (true) { // while 的<条件>永远为 true，所以它是无限循环
    cin >> num; // 每次输入一个 num 的值
    const int mod {num % 2}; // num 模除 2 的值
    if (mod == 1) { // 如果 mod 为 1 满足，那么 num 就是奇数
        cout << "奇数" << endl;
    }
    else { // 否则 num 就是偶数
        cout << "偶数" << endl;
    }
}
```

这段代码的结构是 **while** 套上 **if-else**，有点像俄罗斯套娃。其实，这种结构与结构之间的嵌套（Nesting）在程序设计中是普遍存在的。

我们可以想像自己就是一个程序，按照顺序的方式执行代码，遇到 **while** 就检测要不要循环，遇到 **if** 就检测要走哪个分支，想必会更容易理解这种过程。

1. 在程序定义了 **num** 之后，因为 **while** 的条件是 **true**，所以循环块的代码开始执行。
2. 当用户输入了一个值之后，程序会定义 **mod** 来存储 **num** 对 2 的模值。接下来判断 **mod**，并给出相应的输出。至此，一次循环完成。
3. 程序检测 **while** 的条件，因为条件是 **true**，所以循环块的代码开始执行（回到上一步）。

读者可以自行测试这个程序的功能。

让我们思考一下另一个问题：如果用户输入很多次之后要结束程序了呢？我们的 **while(true)** 结构就不尽如人意了——它永远不能自己停下来。虽然我们自己测试的时候可以用窗口上方的关闭按钮来强行关闭，但这一招并不是在任何时候都管用的。现在我们要想一种新的写法，用户可以通过输入特定的内容来结束程序。

一种思路是，输入 **-1** 结束程序。因为用户必须输入正整数，所以 **-1** 已经是非法输入了，不会与程序的正常功能相冲突。那么我们可以试着把这段代码写出来：

```
int num {0}; // 定义 num，这里必须初始化，因为 while 中就需要用到 num 的值
while (num != -1) { // 如果 num 不等于 -1，就说明用户还需要继续输入
    cin >> num; // 每次输入一个 num 的值
    const int mod {num % 2}; // num 模除 2 的值
    if (mod == 1) { // 如果 mod 为 1 满足，那么 num 就是奇数
        cout << "奇数" << endl;
    }
    else { // 否则 num 就是偶数
        cout << "偶数" << endl;
    }
}
```

这段代码看上去好像满足了我们的要求。但是真的是这样吗？以下是一个程序运行示例：

---

```
5
奇数
8
偶数
1
奇数
-1
偶数
```

---

至此程序结束，但是好像出现了一些问题：

- 输入 `-1` 之后，程序应该直接结束，而不是继续给输出一个结果，然后再结束。这说明我们的程序逻辑可能有一些问题。
- 另外，当我们输入 `-1` 之后，即便它多做了一次判断，也不该输出“偶数”这个结果。这说明我们的判断逻辑可能也有潜在问题。

我们先来分析第一个问题。其实只要你把自己想像成程序，然后模拟一下整个过程，就可以理解问题何在了。当用户输入 `-1` 之后，程序并不会直接终止，而是继续顺序执行“定义 `mod`”“判断和输出”这些步骤。要解决这个问题，我们可以用后面讲到的 `break` 语句，或者把程序的逻辑修改成这样：

```
cin >> num; //在while块之外先接收一次输入
while (num != -1) {
    //定义mod和选择结构部分的代码省略
    cin >> num; //在循环末尾接收下次输入，这样下一步就会进行条件判断
}
```

另一个问题是我们的选择结构，貌似在输入 `-1` 时会判断为偶数。负整数并不是本程序的预期输出，所以这个问题可以无视；但出于知识学习的目的，我在这里需要讲一下。这个问题的根源在于我们的取余操作，**负数对正数取余得到的结果可能是负数！**

因此  $-1 \% 2$  的结果是 `-1`，它当然不等于 `1`，所以就会被归入 `else` 状况之下，给出 “偶数”的输出。

解决方法可以是把 `if` 的条件改为 `mod==1 || mod== -1`，用来判断负奇数；或者是直接用 `mod==0` 判断偶数，而把剩下的情况当作奇数来对待。以下是实现代码：

```
int num; //定义num，可不初始化
cin >> num; //先输入第一个num
while (num != -1) { //如果num不为-1，就继续循环
    const int mod {num % 2}; //num模除2的值
    if (mod == 0) { //判断是否为偶数
        cout << "偶数" << endl;
    }
    else { //mod可能是1或-1，我们把它们都归入else中
        cout << "奇数" << endl;
    }
    cin >> num; //接收下一个输出，下一步就会判断num是否为-1
```

```

    }
    cout << "程序结束"; // 输出一个提示语，告诉用户程序结束
}

```

图 3.9 是这个程序的流程图，读者可以参考它来理顺这个代码的逻辑。

我们发现，这个程序不仅可以判断正整数的奇偶，还可以判断负整数的奇偶。因此，它的功能可以得到拓展。不过这样就会出现一个新的问题——输入 -1 的时候怎么办呢？如果还把 -1 作为判断程序结束的标志，那我们就不能通过这个程序来判断 -1 的奇偶；如果输入 -1 的时候也要给出奇偶判断，那么我们就应该用新的方式来标记程序结束。比如说，如果用户输入一个 q (或者别的字母)，我们就结束程序。



图 3.9: while 循环奇偶判断流程图

这种实现方式是可行的，并且相当好用。我在这里先放代码，然后再做解释：

代码 3.3: while 循环奇偶判断.cpp

```

// 这是一个基于while循环的奇偶判断程序，当输入为字母时停止循环
#include <iostream> // 标准输入输出头文件

```

```

using namespace std; //命名空间使用
int main() { //主函数
    int num; //定义num, 可不初始化
    while (cin >> num) { //输入num并使用cin的返回值作为判断条件
        const int mod {num % 2}; //num模除2的值
        if (mod == 0) { //判断是否为偶数
            cout << "偶数" << endl;
        }
        else { //mod可能是1或-1, 我们把它们都归入else中
            cout << "奇数" << endl;
        }
    }
    cout << "程序结束"; //输出一个提示语, 告诉用户程序结束
}

```

这里最巧妙的操作在于使用 `cin>>num` 作为 `while` 循环的判断条件。前面我们提过, `cin>>num` 的返回值还是 `cin` 本身。而 `ostream` 类重载了 `operator bool`, 所以作为 `ostream` 类的对象, `cin` 可以被隐式类型转换为 `bool` 类型, 这是它能作为判断条件的前提。

那么什么情况下 `cin` 会被隐式类型转换为 `true` 呢? 就是在输入状态一切正常的情况下。在 C++ 中, 有很多情况会导致输入状态不正常, 比如:

- 期望输入的是一个整数, 但输入了非数字的字符, 或者输入了浮点数<sup>21</sup>。
- 遇到了 EOF<sup>22</sup>。
- 其它可能导致 I/O 错误的软/硬件问题。

一般情况下 `cin` 的状态是正常的, 转换为 `bool` 类型后返回 `true`; 而一旦遭遇这些情况, `cin` 的状态就会改变, 从而转换为 `bool` 类型后返回 `false`。

所以说, `while(cin>>num)` 实现的功能是: 先输入一个 `num`, 然后返回值 `cin` 被隐式类型转换为 `bool`。如果输入的是整数, 那就一切正常, 返回 `true`, 那就进入循环; 如果输入的是字母 `q`, 或者别的字母, 那就返回 `false`, 那就不会进入循环。<sup>23</sup>

## for 结构

`for` 结构的基本格式是这样的<sup>24</sup>:

```

for (<初始化语句>; <条件>; <迭代操作>) {
    <若干操作>;
}

```

`for` 圆括号内的三个部分都可以留空不填<sup>25</sup>。

`for` 循环和 `while` 循环一样, 都可以用来执行循环。如果不考虑一些特殊的情况(比如条件留空或含 `continue` 语句), 上述 `for` 循环结构完全等价于下面的 `while` 循环:

<sup>21</sup> 输入浮点数的情况更加特殊, 程序会把小数点之前的部分单独作为一个整数来处理(表征出来就是, 多循环了一次), 而将小数点作为非数字字符阻断, 留到下次输入。

<sup>22</sup> 文件结尾(End-of-file)是一个特殊的标记, 表示程序不能再从信息源读取更多信息。EOF 不是 ASCII 中的一部分, 不过有些系统会用值为 -1 的字符型数据来表示它。

<sup>23</sup> 值得注意的是, 如果输入浮点数, 情况还有会有所不同。这里就不再深究了。

<sup>24</sup> C++11 起另有一个很好用的范围 `for` 循环, 但是现在还没到讲它的时候。

<sup>25</sup> 这点与 `if` 和 `while` 结构都不同, 它们的 <条件> 如果留空, 那就不知做什么了。

```
{
    <初始化语句>;
    while (<条件>) {
        <若干操作>;
        <迭代操作>;
    }
}
```

所以理论上，我们可以用 **while** 循环来替代所有常规的 **for** 循环，但 **for** 循环写起来比较简洁（尤其是在涉及数组数据处理方面），所以 **for** 是相当常用的循环语法。关于循环的思路，我已在 **while** 循环部分介绍过，这里就着重介绍一下 **for** 循环的特殊语法。

首先是 <初始化语句>。虽然叫做“初始化”，但既可以是定义语句，又可以是赋值语句（对先前已经定义过的变量赋值），甚至还可以是其它类型的语句，比如输入、输出、计算等各种类型。之所以叫做“初始化”语句，只是因为实际编程中我们一般在这里进行循环的初始化<sup>26</sup>，而不是做些别的杂七杂八的工作。初始化语句部分只会执行一次。

其次是 <条件>。**for** 结构中的条件允许留空，留空的话将默认以 **true** 作为条件。条件会在每次循环开始前都执行一次。

最后是 <迭代操作>。迭代操作永远在 **for** 块内的 <若干操作> 运行完毕后才会执行。值得注意的是，即便在 **for** 块内执行了 **continue** 语句，迭代操作也是不会被跳过的，它依然会执行。迭代操作会在每次循环结束后都执行一次。

以下是一个典型的 **for** 循环输出 1 到 10 的示例代码：

```
for (int i=1; i<=10; ++i) { // 定义并初始化 i，从1起到10止，每次循环自增
    cout << i << endl; // 输出i的值并换行
}
```

接下来我们分析一下这段代码的内容：

首先，定义并初始化 **i** 的值为 1，这是循环初始化的部分。接下来会判断条件，因为此时，**i<=10** 的返回值为 **true**，所以循环部分开始执行，程序输出 1 并换行。最后迭代操作执行，**i** 变为 2。

下一次，先判断条件，此时 **i<=10** 的返回值依然为 **true**，所以循环部分开始执行，程序输出 2 并换行。最后迭代操作执行，**i** 变为 3。

中间的部分同理，不再赘述。

直到 **i** 的值为 10，此时 **i<=10** 的返回值为 **false**，所以循环部分开始执行，程序输出 10 并换行。最后迭代操作执行，**i** 变为 11。

下一次，因为判断条件 **i<=10** 的返回值为 **false**，这个循环不再运行，宣告终止。

各种 **for** 循环结构还可以嵌套，形成更复杂的代码，从而实现更复杂的功能。比如下面的代码是一个两层 **for** 循环，实现的功能是按照 5×5 的布局输出 1~25 的数字。

```
for (int i = 0; i < 5; i++) { // i++ 与 ++i 的效果相同
    for (int j = 1; j <= 5; j++) {
        cout << i * 5 + j << ' ';// 每输出一个数字，以空格隔开
    }
    cout << endl; // 每输出五个数字，以换行隔开
}
```

请留意，外层 **for** 循环的初始化为 **int i=0**，条件为 **i<5**；而内层循环的初始化为 **int j=1**，条件为 **j<=5**。

<sup>26</sup> 所谓“循环的初始化”指的是每次循环开始要预先做的准备工作，而不是在定义变量时做的初始化。

同样是循环 5 次，为什么使用不同形式的初始化和条件呢？读者可以尝试修改它的初始化和条件，然后自行观察输出结果，想必就能领会到其中关键所在了。

### do-while 结构

**do-while** 结构非常容易理解，它的基本格式是这样的：

```
do {
    <若干操作>;
} while (<条件>);
```

它与 **while** 结构的区别是：**while** 结构会先检验条件，判断是否运行 **while** 块内的部分；而 **do-while** 结构会先运行块内部分的 <若干操作>，再检验条件，判断是否再次运行块内的部分。正因如此，**do-while** 结构会保证它的块内代码至少运行一次。

这种结构在我们日常编程中用途有限，而且也完全可以用 **while** 或 **for** 替代，所以这里就不展开讲了。

### continue 和 break 语句

在循环结构中常常还会用到 **continue** 或 **break**，它们可以为我们写代码带来方便。鉴于我们已经在 **switch-case** 结构中介绍过 **break** 语句，那我们就先来讲讲它吧。

**break** 可以用于循环结构或 **switch-case** 结构中，其作用是退出当前所在的循环或 **switch** 块。

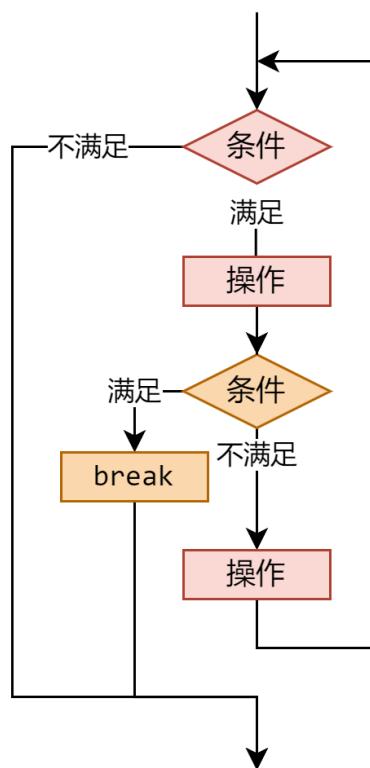


图 3.10: **break** 结构的流程图

举例来说，我们可以用这种方式来重写代码 3.3 的主函数部分，效果相同。

```
int num; // 定义 num，可不初始化
while (true) { // 永远循环，靠内部的break可以退出循环
```

```

    cin >> num; // 输入 num
    if (!cin) { // 如果 cin 状态不正常，将执行 break 语句
        break; // 退出该循环
    }
    if (num % 2 == 0) { // 可以不用定义中间变量 mod，直接用 num%2 来表示
        cout << "偶数" << endl;
    }
    else {
        cout << "奇数" << endl;
    }
}

```

在这里，我们用 `!cin` 来作为判断条件，如果 `cin` 遭遇了不正常输入，它隐式转换会得到 `false`，那么 `!cin` 就是 `true`，于是 `break` 就会执行。

在 `while` 语句中，`break` 执行的效果是直接退出循环，后面的内容也不再执行。在 `for` 语句中，`break` 执行的效果也是直接退出循环，而且迭代操作也不会执行（因为即将退出循环，就没有迭代的必要了）。

读者需要特别注意：`break` 语句只能退出当前所在的循环或 `switch-case` 结构，而且只能退出一层。我们不能使用 `break` 来退出 `if` 块（至少不能仅以退出 `if` 块为目的）。在上面的代码中我们也能看到，使用 `break` 的目的不是为了退出 `if(!cin)`（虽然事实上也实现了这样的效果），而是为了退出 `while(true)`。

请读者分析下面的代码，并猜测可能的输出结果，然后再自己运行一下，加以验证。

```

for (int i=0; i<5; i++) {
    for (int j=1; j<=5; j++) {
        if (i + j > 6) // 如果 i+j>7 就退出
            break; // 会退出到哪里呢？
        cout << i * 5 + j << ' ';// 每输出一个数字，以空格隔开
    }
    cout << endl; // 换行
}

```

实际的运行结果是

---

```

1 2 3 4 5
6 7 8 9 10
11 12 13 14
16 17 18
21 22

```

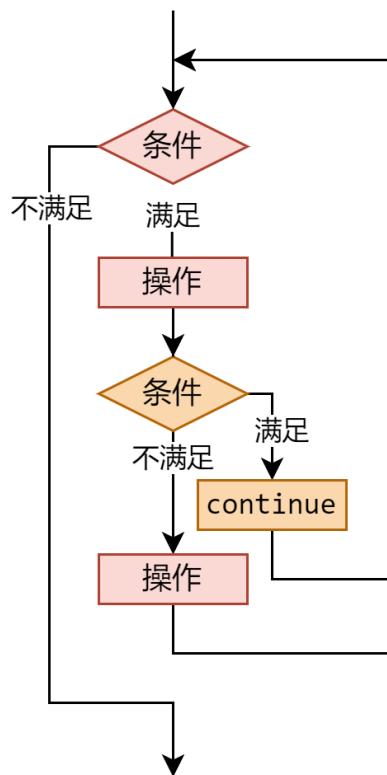
---

从结果上可以看出，`break` 只退出了内层的 `for` 循环，而不会退出外层的 `for` 循环。否则它应该输出到 14 就彻底终止了。

再来看 `continue`。它只能用于循环结构中，作用是跳过后面的代码，直接开始下一轮循环。

需要特别注意的是，`continue` 用在 `for` 循环中，不会跳过迭代操作。也就意味着，在 `for` 循环中使用 `continue` 不是“重新开始本轮循环”，而是“直接从下一轮循环开始”。

`continue` 的用途很广泛，我们来举个实际一点的例子：有些时候，我们需要用户输入一些数字，但我们没有一个机制来防止用户输入不合理的内容（比如，需要用户输入数字，但用户却输入了字母）。一旦用户输入了不合理的内容，`cin` 就会进入错误状态，然后拒绝再读取输入。

图 3.11: `continue` 结构的流程图

为了让 `cin` 读取输入，我们需要清除状态，并清理输入流。我们暂时还不必关注这么多细节，我可以用一个简单的函数来把这项工作包揽：

```

void input_clear() {
    cin.clear(); // 清除错误状态
    while (cin.get() != '\n') // 清除本行输入
        continue;
}

```

我们会在下一章讲函数，在此之前你只需要管怎么使用就可以了。

现在我们试着用这个方法来写一个输入数字求和的程序。用户可以输入若干整数，当输入 0 时结束循环，并给出以上所有数据的和。这里是完整代码：

代码 3.4: sum.cpp

```

#include <iostream>
using namespace std;
void input_clear() {
    cin.clear(); // 清除错误状态
    while (cin.get() != '\n') // 清除本行输入
        continue;
}
int main() { // 主函数
    int sum {0}, x {1}; // x 初始化为 1 是为了防止误结束循环
    while (x != 0) { // 当 x 为 0 时结束循环
        cin >> x; // 输入 x 的值
        sum += x;
    }
    cout << "The sum is: " << sum;
}

```

```

if (!cin) {
    input_clear(); //清除错误和错误输入
    x = 1; //x赋为1, 否则可能导致循环误结束
    cout << "请输入数字! 输入0结束计算" << endl;
    continue; //跳过sum+=x的操作
}
sum += x; //这是一个复合赋值运算符, 等效于sum=sum+x
}
cout << "总数为: " << sum;
return 0;
}

```

这段代码的主函数部分分别定义了 `sum` 和 `x`, 其中的 `sum` 用于求出所有数据的和, 而 `x` 用于在每个迭代过程中存储新的数据。在 `while` 块内, 程序将先读取 `x` 的输入。如果发现 `cin` 的状态不正常, 就说明用户输入了不合理内容<sup>27</sup>。

如果用户读取了不合理内容, 我们需要做两件事: 一是将 `cin` 恢复到正常状态, 以便下次输入; 二是用 `continue` 跳过后面的部分, 以防 `sum+=x` 误执行导致计算结果出现问题。

这里涉及到了一点复合赋值运算符的应用, 我们简单一说便好: 复合赋值运算符的返回值是赋值之后的左操作数的引用。比如加赋值, `sum+=x` 其实相当于 `sum=sum+x;` 减赋值, `sum-=x` 相当于 `sum=sum-x;` 还有乘赋值、除赋值、模赋值等运算符, 它们都可以按照相同的规律来解释。

### 3.4 作用域初步

现在我们希望写这样一个代码, 它接收一个正整数 `n` 输入, 然后计算  $\log_2 n$  的下取整值。因为只需要求下取整的结果, 不要求精确值, 所以我们可以用相对简单的方式来实现它。

对于一个输入 `n` 来说, 如果它等于 1, 那么对数下取整的值当然就是 0 了; 如果它大于 1, 那我们只需要使用 `for` 循环, 每次将 `n` 整除 2, 直到把 `n` 变为 1 为止。读者可以自行举两例并手动计算之, 以此来证明算法的正确性。

我们考虑用这段代码来实现此功能:

```

int n; //定义一个int型变量n
cin >> n ; //输入n的值
for (int i = 0; n > 1; ++i) {
    n /= 2; //这也是复合赋值运算符的应用, 等效于n = n / 2
}
cout << i; //i就是最终结果
//error: 'i' was not declared in this scope

```

我们的代码思路清晰, 在 `for` 循环处定义了 `i`, 并且初始化和判断条件都设计得非常准确, 看上去毫无瑕疵。但是如果你把这段代码拿去编译, 就会发现有问题。

编译器的报错信息已经以注释的方式放在代码中, 它的意思是: “`i` 在这个作用域中没有定义。”但是我们在前面明明定义了 `i` 啊! 而且, 我们使用 `i` 的位置都在定义之后, 不应该出现上述问题才对。

其实这个问题的关键就在于 **作用域 (Scope)**。在 C++ 中, 并非所有变量一经定义就可以永久使用。在很多场合之下, 我们需要使用一些临时变量, 它们可以帮助我们更简单地实现一些功能。但

---

<sup>27</sup>实际情况可能更复杂, 比如遇到了 EOF。我们可以用一些代码来专门检测不正常的原因是否是 EOF 造成的, 但是我们在本章还没有必要细究这些问题。

是任何变量都会占用内存空间，如果临时变量太多，可能会导致内存空间被浪费（它们大都只在很短的时间里使用寥寥几次，却要长期占据宝贵的内存空间）。

我们希望这种临时变量只有很短的生存期，它们只在我们需要的时候“被定义出来”，而在我们不需要的时候“被销毁回收”，腾出内存空间。C++ 如何控制变量的生存期呢？就是通过作用域来实现的。

这里我们只对作用域作简单介绍。最简单的情况，一对花括号 {} (Braces<sup>美</sup>/Curly brackets<sup>英</sup>) 套住的范围就是一个作用域<sup>28</sup>。下面的代码展示了一个最基本的嵌套作用域问题。

```
int main() { // 这是一个外层作用域
    int x; // 在此作用域内定义的变量可以在更内层的作用域中使用
    { // 这是一个内层的作用域
        int y; // 在此作用域内定义的变量可以在本层作用域中使用
        x = y = 2; // 外层作用域的x和本层作用域的y都可以使用
    } // 作用域结束，在此作用域内定义的变量全部失效
    x = 3; // 本层的x可以使用
    y = 2; // 错误！y的生存期已经结束，不能再使用
    {
        // 这是另一个作用域了。在这里，变量x能使用，但y不能使用
        int y; // y经过定义就可以使用了，但这里的y不同于前面的y，它是另一个变量
    }
}
```

**if, else, for, while** 等等结构控制语句也是自带一层作用域的。比如说，**for** 的初始化操作中定义的内容，只在 **for** 及其循环体内可以使用，然而一旦出了这个循环体，**i** 就失效了，无法使用。这个不受花括号使用的限制——有些情况下，我们可以省略花括号，但这些控制语句仍然自带一层作用域

```
int main() { // 外层作用域
    if(<条件>) // 自带一层作用域
        int x; // 相当于在内层作用域定义了变量x
    x = 3; // 错误！在外层作用域中，x已失效，不能使用
}
```

在 **main** 函数体之外，还有一个全局作用域。全局作用域仅供我们定义变量，而其它的操作，如赋值，或者结构控制，必须在函数体内完成。全局作用域下的变量又叫全局变量，它们的生存期是从程序开始到程序结束。**cin** 和 **cout** 就是两个典型的全局变量，它们定义在 **std** 命名空间中。

我们会在第七章，详细讲解作用域及命名空间的概念。

## 3.5 实操：简易计算器的设计

在本章的开首，我们就在考虑设计一个计算器。但因为尚不掌握程序流程控制的方法，所以不能实现。

现在读者已经具备这种能力了，那就让我们写一个简单的计算器，以此温习我们本章的编程知识。

我们先设计一个只支持四则运算和取模的计算器，比如下面的例子，黑体字是输入内容，白体字是输出内容：

---

<sup>28</sup>在变量定义的统一初始化时，我们也用到了花括号，但是那个不能称之为一个作用域。详细的内容我们暂且不讲。

```
8/3
=2.66667
8%3
=2
q
程序结束
```

显然，我们需要用 `double` 或者别的浮点类型来存储数据，这样计算除法时才不致有精度损失。那么遇到取模运算的时候要怎么办呢？我们可以用显式类型转换的方法来变成 `int` 型，然后再计算。

接下来考虑输入的问题，类似于 `8/3` 这样输入要怎么读取呢？实际上，这点我们不用太过关心，因为当输入数据为数值类型时，程序会自动在输入流的非数值的符号处阻断（我们会在第十三章中更详细地进行探讨，这里读者只需知道结论即可）。接下来我们用一个字符输入就可以读取这个运算符字符。最后再用另一个数值类型读取右边的数字即可。

```
double num1, num2; // 定义两个操作数 num
char op; // 定义一个字符用来存储运算符输入
cin >> num1 >> op >> num2; // 依次输入左操作数、运算符和右操作数
```

这样就可以读取 `8/3` 这种格式的输入了。

接下来的主要问题是读取到的 `op` 如何处理。这个计算器可以支持四则运算和取模这五种运算，所以 `op` 的可能取值有五种，分别对应着加减乘除模的 ASCII 码值。要根据具体情况来做不同的运算，那么用 `switch-case` 结构当然是比较方便的选择了。

```
switch (op) { // 判断 op 是什么
    case '+': // 根据 op 的值，采取相应的操作
        double result {num1 + num2}; // 定义成 const 更好，可以防止误修改
        break;
    case '-':
        double result {num1 - num2};
        break;
    case '*':
        double result {num1 * num2};
        break;
    case '/':
        double result {num1 / num2};
        break;
    case '%':
        int result {static_cast<int>(num1) % static_cast<int>(num2)};
        // 显式类型转换，将 double 型的 num1 和 num2 转换为 int，这样才能计算
        break; // 末尾的 break 可省略
}
cout << result; // 试图输出 result 的值
// 'result' was not declared in this scope
```

这段代码如果真的运行的话，编译器可能会报大量的错误。我以注释的方式将其中一条放在这个代码中，它的含义是“`result` 在这个作用域中没有定义”。

这是因为我们遇到了老问题，`switch` 内部是一个内层作用域，我们在内层作用域中定义的 `result` 只在 `switch` 的作用域内有效，一旦离开这个作用域，`result` 的生存期就结束了，这个变量失效。于是编译器就会报告这样的错误。

解决这个问题的方法有两种，一是直接不定义 `result` 了，在每个 `case` 之下直接输出；二是在外层作用域中定义变量 `result`（这时就不能再定义成 `const` 了，因为我们要使用赋值的方式来存储计算结果）。这里我们采用第二种方法。

最后，我们希望这个计算器可以多次运行。我们指定，当用户提供一个非预期的值（即，每行的开头不是 `num1` 预期得到的数字，而是字母或者别的什么，比如 `q`）时程序结束，否则就可以反复循环运行。

这里可以用 `cin` 的状态作为 `while` 循环的条件来判断循环是否要终止，而在 `cin` 之后必须检测一次 `cin` 的状态，如果不正常，就要 `break`。

```
while (cin) { //如果cin状态正常，就进行循环（非必需，改成true也可）
    double num1, num2, result; //定义两个操作数num及结果值result
    char op; //定义一个字符用来存储运算符输入
    cin >> num1 >> op >> num2; //依次输入左操作数、运算符和右操作数
    if (!cin) { //如果cin状态不正常，说明输入了非预期内容，退出循环
        break;
    }
    //switch-case结构部分省略
    cout << "=" << result << endl; //输出result的值
}
cout << "程序结束" << endl; //提示用户程序结束
```

那么我们的思路就比较清晰了。现在让我们把头文件包含等等杂七杂八的代码加上，就可以形成一个完整的程序了。

代码 3.5: 简易计算器.cpp

```
//这是一个简易计算器，可以支持简单的四则运算或模运算。不支持复杂表达式的计算
#include <iostream>
using namespace std;
int main() {
    while (cin) { //如果cin状态正常，就继续循环
        double num1, num2, result; //定义两个操作数num及结果值result
        char op; //定义一个字符用来存储运算符输入
        cin >> num1 >> op >> num2; //依次输入左操作数、运算符和右操作数
        if (!cin) { //如果cin状态不正常，说明输入了非预期内容，退出循环
            break;
        }
        switch (op) { //判断op是什么
            case '+': //根据op的值，采取相应的操作
                result = num1 + num2; //通过赋值的方式存储结果
                break; //记得用break
            case '-':
                result = num1 - num2;
                break;
            case '*':
                result = num1 * num2;
                break;
            case '/':
                result = num1 / num2;
        }
        cout << "=" << result << endl;
    }
    cout << "程序结束" << endl; //提示用户程序结束
}
```

```
        break;
    case '%':
        result = static_cast<int>(num1) % static_cast<int>(num2);
        // 显式类型转换，将double型的num1和num2转换为int，这样才能计算
        // 末尾的break省略
    }
    cout << "=" << result << endl; // 输出result的值
}
cout << "程序结束" << endl; // 提示用户程序结束
return 0;
}
```

读者可以自行测试此计算器的效果。

# 第四章 函数初步

在前面的章节中我们讲过了数据、运算符、语法/语义、语序，还有流程控制。从细枝末节到整体结构，我们可以一窥编程大厦的样貌。笔者犹记得，自己当时只学了结构控制，就拿着 C++ 来写命令行界面小游戏了——其实不难，无非就是人机战斗，用户给个输入，用选择结构判断该做什么，由程序来算几个四则运算，再给出输入。最后外面套个 `while(true)` 循环，一个自制小游戏就这么做完啦。

仅有我们此前所学的知识已经足够做出一个完整的、有交互功能的程序，只是工作量会大一点而已。但是别高兴太早，当你真正去实践编程的时候就会发现，要真正实现复杂功能，还是需要极高的工作量，并且其中的许多工作是无意义重复的。在我的游戏当中，我选择用 `Ctrl+C/V` 来解决这个问题，但这还是很麻烦。而且，如果我发现这部分的内容需要调整，我就要把所有相关代码全改一遍！

本章我们先来解决这些重复代码的问题，我们的选择是函数（Function）。

## 4.1 函数的概念

什么是函数？它是数学意义上的函数吗？

在数学意义上，一个函数  $f$  是从一个集合  $\mathbb{X}$ （值域）到另一个集合  $\mathbb{Y}$ （到达域）的某种映射关系<sup>1</sup>。对于  $\mathbb{X}$  集合中的某个元素  $x$  来说，总有唯一确定的  $\mathbb{Y}$  集合中的元素  $y = f(x)$  与之对应。举例来说，我们记  $f(x) = x^2 + 2x + 1$ ，其中  $x \in \mathbb{R}$ ，那么对于某个输入  $x = 3$ ，就能通过这个函数算出  $f(3) = 16$  来，这个结果就是函数的输出。

三角函数也是数学意义上的函数。比如正弦函数，它的名字就是  $\sin$ 。不过它不像我们刚才用到的  $f$  那样，我们不知道它的表达式，如果你要问我  $\sin 1$  是多少，我也不方便徒手计算，要用计算器才行。但是无论何时何地，我算出的  $\sin 1$  和张三算出的  $\sin 1$ 、李四算出的  $\sin 1$  都是一样的。所以说， $\sin$  的输入信息和输出信息之间有确定的对应关系。我不知道  $\sin$  的内部构造，也不知道计算器是用什么原理来实现的，但是我给它某个输入，它就会给我对应的输出，这就是一个黑盒（Black box）（见图 4.1）。



图 4.1:  $\sin$  函数就是一个黑盒，给定某个输入信息就会有对应的输出信息

在编程当中，函数的概念可以借用数学上的函数概念来阐释。举个例子，`cmath` 库中有一个函数 `sin`，它可以接收一个 `double` 类型的参数，并返回一个 `double` 类型的结果。所以我们可以这么说它的定义域  $\mathbb{X}$  就是 `double` 型的所有数据，到达域  $\mathbb{Y}$  也是 `double` 型的所有数据。

<sup>1</sup>这里介绍数学意义上的函数，只为引入编程意义上的函数概念。如果读者拥有相关数学基础，自然更好；即便没有相关数学基础，也不妨碍对后续知识的理解和应用。

```
#include <iostream> // 标准输入输出头文件
#include <cmath> // sin函数定义在cmath库中
using namespace std; // 使用命名空间std
int main() {
    cout << sin (3.14159); // 使用sin函数求出sin(3.14159)的值
    return 0;
}
```

这段代码的输出是

---

2.65359e-06

---

它是一个非常小的数，这是合乎我们的预期的。<sup>2</sup>

数学上还有多元函数的概念，也就是接收一个或多个输入，并给出一个输出的函数。在程序设计中，函数也可以接收一个或多个输入，甚至接收空输入（我们在前面用到的 `numeric_limits<int>::max()` 就是一个空输入的函数）。但是函数只能返回一个输出<sup>3</sup>或者返回 `void`<sup>4</sup>（即空输出）。我们在下一节中就会提到这些。

我们熟识的运算符就可以看成是一种函数<sup>5</sup>。加法运算符就是一个黑盒，它的两个操作数就是两个输入信息，而返回值（输出信号）就是它们的和。运算符中有操作数和返回值的概念，这些概念对应到函数中就是参数和返回值。

C++ 的各种库为我们提供了许多函数，我们可以直接使用这些函数来实现我们想要的功能，而不需要自己把每个功能实现出来。比如说 `cmath` 库中有丰富的数学函数，除了刚才的正弦函数以外，还有求绝对值函数 `abs`、开平方函数 `sqrt`、下取整函数 `floor` 等。如果真的要我们自己写代码实现的话，这是非常麻烦的（试想，如何用一些结构控制语句和四则运算来求开平方呢？这好像很难办到）。

而 C++ 的函数封装了一些已经实现好的功能，我们不再需要考虑如何写一个开方代码，思路是什么，需要注意哪些细节——直接用 `cmath` 库给我们的解决方案就好了。

```
double num; // 定义一个num用来接收输入
cin >> num; // 输入num的值
cout << sqrt (num); // 输出num开方后的值
```

我们还可以自己定义函数。比如在[代码 3.4](#)中，我提供了一个 `input_clear` 函数，用来清除 `cin` 的错误状态并清理本行输入。这个函数在其它地方也可以用<sup>6</sup>，只需要写一个 `input_clear();` 就可以了，而不需要把函数体当中的那三行代码再写一遍——这样就很方便了。

而且更重要的是，如果我有朝一日不使用 `cin` 而是用别的来进行输入（还记得吗？`cin` 只是 `istream` 类的一个对象，我们还可以使用这个类或者派生类<sup>7</sup>的其它对象来进行输入），那么我只需要在 `input_clear` 的定义中修改一次即可。但如果我不用函数而是每次都写那三行代码呢？修改起来也会相当麻烦。

使用函数还有一个优点，我们可以把一个复杂的大任务拆分成几个简单的小任务。其实在之前写代码的过程中，我们也是有意把程序的思路梳理清楚的，这个功能应该怎么做，那个功能应该怎么做。而函数看上去更加直观，我们为每一项操作起了一个名字，然后分别把它们写到不同的代码中

<sup>2</sup>注意，`cmath` 库中的 `sin` 函数接收的参数是按照弧度值传入的，所以  $\sin 3.14159 \approx 0$ 。

<sup>3</sup>如果我们需要让返回多个值，可以使用结构体 `struct` 或者 `std::tuple`。我们将来会接触到这种情况的。但这并不违反“至多只能返回一个输出”的规定，因为一个结构体或 `tuple` 就是单个输出。

<sup>4</sup>`void` 也是一个基本数据类型，但它不能用来定义数据，可以用来定义指针。在函数中，我们会经常见到这种类型的。

<sup>5</sup>实际上运算符和函数并不相同，不过它们有很多相似之处，可以用于类比。

<sup>6</sup>前提是需要定义，或者声明并在有关的代码中提供了定义。详见下一节。

<sup>7</sup>有关继承和派生类的概念，我们将在第九章介绍。

去，这样就方便很多了。

接下来我们就开始学习如何定义和使用函数，以后我们会频繁用到的。鉴于我们还有很多复合数据类型相关的知识没有学习，本章只是对函数作一些初步讲解。

## 4.2 函数的定义和使用

### 基本技术

定义一个函数的基本语法是：

```
<返回类型> <函数名>(<参数列表>) {
    <若干操作>; //花括号中的内容又叫函数体
}
```

其中 <返回类型> 可以是任何一个类型，也可以是 `void`（空）类型。<参数列表> 可以接收若干个参数，也可以不接收参数。一般来说，每个函数的参数都是固定数量的，不过也存在“变长实参”的情况，我们暂不讨论。

而函数体内可以进行各种操作，比如赋值，选择、循环等等。注意：如果返回类型不为空，那么函数体内必须有 `return` 返回值，否则将发生未定义行为，后果难料。

我们在此之前已经接触过一些函数，除了 `numeric_limits<int>::max()` 和 `input_clear` 以外，还有一个我们一直忽视的函数，那就是 `main`。主函数也是一个函数，它的返回类型必须为 `int`，可以接收参数<sup>8</sup>。因为它的返回类型为 `int`，所以它也需要有返回值，我们一般写作 `return 0;`，这样你就明白之前的那些代码中为什么总要加上这句了吧<sup>9</sup>。

主函数是程序运行的起点。程序运行时会自动从主函数的第一句开始向下运行，直到主函数结束，或者遇到了返回语句 `return`。而其它的函数不会直接被使用，我们必须手动使用。下面我们就来尝试一个最简单的例子。现在我们试着自己来写一个求绝对值的函数  $f(x) = |x|$ ，它可以允许对浮点数求绝对值。让我们想一想，这个函数的定义要怎么写？

首先确定返回类型和参数。这个函数允许浮点数的计算，所以返回类型肯定也要能够表示浮点数——任何一种浮点类型都能做到，所以我们就用 `double` 好了。

接下来确定参数。这个函数只接收单个浮点数输入，所以我们只需要一个 `double` 型的参数，起名为 `x`。

函数体内的部分需要分情况处理，如果  $x \geq 0$ ，那么直接返回原数就可以；如果  $x < 0$ ，那么就需要返回它的负值。既然涉及到分情况处理，那当然要用选择结构来实现啦。

```
double f(double x) { //返回类型为double，接收一个double型参数x
    if (x >= 0) { //如果x>=0，返回x本身
        return x;
    }
    else { //如果x<0，返回x的负值
        return -x;
    }
}
```

接下来我们就可以使用这个函数来求绝对值，而无需每次用到时都反复写好几遍代码。这个“使用”过程又叫作调用（Call）。

<sup>8</sup>但是，主函数接收参数的情况不同于其它函数，这里不作细致讨论。我们也不常使用这种写法。

<sup>9</sup>实际操作中，如果没有在 `main` 函数中写返回语句，编译器会为我们自动添加。所以取决于编译器的具体情况，我们可以不必在 `main` 函数中写返回值。

```

int main() {
    cout << f(-.1) << endl; //输出-0.1的绝对值，然后换行
    for (int i = -2; i <= 2; i++) { //输出-2到2每个数的绝对值
        cout << f(i) << ' ';
    }
    return 0;
}

```

程序的运行结果如下：

---

```

0.1
2 1 0 1 2

```

---

第一行是 `-0.1` 的绝对值，看起来确实达到我们的要求了。第二行是一个 `for` 循环，先定义了 `int` 型的 `i`，然后让它从 `-2` 自增到 `2`，输出 `i` 的绝对值。程序的结果看上去也是正确的。

但细心的读者可能会发现一个问题：我们定义的 `f` 接收的参数类型是 `double`，但是为什么 `f(i)` 不会报错呢？这里的 `i` 不是 `int` 类型吗？

其实这是因为隐式类型转换在起作用。还记得我们当时的说法吗？如果某处期望一个类型，但代码中提供的是另一个类型的数据，编译器可能考虑对此数据进行隐式类型转换，以符合类型需要<sup>10</sup>。所以实际传入的参数并不是 `i` 本身，而是 `i` 隐式转换为 `double` 之后的数据。

## 函数的作用域

我们在先前简单介绍过作用域的概念，凡是局部变量都有它的生存期，一个作用域结束后，它的生存期宣告终止，我们不能再使用这个变量。之前那些都属于块作用域（Block scope），比较容易理解。这里我们再谈谈函数有关的作用域。

每一个函数体都是一个独立的作用域，各个函数作用域之间互相没有嵌套关系——这点和块作用域很不相同。在讲块作用域的时候我们说，内层作用域中可以使用外层作用域里定义的变量。但是函数作用域彼此独立，一个函数中定义的变量在另一个函数中不可见<sup>11</sup>，如图 4.2 所示。

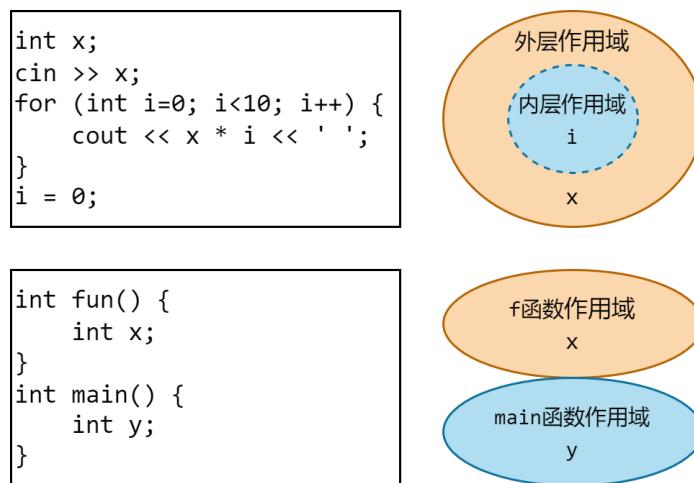


图 4.2: 块作用域与函数作用域的区别

<sup>10</sup>仍要提醒读者，这是不完全的。比如，`double` 类型的取模是不允许的，这时编译器不会尝试将其转换为 `int` 型。

<sup>11</sup>一个函数作用域的变量对另一个函数不可见，这并非意味着生存期结束且变量被销毁。我们在第七章中会再谈及此类问题。

这种关系有点像是动物园。在动物园中，每个动物（局部变量）都被关在自己的围栏（函数作用域）内生活，而其它围栏内的动物不能和它们互相接触。动物饲养员则像全局变量，它可以进入各个围栏内部。

那么假如我需要在两个函数中传递信息（比如，在主函数中告诉 `f` 函数，需要求哪个数的绝对值），我该怎么办呢？这时就需要用到参数传递和返回值了。

## 实参与形参

回顾一下我们刚才写的自定义绝对值函数 `f`。

```
double f(double x) { // 返回类型为 double，接收一个 double 型参数 x
    if (x >= 0) { // 如果 x>=0，返回 x 本身
        return x;
    }
    else { // 如果 x<0，返回 x 的负值
        return -x;
    }
}
```

这里的 `x` 是它的一个参数。我们在调用 `f` 时，程序会新定义一个 `x`，而括号内的数据会作为初始化信息传递给 `x`。举例说，`f(-.1)` 在主函数中被调用，这时程序会新定义一个 `x` 并初始化为 `-.1`，然后开始顺序执行 `f` 中的代码。换言之，每次调用 `f`，程序都会重新定义一个 `x`，所以我们可以把参数列表当作是“函数中的一种变量定义语句”。

那么如果我把这个函数中的所有 `x` 都换成 `y`，这个函数的本质发生变化了吗？

```
double f(double y) { // 返回类型为 double，接收一个 double 型参数 y
    if (y >= 0) { // 如果 y>=0，返回 y 本身
        return y;
    }
    else { // 如果 y<0，返回 y 的负值
        return -y;
    }
}
```

答案当然是不会！名字是可以随便起的，无论用 `x` 还是 `y`，还是换成 `num` 或者 `arg`，都没问题。但是，只要函数的逻辑不变，它的本质就没有发生变化，对于相同的输入，永远会给出相同的输出。数学上也是如此，对于相同的定义域来说， $f(x) = |x|$  和  $f(y) = |y|$  当然是同一个函数。

真正会改变函数值的不是参数的名字，而是具体传入函数的输入信息。`f(3)` 和 `f(4)` 当然是不同的；`f(3)` 和 `f(-3)` 虽然相同，但它们传递的信息还是有差异的，只是输出恰好一致。所以说，同样是“参数”，`x` 或 `y` 仅仅代表一个名字，而 `3` 或 `4` 却能传递真正有效的信息，看来它们还是有一定的区别的。

我们把 `x` 和 `y` 这类在函数定义过程中用到的名字称为形参（Parameter/Argument），而把 `3` 和 `4` 这类在函数调用过程中传递了真实信息的数据称为实参（Argument/Parameter）。<sup>12</sup>请注意，实参未必就“没有名字”，它可以是字面量、常量表达式、常量或变量；形参和实参的区别应该在于：你是在定义还是在调用。

<sup>12</sup>parameter 和 argument 这两个词都可以用来指代参数。不过习惯上，parameter 指代形参，而 argument 指代实参。

## 函数的使用

举个例子，现在我们要写一个 `max` 函数，它可以接收两个整数输入，返回值是其中较大的那个整数。

首先确定返回类型和参数。这个函数只需接收整数参数，返回整数值，所以返回类型可以是任何一种整型，具体哪种就看我们需要了。为了能涵盖比较大的数据范围，我们在这里可以选择 `long long`。我们需要两个参数，它们都是 `long long` 类型的。名字可以随便起，比如 `a` 和 `b` 吧。

函数体内的部分需要返回 `a` 和 `b` 的较大值，这就需要我们分析一下如何实现：如果 `a` 大于 `b`，那我们就返回 `a`，这没问题；如果 `a` 不大于 `b`，那我们就返回 `b`，也不会出错。所以我们需要一个选择结构。我们可以这么写：

```
long long max(long long a, long long b) {
    //接收两个long long参数，给出long long返回值
    if (a > b) { //当a>b时返回a
        return a;
    }
    else { //否则返回b
        return b;
    }
}
```

接下来写一下主函数的部分就可以了。这是我们熟知的内容，所以我只放代码，不再讲解思路。

```
int main() {
    long long a, b; //定义int型变量a,b
    cin >> a >> b; //输入a,b的值
    cout << max(a, b); //调用max
    return 0;
}
```

请留意：主函数中的 `a` 和 `b` 与 `max` 函数中的 `a` 和 `b` 是完全不同的概念。它们分别属于不同的、相互独立的作用域，互不干涉。

如果我们要输入三个数并找出其最大值呢？一种方法是，先用 `max` 求出其中两数的最大值，然后再求出它和第三个数的最大值。

```
int main() {
    long long a, b, c; //定义三个int型变量a,b,c
    cin >> a >> b >> c; //输入a,b,c的值
    cout << max(max(a, b), c); //输出它们三个的最大值
}
```

我们来解析一下 `max(max(a,b),c)` 这段代码，不难：

首先，`max(a,b)` 先行计算，返回的是 `a, b` 中的最大值。这个返回值又作为 `max` 的参数与 `c` 比较，最后返回的就是 `a, b, c` 中的最大值。

当然还有别的方法，我们也可以定义一个 `max3` 函数<sup>13</sup>，接收三个参数并求出它们的最大值。

```
long long max3(long long a, long long b, long long c) {
    //接收三个long long参数，给出long long返回值
    const long long max_ab {max(a,b)}; //先求出a,b的最大值
    if (max_ab > c) { //如果a,b的最大值大于c，返回前者
}
```

<sup>13</sup> 我们很快就会讲到重载，那时就可以直接重载一个 `max` 函数。

```

        return max_ab;
    }
    else { // 否则返回后者
        return c;
    }
}

```

这样我们就可以用 `max3` 函数来求三个数的最大值了。

初学者可能有这样的误会，认为只有主函数才能调用其它函数。不是的，所有函数都可以调用其它函数，只要有声明（我们稍后谈到）。函数可以调用它自身，这就是递归<sup>14</sup>。其它函数甚至可以调用主函数，但这往往容易引起困惑，所以我不推荐这么做。

如果涉及大量数据求最大值的问题，这种方法就显得不太合适，这时我们可能考虑寻求 `valarray` 等其它方案，这些我们之后再谈。

## 函数的声明

有些程序员在写代码的时候更倾向于把主函数写在靠前的位置上，这样一来，程序的主要操作是什么就一目了然——毕竟，主函数是程序开始执行的地方。

不过如果单纯只是这么写的话，编译器就会报错。以下是例子：

```

int main() {
    long long a, b, c; // 定义 long long 型变量 a,b,c
    cin >> a >> b >> c; // 输入 a,b,c 的值
    cout << max3(a, b, c); // 输出 a,b,c 的最大值
//error: 'max3' was not declared in this scope
}

long long max(long long a, long long b) {
    // 省略
}

long long max3(long long a, long long b, long long c) {
    // 省略
}

```

编译器报错信息的含义是：“`max3` 在这个作用域中没有定义。”读者可能会误会，认为把 `max3` 定义在 `main` 函数中就可以了。但是，传统意义上的一个函数不能定义在其它函数当中，而必须定义在全局作用域中<sup>15</sup>！这些函数之间的关系是并列的，而非 `max3` 从属于 `main`。

Visual C++ 编译器返回的报错信息更清晰：“error C3861: 'max3': identifier not found。”它的意思是：“没有找到 `max3` 这个标识符。”这是因为，编译器处理代码是从前向后处理的。它遇到一个函数声明（或定义）就会记住；然后一旦遇到函数调用，就去寻找能匹配这个调用的函数。现在我们把 `max3` 的调用放在前，而 `max3` 的定义放在后，那么编译器自然会“not found”了。

我们既想要把长篇累牍的函数体放在后面以突出重点（主函数），又不得不在主函数之前把“存在这样一个函数”的信息告诉编译器，那么最好的解决方案就是声明（Declaration）了。

所谓函数的声明，就是把一个函数的外部特征告诉编译器，这样编译器就可以在调用相应函数时匹配到它。至于这个函数会做什么，先不要管它，等到定义（Definition）中再告诉编译器。

声明一个函数的基本语法是

```
<返回类型> <函数名>(<参数类型列表>); // 注意用分号结尾
```

<sup>14</sup> 我们会在后面的内容中讲到递归。

<sup>15</sup> 我们在精讲篇中会介绍 lambda 闭包，它是一种可以在局部作用域中定义的函数。

简单说来就是，把花括号包起来的函数体部分去掉，然后用一个分号结束该句即可。

在函数声明的语法中，我们只需要列出参数的类型，而不需要指明参数的名称——它只是个形参而已，定义成什么名字其实无所谓，而且我们又无需在函数体（因为没有写）中使用它，所以仅仅指明类型，让编译器知道它该接收什么参数就足够。

如果我们要为前面定义的 `f` 写一个声明，可以这么写：

```
double f(double); // 形参名可省略，或者任意取名
```

如果我们要为 `max` 和 `max3` 写声明，可以这么写：

```
long long max(long long, long long);
long long max3(long long, long long, long long);
```

所以我们把代码整理一下，就可以形成一个完整的程序：

代码 4.1: max3.cpp

```
// 这个程序可以接收三个输入整数，并给出它们的最大值
#include <iostream> // 标准输入输出头文件
using namespace std; // 使用命名空间 std
long long max3(long long, long long, long long); // 声明
int main() {
    long long a, b, c; // 定义 long long 型变量 a,b
    cin >> a >> b >> c; // 输入 a,b 的值
    cout << max3(a, b, c); // 输出 a,b,c 的最大值
    return 0;
}
long long max(long long a, long long b) {
    // 接收两个 long long 参数，给出 long long 返回值
    if (a > b) { // 当 a>b 时返回 a
        return a;
    }
    else { // 否则返回 b
        return b;
    }
}
long long max3(long long a, long long b, long long c) {
    // 接收三个 long long 参数，给出 long long 返回值
    const long long max_ab {max(a,b)};
    if (max_ab > c) { // 如果 a,b 的最大值大于 c，返回前者
        return max_ab;
    }
    else { // 否则返回后者
        return c;
    }
}
```

这里有个问题需要读者留意：为什么这里只需要声明 `max3` 而不需要声明 `max` 呢？其原因在于，`max` 是在 `max3` 中调用的，而 `max` 的定义在 `max3` 之前，所以编译器不会找不到 `max`，参见图 4.3 的蓝色箭头。

```

unsigned factorial(unsigned n) {
    if (n == 0){
        return 1;
    }
    else {
        return n * factorial(n - 1);
    }
}

```

图 4.3: main 函数调用了 max3, 而 max3 调用了 max, 它们都能够找到对应的函数

这也就是说, 其实 main 函数没有调用 max, 调用 max 的其实是 max3。这种关系对于初学者来说可能相当复杂, 不过即使现在很难理解也没有关系, 我们在日后的学习过程中会慢慢体会到的。

### “值计算”与“副作用”

用函数的视角来理解运算符是一种很重要的思想。C++ 中的自增/自减运算符可以当作一个函数来看待, 它接收某个类型(实际上它可以接收很多类型)的参数, 然后返回同类型的参数。如果用于后缀形式, 它的返回值是原数; 如果用于前缀形式, 它的返回值是原数加/减 1 后的值<sup>16</sup>。

然而它的作用不仅仅停留在“提供一个返回值”, 它还有改变原数(参数)的功能。自增/自减作用于某个变量, 会导致这个变量增加/减少 1。这类功能不是通过返回值表现出来的, 我们称它为副作用(Side effect); 而对应地, 把通过返回值表现出来的功能称为值计算(Value computation)。

有些时候, 比如在 `for` 循环的迭代操作中, 我们使用自增/自减运算符是为了使用它的副作用, 而不是值计算。对于函数来说也是同理。还记得我之前提供的 `input_clear` 函数吗? 它的返回类型是 `void`(空), 我们不需要接收它的什么返回值, 而只需要使用它的副作用(清除错误状态, 清除本行输入)就行。

```

void input_clear() {
    cin.clear(); // 清除错误状态
    while (cin.get() != '\n') // 清除本行输入
        continue;
}

```

我们还会自定义很多这类的“只用其副作用而不用其值”的函数, 这种情况下我们可以将返回类型定为 `void`。`void` 函数中可以写 `return`; 或者 `return void{};`<sup>17</sup>这样的返回语句, 或者压根什么返回语句也不写, 没有问题。

## 4.3 函数重载

C++ 中的运算符可以接收不同类型的操作数。`.1+.2`, 这里的操作数都是 `double` 类型的; `1+2`, 这里的操作数都是 `int` 类型的; `1ll+2ll`, 这里的操作数都是 `long long` 类型的(还记得吗, 整数带 `ll` 后缀将是 `long long` 类型的)。我们希望同一个函数也能像运算符那样, 支持多种类型。比如 `max` 吧, 它是否既可以支持 `double` 类型又可以支持 `int` 类型呢?

<sup>16</sup>这里忽略整数溢出和浮点精度问题。

<sup>17</sup>`void{}`相当于一个 `void` 类型的临时对象, 我们会在后面经常遇到这类语法的。

这种可能性是完全存在的。想想吧，当我们调用函数时，编译器能知道的信息只有函数名和实参——而实参能够提供“参数类型”这则信息。因此我们如果定义一个“接收不同类型参数”的同名函数，编译器仍然可以分得清。

当然这只是可行性分析，不代表该语言能允许我们这样做——C 语言就不支持这种方法。不过好消息是，C++ 支持这种方法，它叫做**函数重载（Function overloading）**。

要重载一个函数是非常简单的，只要你用同一个函数名写不一样的参数列表就行了。举例来说，我们在上一节中定义了 `max3`，其实我们可以直接用 `max` 这个名字，它就是对原来 `max` 的重载。

```
long long max(long long, long long); //本行代码是非必需的
long long max(long long, long long, long long); //此重载max接收三个long long参数
int main() {
    long long a, b, c;
    cin >> a >> b >> c;
    cout << max(a, b, c); //编译器自动找到max(long long,long long,long long)
    return 0;
}
long long max(long long a, long long b) {
    //省略
}
long long max(long long a, long long b, long long c) {
    //省略
}
```

我们还可以重载 `double` 版本的两数 `max` 函数和三数 `max` 函数，只需要在上面的代码中再添加：

```
\\"声明部分如下
double max(double, double); //此重载接收两个double参数
double max(double, double, double); //此重载接收三个double参数
\\定义部分如下
double max(double a, double b) {
    if (a > b) {
        return a;
    }
    else {
        return b;
    }
}
double max(double a, double b, double c) {
    const double max_ab {max(a,b)};
    //这里编译器会调用max(double,double)而非max(long long,long long)
    if (max_ab > c) {
        return max_ab;
    }
    else {
        return c;
    }
}
```

图 4.4 即 `max` 函数的四种重载方案。每种重载方案都可以视为一个独立的函数，只是它们名字不同而已。

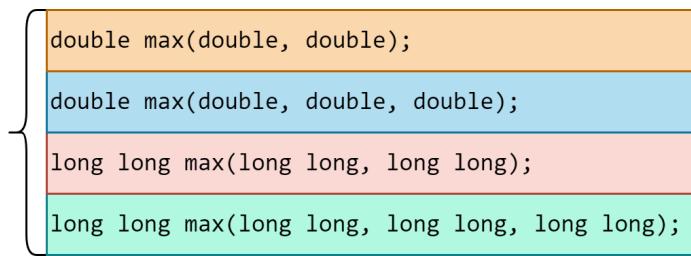


图 4.4: `max` 函数的重载

注意，同一个函数的返回类型不能重载。我们写成这样是不行的：

```

// 声明部分如下
double max(double, double);
long double max(double, double);
//error: ambiguating new declaration of 'long double max(double, double)'

```

编译器的报错信息意为：“`long double max(double, double)` 的定义会引发歧义。”这不难理解。因为在调用函数的时候我们无法指定这个函数的返回类型，所以一旦我们定义了两个完全相同的函数——除了它们的返回类型不同以外，编译器就不可能把它们分开。这就是一种二义性（Ambiguity）。

二义性说白了就是歧义。我们日常使用的语言不可避免地会有歧义，但大多数时候我们都能根据实际情况来判断状况，而不致造成错误理解（但这种情况还是时常发生，不是吗）。编译器不具备人的理解能力，它面对二义性问题时无法判断程序员的真实意图，所以只会报错。

在涉及到重载时，还有很多种情况会导致二义性的发生，比如这个：

```

void fun(long long); // 声明了一个接收 long long 型变量的函数 fun, 定义省略
void fun(double); // 声明了一个接收 double 型变量的函数 fun, 定义省略
int main() {
    fun(1ll, 1.); // 实参分别是 long long 类型和 double 类型, 如何是好?
// (GCC) error: call of overloaded 'fun(long long int, double)' is ambiguous
// (MSVC) error C2666: 'fun': overloaded functions have similar conversions
// (clang) error: call to 'fun' is ambiguous
}

```

先不看编译器的报错信息，我们来分析一下这段代码的行为。我们重载了 `fun` 函数，其中一个版本可以接收 `long long` 类型的参数，而另一个版本可以接收 `double` 类型的参数。在主函数中，我向 `fun` 中传入了两个参数，分别是 `long long` 型的 `1ll` 和 `double` 型的 `1.`。现在问题来了，编译器究竟是会把 `1ll` 隐式转换成 `double` 类型，还是会把 `1.` 隐式转换成 `long long` 类型呢？

编译一下就会发现，无论是 GCC，Visual C++ 还是 Clang，都会给出类似于歧义的报错。其中 MSVC 的报错提到了“have similar conversions”，这正是问题的根源：“把 `1ll` 转换为 `double`”或者“把 `1.` 转换成 `long long`”都是可行的，这两种方案高度相似，编译器也无法决定怎么转换，于是就出现了二义性问题。

怎么解决呢？我们可以使用强制类型转换的方法来把一个参数转换成别的类型，比如都转换成 `double`，这样编译器就能直接找到匹配的函数了。

还有一种方式就是再定义一个 `fun(long long, double)` 重载函数。但是这样就别提有多麻烦了——那我们是不是还要考虑定义一个 `fun(double, long long)`，以防有人使用 `fun(1., 1ll)` 呢？这

样的麻烦无穷无尽, 那是很烦人的。<sup>18</sup>况且, 因为 `int` 既可以转换为 `double` 又可以转换为 `long long`, 那么在调用 `max(1,2)` 时也会出现“编译器不知该使用哪个重载”的二义性问题, 那么我们还要再定义 `int max(int,int)` (以及各种变形) 呀?

不过这仅是出于语法层面的考量而言的。实际编程中我们大多会使用同一类型的数据来求最大值, 所以只需要定义两(三)个同类型参数的重载, 就足以应付大多数需求了。至于更细致的知识, 尤其是有关编译器如何处理“类型转换”和“匹配哪个函数”的问题, 其实非常复杂, 本章不再赘述。我们会在泛讲篇中更深入地探讨这个问题。

## 4.4 默认参数的设置

在数学上, 对数函数  $\log_a N$  是一个二元函数, 其参数分别为  $a$  和  $N$ 。不过在数学当中, 我们一般都用  $a = e$  作为对数的底。因此为了方便, 我们就约定俗成, 不写底数的时候默认以  $e$  为底, 写成  $\log N$ 。有些数学文献中也用  $\ln N$  代替底数为  $e$  时的对数函数。

$n$  次方根函数  $\sqrt[n]{a}$  也是一个二元函数, 其参数分别为  $n$  和  $a$ 。我们最常用的是二次方根, 所以当  $n = 2$  时, 我们习惯省略它, 直接写作  $\sqrt{a}$ 。

以上是在数学中常见的。而在 C++ 语言中, 我们也可能遇到类似的问题——某个函数的某个参数在多数时候都是某个特定值, 我们能否想一种方法, 省略掉一些常常为特定值的参数, 这样我们就不必在每次调用此函数时都写这个参数了呢?

答案是肯定的。例如我们要写一个开  $n$  次方根函数, 我们就可以用上一节提到的重载方法, 写一个接收  $a$  和  $n$  的 `root` 函数, 再写一个只接收  $a$  的 `root` 函数(将  $n$  当作 2)。鉴于 `root` 函数本身的实现比较麻烦, 我们退而求其次, 使用 `pow(a,1./n)`<sup>19</sup> 来实现它。

```
#include <iostream>
#include <cmath>
using namespace std;
double root(double, int);
double root(double);
int main() {
    cout << root(64, 3) << endl; // 将调用 root(double,int)
    cout << root(64) << endl; // 将调用 root(double)
    return 0;
}
double root(double radicand, int index) {
    return pow(radicand, 1. / index); // 注意 index 是 int 型, 所以要用 1./index
}
double root(double radicand) {
    return pow(radicand, .5); // 默认开平方根, 就可以直接用 0.5 次幂表示
}
```

关于函数重载, 读者已经比较熟悉了, 所以这里我就不再对代码作过多解释。这里的 `root(double)` 也可以定义成下面这个样子, 效果相同:

```
double root(double radicand) {
    return root(radicand, 2); // 利用 root(double,int) 的返回值
}
```

<sup>18</sup> 所以你就能理解为什么我们需要泛型编程了, 详见本章第六节。

<sup>19</sup> `double pow(double base, double exp)` 是 `cmath` 库中的一个函数, 返回值为 `base` 的 `exp` 次幂。这个函数还重载了其它浮点类型的版本, 但我们在那里不需要它们。

除了重载之外，我们还可以使用另一种方法：为函数设置默认参数（Default argument）。

当我们在声明或定义一个函数的时候，我们可以为它设置一些默认值。这样，当我们调用函数的时候，如果把对应的参数留空，编译器就会用默认值来替代它。以下是一个例子：

```
double root(double, int = {2}); // 等号不能省略，花括号并非必需，但建议使用
int main() {
    cout << root(64, 3) << endl; // 将调用 root(double,int)，第二个参数传入3
    cout << root(64) << endl; // 将调用 root(double,int)，第二个参数取默认值2
    return 0;
}
double root(double radicand, int index) {
    return pow(radicand, 1. / index); // 注意 index 是 int 型，所以要用 1./index
}
```

看起来函数重载和默认参数这两种方法都可以达到我们的目的，那么它们的区别是什么呢？

其实这两种方法最大的区别在于，函数重载是“定义了两个函数”，而设置默认参数“不会定义新函数”！在函数重载的情况下，我们计算 `root(64,3)` 时，调用的是 `root(double,int)`，而计算 `root(64)` 时，调用的是 `root(double)`；但如果只是设置了默认参数，那么无论计算 `root(64,3)` 还是 `root(64)`，都会调用 `root(double,int)` 这个函数。如果我们只是想设置默认参数而无需做大的细节改动，那么用默认参数的方法当然比重载要方便了（我们无需重新定义一个函数！）。

而函数重载应该用在更需要处理细节改动的情况下，比如两个数求最大值和三个数求最大值，函数内部的细节是不太相同的——这时若用默认参数就显得更麻烦了，而不是更简单。

在默认参数方面，其实还有很多需要留意的内容。不过大部分问题不常见，我在泛讲篇中就不多说了，只说几个小的要点。

其一，默认值最好是在函数声明时就规定出来，而不是等到定义时再规定。这是因为，编译器遇到函数调用的时候，会向前找寻函数声明。只有在函数声明中规定了默认值的情况下，编译器才会考虑匹配；如果在函数声明中没有规定默认值，编译器就找不到这个函数（或者好巧不巧地匹配上了其它同名函数，但那不是我们想要的），即便我们后面的函数定义中再规定默认值也是枉然！

其二，有默认值的参数必须在函数参数列表的右侧。这可能有点费解，我举个例子你就懂了。

```
// 假设这四个函数定义在不同场合下，互不影响
double root(double, int = {2}); // 允许
double root(int = {2}, double); // 这是禁止的！
// error: default argument missing for parameter 2 of 'double root(int, double)'
double root(double = {1.}, int = {2}); // 允许
double root(int = {2}, double = {1.}); // 允许
```

编译器的解释方式是，但凡某个参数有默认值，从它开始到后边的所有参数都应该有默认值。因此，我们定义函数时，必须要把有默认值的参数放在参数列表的右侧。

我们在先前接触过一个自定义函数 `input_clear()`，它可以清理错误输入。我们最常用 `istream` 类的 `cin` 对象来进行输入，但这不代表我们在其它时候不需要用别的对象。为此，我们可以改一下 `input_clear` 函数，让它可以接收任何一个 `istream` 类的对象<sup>20</sup>。而鉴于我们常用 `cin`，所以不妨把它设为默认参数。

```
// 不单独声明，直接定义
void input_clear(istream &in = {cin}) { // 不提供参数时使用默认参数 cin
    in.clear(); // 清除错误状态
    while (in.get() != '\n') // 清除本行输入
```

<sup>20</sup>事实上还可以接收 `istream` 派生类的对象，但这就说得远了。

```

    continue;
}

```

这里的形参长得有点不同，它是一个引用，我们会在下一章中讲到。

## 4.5 函数递归

我们提及过，不只有主函数可以调用其它函数，任何函数都可以调用其它函数。我们也在先前的例子中看到，在 `max(double, double, double)` 中调用 `max(double, double)` 是完全可行的。那么一个函数能否调用它本身呢？C/C++ 给出的答案是可以。这种“一个函数调用它本身”的情况，在编程领域有一个专门的名字，叫作 **递归（Recursion）**。

你可能会有困惑：我怎么能用一个东西来定义它本身呢？那不就成了逻辑学上的循环定义了？别急，且先听我慢慢道来。

数学上有个函数叫作阶乘（Factorial），对于非负整数  $n$ ，它的定义是  $n! = 1 \times 2 \times 3 \times \dots \times n$ （特殊定义  $0! = 1$ ）。它还有一种递推定义： $0! = 1; n! = n \times (n - 1)!$ 。

如何理解这个递推定义呢？其实就是， $0!$  的值是给定的，而  $n!$  的值要由  $n$  乘以  $(n - 1)!$  来计算。那么  $(n - 1)!$  如何计算呢？要通过  $n - 1$  乘以  $(n - 2)!$  来计算。这样不断推导下去，就能一直推到  $0!$  这个确定的数据，因此只要知道了  $0!$ ，就可以一步一步推出  $n!$  的值来。

而在编程当中，我们可以用函数递归的方法来计算阶乘，下面是一个示例代码：

代码 4.2: Factorial\_with\_Recursion.cpp

```

// 用递归方法实现阶乘函数
#include <iostream>
using namespace std;
// 如果 factorial 的参数大于 12，就会发生运算结果溢出。可以考虑用 unsigned long long
unsigned factorial(unsigned n) { // 阶乘的参数和值都不是负数，所以定义成 unsigned
    if (n == 0) { // 终止条件，不可或缺
        return 1; // 0! 为 1，直接返回
    }
    else { // 如果没有到达终止条件，就调用 factorial(n-1)，离终止条件更进一步
        return n * factorial(n - 1); // 递归，n! 用 n*(n-1)! 表示
    }
}
int main() {
    cout << factorial(12); // 计算并输出 12! 的值
    return 0;
}

```

我来解释一下这段代码，并回答一下读者可能关心的问题。

“我用 `factorial` 来定义 `factorial`，编译器不会报错吗？”

不会的，因为函数的定义就是一种声明。所以编译器在 `factorial` 函数给定参数列表时就已经知道了它的存在（图 4.5）。

“这段代码会怎么运行？”

想像你就是那个程序，在主函数中，当你遇到 `factorial(12)` 时，你会调用这个函数并代入实参  $n=12$  从而求出它的返回值。而在运行时发现，因为  $n==0$  不满足，所以你不得不再去求 `factorial(11)`，然后是 `factorial(10)`，以此类推。如果一切顺利的话，你会在求 `factorial(0)` 的时候直接返回 1，然后用  $1 * factorial(0)$  算出 `factorial(1)` 的返回值；用  $2 * factorial(1)$  算出 `factorial(2)`

```
unsigned factorial(unsigned n) {
    if (n == 0){
        return 1;
    }
    else {
        return n * factorial(n - 1);
    }
}
```

图 4.5: factorial 的定义就是一种声明, 所以编译器能够找到它

的返回值; 用 `3 * factorial(2)` 算出 `factorial(3)` 的返回值……一直到你算出 `factorial(12)` 为止。图 4.6 展示了这一过程。

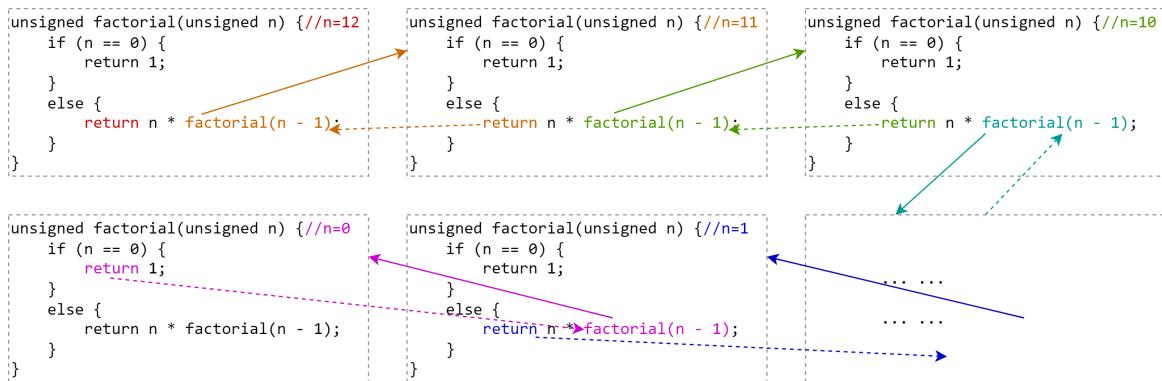


图 4.6: factorial 函数递归的运行过程

“函数递归运行的过程中, 难道还能真的像图 4.6 那样, 运行到一半再跑到开头吗? ”

读者可能容易产生这样的误会, 认为“既然它是同一个函数, 那递归的时候不就是在这一一个函数中打转嘛”。

不然。这个函数在编译时和运行时的概念是不同的。在编译时, 编译器会认为它是一个函数; 但在运行时, 内存中会为程序创建一个调用栈 (Call stack) 来存储函数调用的信息。每次函数调用 (包括一般调用和递归调用) 都会在栈中压入一个堆栈帧 (Stack frame), 这个堆栈帧中存储了本次调用的必要信息 (比如, 本次调用的实参是什么, 返回地址是什么)。在结束调用时, 返回值传回给调用它的函数 (栈的结构保证了调用它的函数也在调用栈中), 然后堆栈帧移除, 如图 4.7。

简言之, 虽然只有一个函数, 但有很多个堆栈帧, 每个堆栈帧存储了各自的调用信息。所以自然不会有冲突了。

“为什么一定要有终止条件呢? 难道不能像 `while(true)` 语句这样永远循环吗? ”

我们可以把递归结构当作是一种循环, 而 `for` 或 `while` 可以更具体地说是一种迭代 (Iteration)。从功能上讲, 递归和迭代是非常相似的, 它们可以互相替代, 而关于“用递归还是迭代”的讨论也是旷日持久。这里我们先不参与此讨论, 只解答原本的问题。

递归与迭代有一点不同在于, 迭代语法不需要依赖调用栈, 比如 `for(int i=0; i<1e9; i++)`, 无论循环多少次, `i` 永远占据一个 `int` 的内存空间; 但递归是需要依赖调用栈的, 并且调用栈的大小有限, 一旦内存不够, 就会发生堆栈溢出 (Stack overflow), 直接导致程序故障罢工。正因如此, 我们可以写无限循环的迭代, 但不能写一个无限循环的递归。

希望读者看过上述的讲解以后, 已经对递归有了基本的认知。

在编程时, 善用递归可以帮助我们节省很多写代码的麻烦。再举个例子, 我们打算写一个简单的

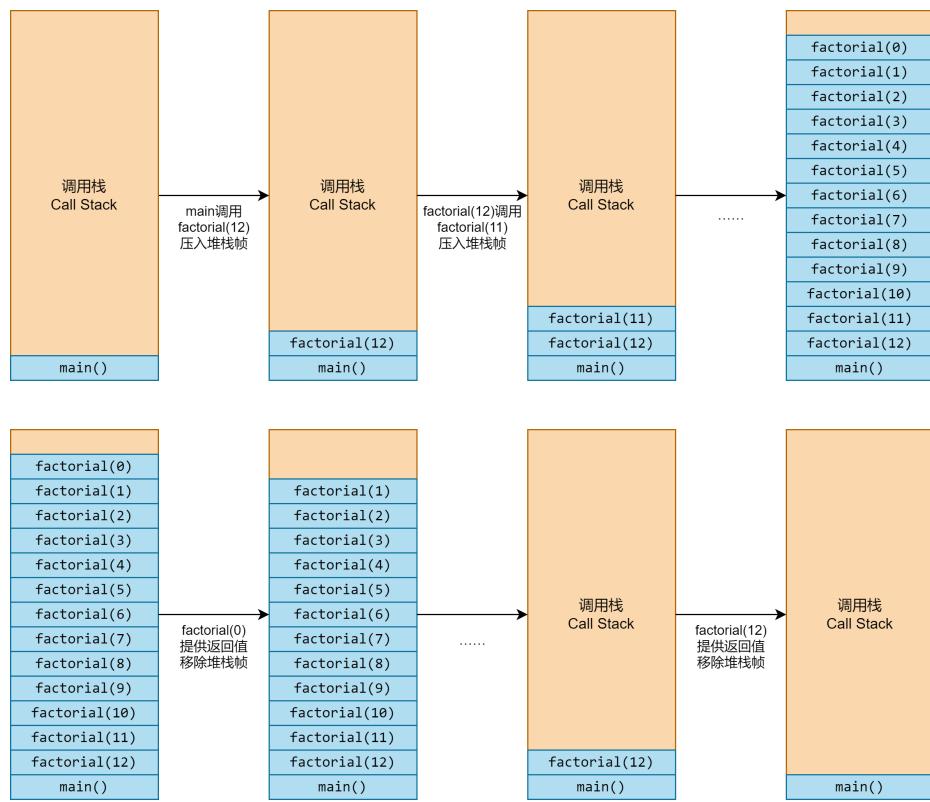


图 4.7: 调用函数时压入堆栈帧, 返回值后移除堆栈帧

求斐波那契数<sup>21</sup>的函数, 通过传入的正整数参数  $n$  来求出斐波那契数列的第  $n$  项。

如果用递归来写这个函数, 就要比迭代轻松不少 (尤其是在我们还没有学习数组的时候)。

```
unsigned fibonacci(unsigned n) {
    if (n <= 2) { // 前两个数的值都是 1, 直接给定
        return 1;
    }
    else { // 第  $n$  个数由第  $n-2$  个数加第  $n-1$  个数得到
        return fibonacci(n - 2) + fibonacci(n - 1);
    }
}
```

## 4.6 函数模板简介

在重载 `max` 时, 我们发现一个问题: 如果我们只提供这两种重载:

```
double max(double, double); // 接收两个 double 类型参数
long long max(long long, long long); // 接收两个 long long 类型参数
```

那么当我们调用 `max(1,2)` 时就会出现二义性问题。这是因为, `int` 隐式类型转换为 `double` 类型或 `long long` 的优先性相同, 编译器就会面临两难处境, 不知该转换成哪种类型是好。解决的方法也很简单, 我们再重载一个 `int max(int, int)` 就好了嘛。

```
int max(int, int); // 接收两个 int 类型参数
```

<sup>21</sup> 斐波那契数列 (Fibonacci sequence), 是一个特殊的数列。它的前两项都是 1, 而后面的每个斐波那契数都由前两项相加得到。前十一个斐波那契数为: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55。

但这不是一劳永逸的方案。如果我们之后又需要做两个 `long double` 求最大值的操作，那么我们还需要再作如下重载：

```
long double max(long double, long double); // 接收两个 long double 类型参数
```

实际操作中我们不一定碰到这么多类型一起出现的情况，但是为了同样的功能写这么多重载，而且每次用别的类型时都要考虑“是否还要加一个重载”，这是很麻烦的。能不能有一种方案，可以针对各种不同的类型，一次性给出同样的解决方案呢？泛型编程（**Generic programming**）的理念就这样应运而生了。

泛型编程理念的核心在于，我们无需考虑具体的数据类型，只要给它们一套通用的模版（**Template**）即可。而编译器会根据代码中的需要，生成对应的函数实例（**Instance**）。

以求两数最大值的函数为例，我们可以定义如下的函数模版（Function template）。

```
template<typename T> // 假设一个模版参数 T, T 可以是任何一种类型
T maximum(T a, T b) { // 返回类型为 T, 接收两个 T 类型的参数
    return a > b ? a : b; // 用条件表达式实现, 代码更简洁
}
```

这里之所以使用 `maximum` 名字是因为 `std` 命名空间中已有 `max` 模版，为了防止重名引起一系列麻烦，就在此处换用它名。<sup>22</sup>

这里我们用 `template` 关键字定义一个模版，这个模版有一个模版参数，名为 `T`，它就是“某个类型”。

然后是函数定义（声明）部分，我们指定了这个函数模版的返回类型为 `T`，名称为 `maximum`，接收两个 `T` 类型参数。

再看函数体，这里涉及到了一个条件表达式，用到了条件运算符“`@? :@`”。条件运算符的功能很容易理解，它接收三个操作数，如果第一个操作数的值是 `true`<sup>23</sup>，那么条件运算符的返回值就是第二操作数；如果第一个操作数的值是 `false`，那么条件运算符的返回值就是第三操作数。在本函数模版中就可以这样理解：如果 `a>b` 得到 `true`，就返回 `a`；否则返回 `b`——这和使用 `if-else` 结构的作用是相似的。

接下来我们就可以在主函数中使用 `maximum` 函数模版了。非常简单：

```
int main() {
    cout << maximum(1, 2) << endl; // 调用 int maximum(int,int)
    cout << maximum(1ll, 0ll) << endl;
    // 调用 long long maximum(long long,long long)
    cout << maximum(1., 2.) << endl; // 调用 double maximum(double,double)
    cout << maximum(1.l, .1l) << endl;
    // 调用 long double maximum(long double,long double)
}
```

编译器在知道了这个函数模版之后，每当遇到需要的函数（比如，调用 `maximum(1,2)`，需要一个 `maximum(int,int)`），就会根据这个模版，制造一个 `maximum(int,int)` 函数来匹配它。

那么在这个主函数中我们用四种类型的参数调用 `maximum` 函数，编译器就会根据需要，制造四个函数：

```
int max(int a, int b) { // int 版本
    return a > b ? a : b;
}
```

<sup>22</sup> 我们会在第七章中讲解如何规避这类问题。

<sup>23</sup> 如果第一操作数不是 `bool` 类型，会进行布尔转换变为 `bool` 类型，下同。

```

long long max(long long a, long long b) { //long long版本
    return a > b ? a : b;
}

double max(double a, double b) { //double版本
    return a > b ? a : b;
}

long double max(long double a, long double b) { //long double版本
    return a > b ? a : b;
}

```

所以说，使用函数模版相当于我们给编译器提供了一个范本，而编译器会根据这个范本来制造对应类型的重载。这样不用我们手动写一大堆函数重载了。

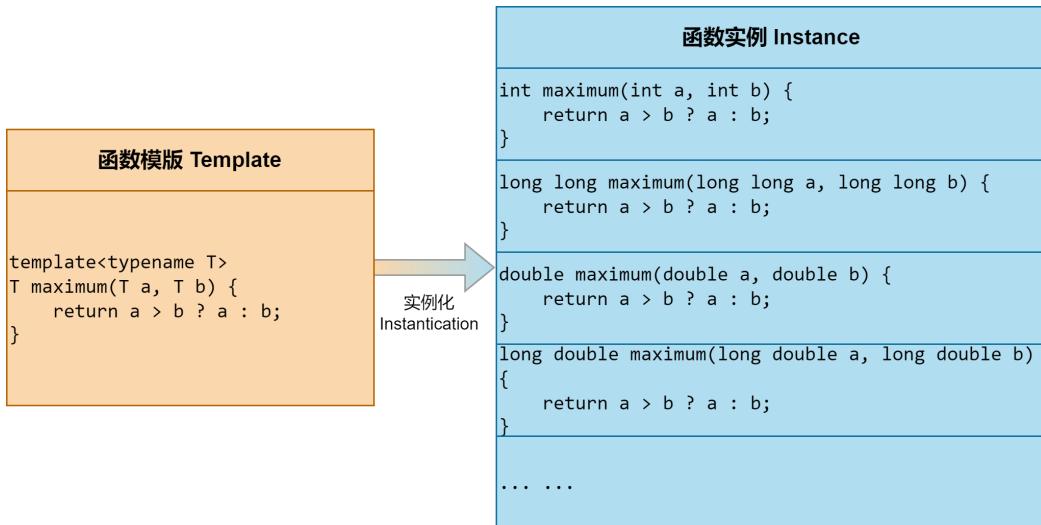


图 4.8: 从函数模版到函数实例

我们可以再加一个三数求最值的函数模版。我们可以把它称为函数模版的重载。

```

template<typename T> //假设一个模版参数T, T可以是任何一种类型
T maximum(T a, T b) { //返回类型为T, 接收两个T类型的参数
    return a > b ? a : b; //用条件表达式实现, 代码更简洁
}

template<typename T> //注意, 每次声明(定义)模版都需要写template, 不能共用
T maximum(T a, T b, T c) { //返回类型为T, 接收三个T类型的参数
    return a > b ? (a > c ? a : c) : (b > c ? b : c); //请读者自行分析
}

```

本节只是对模版函数和泛型编程作一个简单介绍，我们会在第十一章更详细地探讨这方面的问题。

# 第五章 复合类型及其使用

在前面的章节中，我们讲了基本数据类型。不同类型数据的使用可以帮助我们解决一些小规模的问题；再搭配函数的使用，可以让我们的代码更成体系，更有逻辑。

复合类型（Compound type）的出现让函数的功能大放异彩。借助复合类型，我们可以实现更加复杂的功能，比如数据的批量处理（数组），函数的实参修改（指针和引用），以及动态内存分配。很多技术含量较高的代码必须借助复合类型才能实现，所以本章所介绍的内容有着承前启后的作用。

本章主要介绍指针、引用和数组相关的知识；至于结构体、联合体和类，我们留到下一章再讲解。

在本章，我们也将使用 `typeid` 和 `std::is_same` 等运算符或函数，来帮助我们梳理复合类型与基本类型之间的关系，它们是我们处理类型问题时相当有用的工具。

## 5.1 指针

“我认为赋值语句和指针变量可以说是计算机科学最具价值的宝藏。”<sup>1</sup>

——高德纳<sup>2</sup>

### 内存与地址

在一个 C/C++ 程序运行时，内存中会存储着有关这个程序的信息。我们可以把这些内存空间分为五个区段（如图 5.1 所示）：

- 代码段（Code/Text Section）。这个区域存储着一系列指令，可以告诉电脑这个程序要做什么。比如说，函数的定义就在在代码区中，这部分内容告诉电脑它要做什么。
- 数据段（Data Section）。这个数据区不意味着“存储所有的数据”。它只存储一些全局变量或已初始化的静态变量，也就是在函数之外的域中定义的数据<sup>3</sup>
- BSS 段（BSS Sectino）。通常用于存放未初始化的静态变量。
- 栈段（Stack Section）。局部变量（比如主函数中定义的变量）和每次函数调用时的实参等信息，都存储在栈区（想想我们上一章中讲到的“调用栈”）。每次我们定义一个新数据，这个数据的信息就会被压入栈中，当这个数据的作用域结束时，它就会出栈。
- 堆段（Heap Section）。用于动态内存分配。我们会在本章中介绍动态内存分配。

内存中有无数字节，不同的数据需要不同的字节数量来存储，我们可以用 `sizeof` 来计算字节数量的值。

<sup>1</sup>原文：I do consider assignment statements and pointer variables to be among computer science's "most valuable treasures."

<sup>2</sup>高德纳（Donald Ervin Knuth），美国计算机科学家和数学家，1974 年图灵奖得主。高德纳成就颇丰，其中最负盛名的是它的著作《计算机程序设计艺术》（*The Art of Computer Programming*）。

<sup>3</sup>我们会在第七章讲解有关知识。

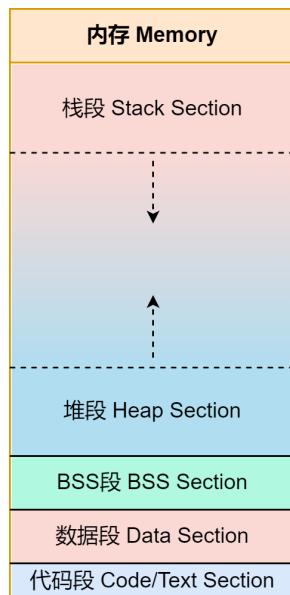


图 5.1: 程序的内存布局

既然内存中有这么多字节，那么程序要怎么判断自己要用的数据在哪里呢？这就需要用到内存地址（Memory address）了。

内存中的每个字节都有一个内存地址。就像我们根据门牌楼号来找到一间房子那样，程序也需要通过内存地址才能快速找到一个字节，并且读取或者修改它的值。

举例来说，我们在主函数中定义一个 `char` 型变量，命名为 `ch`。在运行时，每当遇到了这个指令，程序就会在栈段当中找到一个闲置的字节，用来存储这个变量的值。那假如我们定义一个 `double` 型的变量 `d` 呢？程序会在栈区中找到八个连续的闲置字节（在绝大部分系统中，`double` 型的内存空间占用都是 8 字节），用来存储这个变量的值。

在 C++ 中，`&` 运算符可以对某个变量求地址。一般说来，如果要输出地址的话，地址值都是用十六进制的形式<sup>4</sup>来表示的。`&` 运算符单独作用于一个变量时，返回值是它的地址<sup>5</sup>。

```
int main() {
    double d {3.1415926}; // 定义一个 double 型变量 d
    cout << &d; // 用&取 d 的地址值并输出
}
```

这个程序的输出结果是：

---

0x7ffccff275b8

---

不同计算机的实际情况可能大相径庭，不同编译器处理内存分配的方式也可以大同小异，所以这个地址的具体值因机器而异，我们需要在意的只是格式。在这里，它输出了一个十六进制的结果，这正是 `d` 的地址值。

细心的读者可能会有疑问：`d` 作为一个 `double` 型数据，明明占据了八个字节，为什么输出结果只给了我们一个地址呢？

这是因为，对于任何一种类型，在取地址时只会返回它的第一个字节的地址。因为任何一个变量占据的内存都是连续的一段，所以我们只要知道了第一个字节在哪里，当然就知道整个变量占据了

<sup>4</sup>即，以 `0x` 开头（有些编译环境，如 MSVC，输出的地址值是不带 `0x` 开头的，所以别惊讶），后面表示为 `0~F`（或用小写）的一串数字，如 `0xe23df5cc`。

<sup>5</sup>并不是所有的数据都可以取地址！我们会在精讲篇中说明，它关系到左值和右值的问题。

哪个区间的内存了。



图 5.2: d 在内存空间中的示意图

如图 5.2 所示，既然 `0x7ffccfff275b8` 是 `d` 的地址，而 `sizeof(double)` 又是 8，那么自然而然，`d` 的信息应当存储在从 `0x7ffccfff275b8` 开始的 8 个字节，一直到 `0x7ffccfff275bf` 为止。

所以无论哪个类型，其内存地址指的都是它所占内存空间中的第一个字节的地址<sup>6</sup>。请牢记这点，它会让你后面的学习过程更轻松的。

补充一条：读者可能会在自己尝试的时候发现一个有趣的现象：如果试图输出一个 `char` 类型的地址，输出的结果可能会是各种奇奇怪怪的内容，但绝对不像是个指针。我们会在字符串那一节再介绍这个问题。如果你坚定地想输出 `char` 变量的地址，你可以采用这种写法：

```
char ch {48}; // 定义char型变量并赋初值'0'
cout << (void*)&a; // 指针类型转换为void*, 这样就会以地址值的形式输出
// 也可以把(void*)&a换作reinterpret_cast<void*>(&a)
```

## 指针的定义和使用

仅仅知道地址值还不够，我们很难把它用起来。我们的当务之急是，找到一种可以“存储地址值”的数据类型。可是很遗憾，虽然地址值的表示方式像一个整数，但是我们不能把它的值存到整型数据当中。实际上，C/C++ 专门有一大类存储地址值的数据类型，叫作指针变量，简称指针（Pointer）。

请留意，指针不是一个数据类型，而是“一群”数据类型。“指向 `int` 的指针”和“指向 `double` 的指针”是不同的数据类型。

定义一个指针的基本语法如下，这里有两种主流写法可供选择，它们是等价的：

```
<类型>* <指针名>; /*写在<类型>一侧，与<指针名>分开
<类型> *<指针名>; /*写在<指针名>一侧，与<类型>分开
```

<sup>6</sup>除非你重载了取地址函数 `&`，这种情况我们不讨论。

第一种写法更清晰，它说明了这个名字是一个指针类型。

```
int a {0}; // 定义一个 int 型变量 a，并初始化为 0
int* pa {&a}; // 定义一个 int 指针 pa，并初始化为 a 的地址
```

但是它也会引发一些困惑……

```
int* p1, p2; // p1 是 int* 类型，但 p2 是 int 类型！没想到吧
```

第二种写法则不会出现这种困惑，因为本身就是要在指针名一侧写 \* 的。

```
int *p_1, p_2, *p_3; // p_1 和 p_3 是 int* 类型，而 p_2 是 int 类型
```

这种写法还能降低其它有关问题的理解成本，我们很快就会遇到。

那么究竟使用哪种写法呢？不同的人当然有不同的理解和习惯，不必强求。不过在编写本书的过程中，我选择我最喜欢的写法：只在描述返回类型的时候用第一种写法，否则用第二种写法。

习惯上，我们喜欢把“p 存储了 a 的地址”称为“p 指向了 a”。这当然是一种很形象的表述，但很多教材会为了这点“形象”而忽略了对本质的阐述，这是得不偿失的。

现在让我们假想这样一个过程：我先定义了一个 **double** 型的变量 d，然后又定义了一个 **double\*** 型的变量 pd（这是一个指针变量）。

```
double d {3.14159}; // 定义一个 double 型变量 d
double *pd {&d}; // 定义一个 double* 型指针变量 pd，它指向 d
```

理论上讲，一个程序只要知道了某个变量的地址，就具备了读取或修改对应内存空间的能力。那么 d 这个名字，也就无所谓有没有了。我们能否只根据 pd 中保存的地址值，就实现读取和修改 d 的功能呢？当然可以！

这里我们需要一个运算符：取内容运算符 \*<sup>7</sup>。读者可能会困惑——这不是乘法运算符吗？其实你可以把它当作一种重载！当它接收两个操作数时，它的含义就是乘法运算；当它接收一个操作数时，它的含义就是取指针的内容（见图 5.3）。



图 5.3：我们可以把 \* 接收不同数量操作数的情况当成一种重载

说回原来的话题。通过 \* 运算符，我们可以无需变量名就可以读取一个指针的内容；而且在允许的情况下，我们还可以修改其内容，这个修改是直接对内存中的内容进行修改，和修改 d 的值一样直截了当。

```
double d {3.14159}; // 定义一个 double 型变量 d
double *pd {&d}; // 定义一个 double* 型指针变量 pd，它指向 d
cout << *pd << endl; // 输出 pd 取内容的结果，当然是 3.14159
*pd = 2.71828; // 修改 pd 对应内存中的内容，等效于直接修改 d 的值
cout << d; // 输出 d 的值，会发现结果是 2.71828
```

读者可以自行运行这段程序，或者写一些别的代码试试，会发现 d 和 \*pd 的特征非常相似，它们几乎就是同一个事物！

从本质上讲，变量的名字只是一个“表象”而已，藏在背后的最关键的东西还是内存中对应的若干字节中存储的信息。C/C++ 允许我们自定义变量名，这样的好处在于，我们可以不必像汇编语言

<sup>7</sup>也有些资料将其称为解引用运算符。

那样，用一些枯燥的 `rip`, `rsp`, `eax` 之类的名字来描述真实世界中丰富多彩的信息，而是用一些顾名思义的名词，让我们和别人看了都能懂。

但计算机还是要把这些名字还原成内存中的一个个信息，然后再处理。变量名对于计算机来说，只是一个绑定了相应内存地址的名字罢了。于是 `d` 和 `*pd` 没什么分别；`&d` 和 `pd` 也没什么分别。图 5.4 阐述了它们的关系。

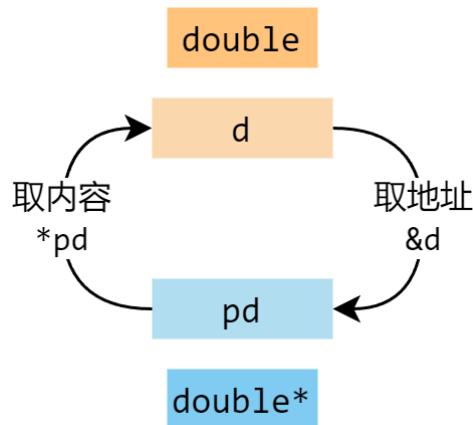


图 5.4: 变量 `d`、地址 `&d`、指针 `pd` 与内容 `*pd` 的相互关系

这个关系是循环的，我们可以先取地址再取内容，最后得到的东西和 `d` 是一回事。所以我们还可以对这个东西再取地址，再取内容，再取地址，再取内容……以至无穷。

```
cout << *&*&*&*&d; //*&*&*&*&d 相当于 *&*&*&d, 相当于 *&d, 相当于 d
```

不过这种写法就太没意义了，绕了一大圈，最后还是回到 `d`。实际编程中我们不会这么做。

但是不要止步于此。指针的存在还为我们提供了一种新的可能性：我们可以绕过“变量名”这一步，直接用指针来访问和修改内存中的信息。这是我们后续要讲的指针参数传递、数组和动态内存分配的基础。

## 指针参数传递

在写函数的时候，我们可能会试图通过函数来改变实参的值。举个例子，我们想写一个函数 `exchange`，来交换它的两个实参的值。为了方便阐述，我在这里就不用函数模版了，我们假定它接收两个 `int` 参数。

```
void exchange(int a, int b) { //这里只需实现副作用，无需求值，故设返回类型 void
    int tmp {a}; //定义一个临时变量 tmp，用以存储 a 的值
    a = b; //赋值，现在 a 获得了 b 的值，我们的目标完成一半
    b = tmp; //赋值，现在 b 获得了 tmp 的值，也就是 a 原来的值
    return; //对于返回类型为 void 的函数来说，return 是可有可无的
}
int main() {
    int a {3}, b {4}; //定义 a=3, b=4
    exchange(a, b); //交换 a 和 b 的值，预期是 a 变成 4, b 变成 3
    cout << a << ' ' << b; //输出 a 和 b 的值，检验一下结果
    return 0;
}
```

这个程序的运行结果是

---

3 4

---

这个代码的逻辑看上去很正确，但是为什么它完全不起作用呢？

回想一下我们之前讲过的内容。每次程序调用一个函数的时候，都会在栈区压入一层堆栈帧。这个堆栈帧中就含有在本次调用过程中创建的变量，如图 5.5 所示，这时调用栈中发生修改的是 `exchange` 对应内存空间中的 `a` 和 `b`，而不是 `main` 函数中的 `a` 和 `b`。这两对变量虽然同名，但它们对应的是内存空间中完全不同的区域。

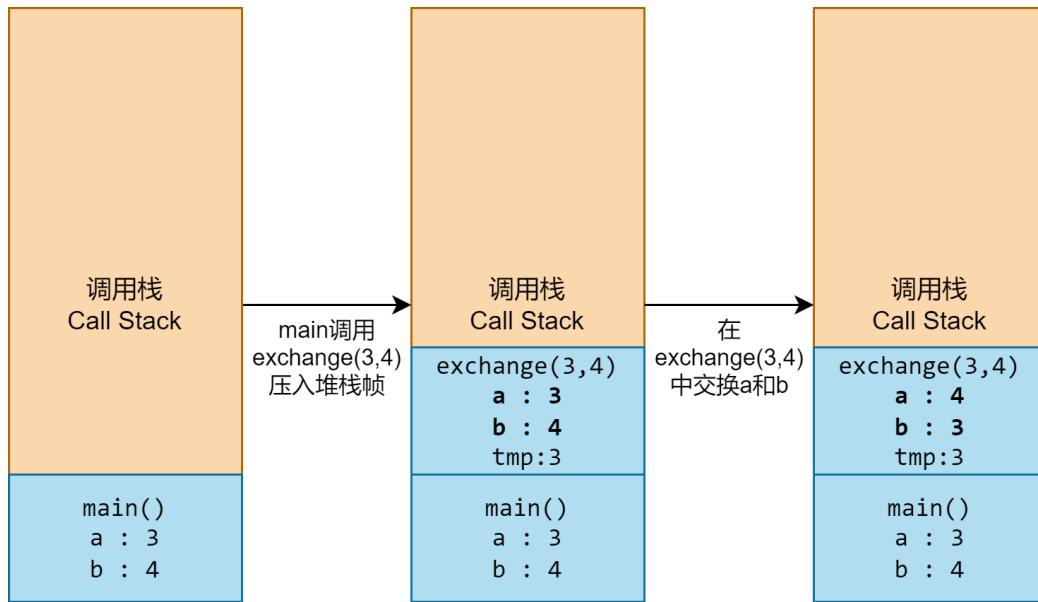


图 5.5: 在调用 `exchange` 函数时，调用栈中发生的变化

所以在函数参数传递的过程中，程序并没有把“整个变量”传递给目标函数，而只传递了它的“值”。至于被调用的那个函数，它会另外定义一个新的变量（它在内存空间中另一自己的一席之地）来接收传入的值。很多资料会把它称为副本（Copy）。

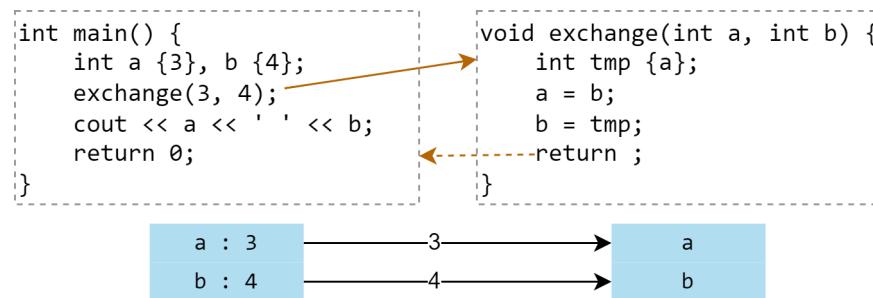


图 5.6: `main` 向 `exchange` 中传入的信息只有值

我们会发现，不同函数之间只能共享值，这种现象好像一种屏障——在某些时候，它显得十分安全，我们不会在一个函数中乱改其它函数中的信息；但在另一些时候，它显得非常不灵活，如果我们用别的函数只能读取而不能修改此函数中的值，函数的功能就会受到很大限制——比如我们试图写一个值交换函数而不能。

所以我们需要找到一种方法，使我们可以跨过函数作用域的屏障，来修改特定的变量值。我们的

答案是使用指针传递（Passing by pointer）。

回顾一下前文中讲过的三个基本要点：

1. 地址是一个值。我们可以把它当作和整数值、浮点值、字符值和布尔值等并列的事物来看待。
2. 只要知道了某个变量的地址，我们就有可能修改内存中相应区域的内容。我们可以通过取内容运算符来读取或修改其中的内容，效果等同于使用相应的变量名。
3. 指针是一大类数据类型，它们存储的信息是地址值。如果一个指针存储了某个变量的值，我们可以说“这个指针指向那个变量”。

搞清了以上三点之后，我们就不难想到另一种可能性——如果我们把 a 和 b 的地址作为参数传给 exchange 呢？

地址是一个值，因此它可以作为实参来传递，没有问题。

只要 exchange 函数知道了 a 和 b 的地址，那么它自然具备了修改 a 和 b 的内容的能力。也没问题。

那么 exchange 函数要用什么形参来接收地址值呢？当然是用指针，因为只有指针才能存储地址值。

于是一切顺理成章，我们的解决方案就给出来了。现在我们就来试着来改一下 exchange 函数：

```
void exchange(int *pa, int *pb) { // 它接收两个int*参数
    int tmp = *pa; // 定义一个临时变量tmp，用于存储*pa的值
    *pa = *pb; // 赋值，现在*pa获得了*pb的值
    *pb = tmp; // 赋值，现在*pb获得了tmp的值，也就是*pa原来的值
}
int main() {
    int a {3}, b {4}; // 定义a=3,b=4
    exchange(&a, &b); // 向exchange函数传递&a和&b的值，也就是地址
    cout << a << ' ' << b; // 输出a和b的值，检验一下结果
    return 0;
}
```

最后的运行结果果然不出我们所料，正是

---

4 3

---

所以这种函数参数指针传递的方式要比我们之前见过的值传递（Passing by value）更有效。但是话又说回来，并不是任何时候用指针传递都比用值传递要更好的。初学者可能很难搞清楚“什么时候要用什么”，其实没关系。你只需要多写一些代码，等到经验丰富之后自然就清楚了。

还请读者留意：虽然我们人为地划分出了“值传递”和“指针传递”的概念，但它们的本质都是在传递值！只不过后者传递的是“指针值”罢了。

## 野指针问题

说完了指针在函数传递中的应用，读者应当对指针的强大有了初步认识。指针为我们编写程序带来了极大的灵活性，但其中也潜藏着一些风险。所以现在就让我们返回到一些细节的讨论上来。

我曾说过，如果定义一个局部变量而不初始化的话，它将会存储一个不确定的值。使用这种不确定的值可能会导致程序运行的结果不可预测，所以我们需要在使用它之前通过初始化、赋值或输入等等各种方法让它成为一个可控的变量。

指针亦如此。如果我们不进行初始化或赋值，让它指向一个安全的位置，那么它可能会指向内存中的任何一个字节，我们把这种指向不确定的指针叫作野指针（Wild pointer）。野指针是很可怕的，一旦我们试图读取或修改对应未知空间中的信息，我们可能会把这个程序乃至其它进程改爆，从而导致严重的后果<sup>8</sup>。因此，在使用指针之前，我们也要进行初始化或者赋值<sup>9</sup>。

我们可以用同类型变量<sup>10</sup>的地址来为指针赋值，我们已经在前面的例子中接触过这种语法了。不同类型数据的地址不能互相赋值，比如我们不能用 `float` 型变量的地址来为 `double*` 型指针赋值。

```
float f; // 定义 float 变量 f
double *pd; // 定义 double* 指针 pd
pd = &f; // 试图将 f 的地址赋值给 pd
//error: cannot convert 'float*' to 'double*' in assignment
```

这个报错信息的意思是：在赋值时，`float*` 类型（即 `&f`）不能隐式类型转换为 `double*` 类型（即 `pd`）。这说明指针类型之间不能随便进行隐式类型转换。

如果我们定义了一个指针，但暂时还不想让它指向哪个变量，而又害怕空置会产生野指针，那么我们还有一个选择：`nullptr`。

```
long long *pll = {nullptr}; // 定义 long long* 型指针并指向空地址 nullptr
```

空地址（`nullptr`）不是一个运算符，它是一个常量地址。`nullptr` 的类型比较特殊，它是 `nullptr_t` 类型的，我们知道这个也没什么用。但这个类型的特点在于，它可以隐式类型转换为任何一个指针<sup>11</sup>类型。因此，`nullptr` 可以为各种指针进行初始化。

`nullptr` 是受保护的。我们对它的内容进行的读取和修改都没有任何意义，也不会造成任何风险。所以当我们不需要这个指针指向什么时，最好让它指向空地址 `nullptr`。我们在动态内存分配章节中就会讲到它的一种应用，那就是在动态内存回收之后，为防野指针出现，就让这个指针指向空地址。

## 指针的运算

接下来讲解一下指针的运算。虽然指针的值很像整型数据，但是它的加减法规则与整型数据截然不同。

两个指针的加法是没有意义的，因为我们根本不能确定加完了之后得到的地址是指向什么的（这不是野指针吗）。

而指针与整型的加/减法是有意义的，尤其是在数组当中。一个指针与整数相加/减，它的返回值是和原来指针相同的类型（比如说，`int*` 与 `short` 相加/减，返回值为 `int*`）。

```
float f, *pf {&f}; // 这里不需要初始化 f，因为只需用到其地址，而不需要其值
cout << pf << endl << pf - 1 << endl; // 输出 pf 和 pf-1 的值，它们都是地址值
long double ld; // 同理，只是改成了 long double 型
cout << &ld << endl << &ld + 1; // 直接输出 &ld 和 &ld-1，效果相同
```

运行结果是这样的<sup>12</sup>：

---

```
0x7ffc4bf3837c
0x7ffc4bf38378
```

<sup>8</sup>对于个人电脑来说，这种后果可能很轻微——大不了我们重启电脑，这时内存中的一切又焕然一新了。但对于服务器等需要长期运转且不能轻易停机的设备来说，损失可能非常大。

<sup>9</sup>我们不能通过“输入地址”的方式来改变指针的值，这同样是很危险的。

<sup>10</sup>实际上，指针也是一种变量。但是人们习惯上把“指针”和“变量”视为互不重合的两个概念。本书也沿用这种习惯，通常把指针和变量这两个概念分开。如果需要把指针当作变量的一部分，会特殊说明。

<sup>11</sup>乃至高阶指针，见后面的章节。

<sup>12</sup>再次提醒读者，实际输出的内存地址因机而异，我们应当关注的是几个输出地址之间的相互关系。

```
0x7ffc4bf38380
0x7ffc4bf38390
```

输出内容是很长的十六进制形式，我们来分析下。

首先输出的两个是 `pf` 和 `pf-1` 的值。从数值上看，前者是 ...37c，而后者是 ...378，后者虽然是 `pf-1`，但地址却减小了 4 个字节。

接下来输出的两个是 `&ld` 和 `&ld+1` 的值。从数值上看，前者是 ...380，而后者是 ...390，后者虽然是 `&ld+1`，但地址却增加了 16 个字节。

之所以会出现这种情况，关键就在于不同类型数据在内存中占用的字节数目不同。一般说来，`float` 型占据 4 字节内存空间，而 `long double` 型占据 16 字节内存空间。

指针和整型的加减法并不是单纯地移动多少个字节，而是移动了多少个数据（可以理解为，偏移量乘以单个数据占用的字节数），如图 5.7 所示。这样设计是有它的道理的，等我们讲到数组，读者就会很容易理解了。

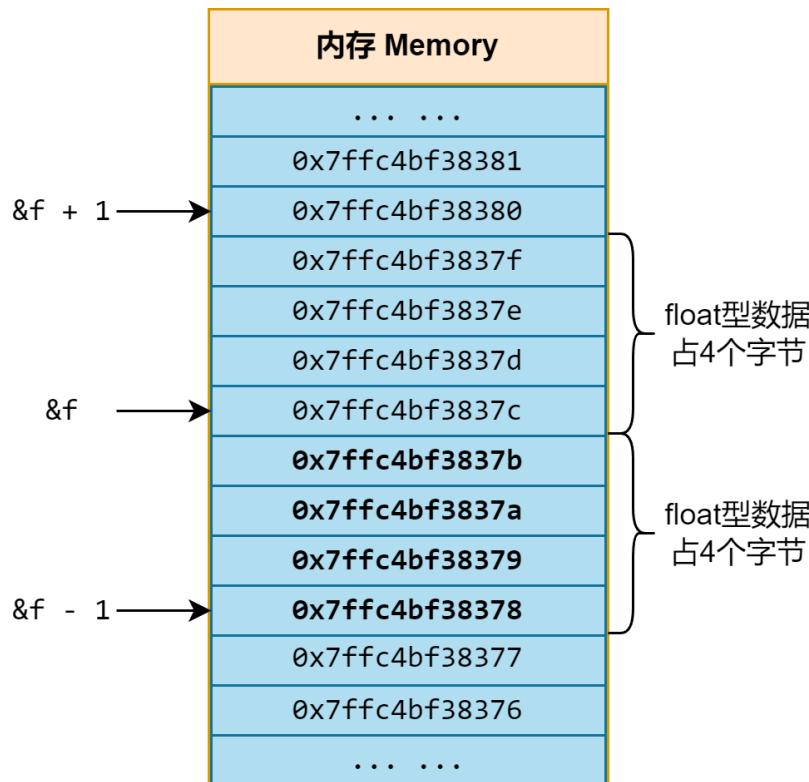


图 5.7: 指针加法运算的偏移量是以数据为单位的，而不是地址值

两个同类型指针可以相减，其返回值是一个特殊的类型，叫作 `ptrdiff_t`。不过我们也无需纠结这种细节，因为 `ptrdiff_t` 可以隐式类型转换为整型，所以我们可以直接把它当作整型来用，比如输出。

```
int a, b, c;
cout << &a << endl << &b << endl << &c << endl;
cout << &a - &b << ' ' << &c - &a << endl;
```

运行结果是这样的：

```
0xffff604ad654
0xffff604ad658
```

```
0x7fff604ad65c
```

```
-1 2
```

我们可以看到，这里的 `a`, `b`, `c` 地址值依次递增 4，而 `&a-&b` 得到 `-1` 而非 `-4`，这说明指针的减法也是在计算数据偏移量，而不是字节数偏移量。

## 5.2 “常量指针”与“指向常量的指针”

我们在前面讲基本数据类型时，曾经用 `const` 限定符来标记一个变量，使之成为常量；或用 `constexpr` 限定符来标记一个变量，使之成为常量表达式。

既然指针也是“变量”的一种，那么我们能否用 `const` 或 `constexpr` 来把它们变成常量或常量表达式呢？

先说结论：我们可以使用 `const`，将指针限定为“常量指针”或者“指向常量的指针”乃至“指向常量的常量指针”。但是我们不能定义成 `constexpr`，因为编译器是不可能未卜先知，知道某个变量在运行时的地址。

那么什么是常量指针，什么是指向常量的指针呢？本节就来介绍这两个概念。

当我们拿到一个没有任何限制的指针时，我们可以对它的指向进行修改——也就是修改这个指针所存储的地址值；我们还可以通过取内容运算符 `*` 来对它的内容进行修改——也就相当于修改对应变量的值，而修改之后这个指针依然指向原来的位置。如图 5.8 所示。

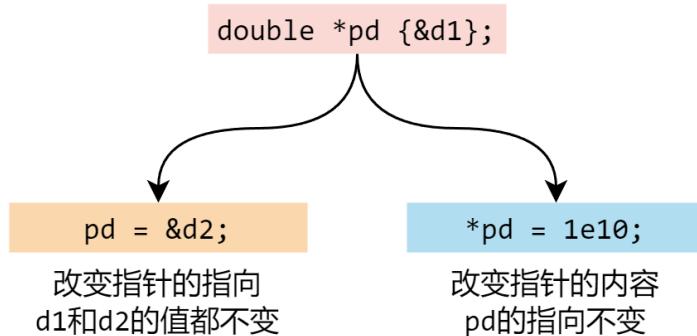


图 5.8: 修改指针的指向/修改指针的内容

我们可以通过 `const` 来限制指针的一部分修改能力，从而将其变为常量指针或指向常量的指针。为了避免在代码上出现误解和混淆，我先介绍概念。

- **常量指针 (Constant pointer)** 限制了指针改变指向的能力。这种指针一经初始化，就不能改变指向。但是这不影响我们可以修改它的内容。
- **指向常量的指针 (Pointer to constant, 简称指针常量<sup>13</sup>)** 限制了指针改变内容的能力。我们不能用这种指针来改变内容，但是它可以改变指向。虽然名为“指向常量的指针”，但它也完全可以指向变量。如果它指向变量，那么我们可以用变量名来修改内容；但是不能用这个指针来修改内容。

我们还可以用更符号化的表述来阐释它们之间的关系：

如果 `pd` 是一个常量指针，那么 `pd` 不能改变；但 `*pd` 是有可能改变的。

如果 `pd` 是一个指向常量的指针，那么 `*pd` 不能改变；但 `pd` 是有可能改变的。

<sup>13</sup>笔者十分不推荐使用“指针常量”这个名字！它极易与常量指针混淆。

当你理顺了它们的区别之后，我们就来讲讲怎么用 **const** 限定符来把指针限制成常量指针或指向常量的指针。这是定义的语法：

```
const int *ptc1; // 定义一个指向常量的指针。它可以改变指向，所以无需初始化
int const *ptc2 {nullptr}; // int与const的位置可互换，也是定义指向常量的指针
int* const cp {nullptr}; // 将const置于*之后，定义一个常量指针，必须初始化
```

怎么理解这个定义语法，并把这三种看上去非常相像的语法区分开呢？我们可以这样想：

**const** 会对它所标记之物作出限制，使其不能改变。如果 **const** 限制的是 **p**，那么 **p** 就不可改变，但 **\*p** 可以改变——这正是常量指针的特性；如果 **const** 限制的是 **\*p**，那么 **\*p** 就不可改变，但 **p** 可以改变——这正是指向常量指针的特性。

回到我们的定义语法。**const int** \*ptc1 这里，我们看，**const** 之后的部分除了 **int**（这个不用管）就是 \*ptc1，于是 **const** 直接限定了 \*ptc1，所以 \*ptc1 不可变，而 ptc1 可变，所以它是指向常量的指针；

**int const** \*ptc2 同理，**const** 直接限定了 \*ptc2，所以 \*ptc2 不可变，而 ptc2 可变，所以它也是指向常量的指针；

到了 **int\* const** cp 这儿，情况有点不太一样。**const** 直接限定了 cp，所以 cp 不可变，而 cp\* 仍然可变，所以它是常量指针。

图 5.9 可以帮助你梳理它们之间的区别。

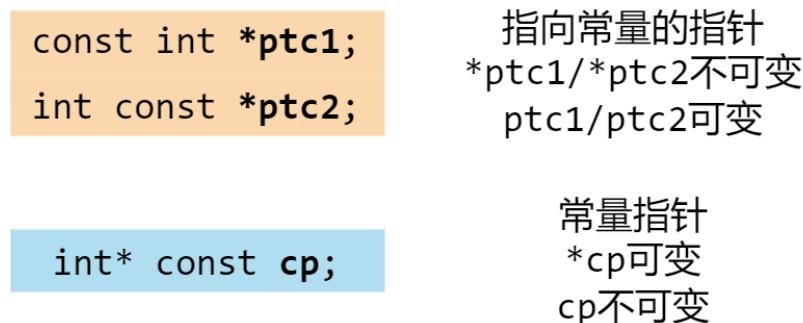


图 5.9: **const** 限制了什么？

知道了这个语法规则，那么定义一个“指向常量的常量指针”的语法也就很容易理解了：

```
const int* const cptc {nullptr}; // 定义一个指向常量的常量指针
```

在这里，第一个 **const** 限制了 \*cptc 不能改变，而第二个 **const** 限制了 cptc 不能改变，所以这个指针一经定义，既不能改变指向，也不能改变内容。

指向常量的指针常见于函数参数，因为很多时候我们需要限制函数对参数的修改能力，确保有些信息是只读的，以免在函数中不小心修改与参数有关的信息。举例来说，cstring 库中有 **strlen** 函数，它可以求出一个字符串<sup>14</sup>的长度。这个函数的声明格式是

```
std::size_t strlen( const char* str );
```

这很好理解。既然它只是求算字符串的长度，那么当然没有必要让它具备修改目标字符串的能力，因此直接定义成指向常量的指针就没有潜在风险了。

<sup>14</sup> 我们会在后面讲到字符串。

### 5.3 (左值) 引用与引用参数传递

我们在前文中介绍了指针参数传递。这种方法固然强大，但是每次传递参数都要使用指针或地址，这就显得有点麻烦了。如果读者了解 C 语言的话，应该就会知道，每次我们使用 `scanf` 来进行输入，都要用取地址符来传入相应变量的地址方可。

```
int num;
scanf("%d", &num); // 传递 num 的地址，这样才能改变 num
```

而在 C++ 中，每次我们使用 `cin` 来输入变量的值时，都不需要取其地址。但是按照我们之前的讲解，如果按值传递的话，我们只能修改副本中的数据，而不会影响原数据。这是怎么回事呢？

```
int num;
cin >> num; // 为什么不用传递地址，也可以改变 num 的值？
```

要解决这个问题，我们就需要了解引用<sup>15</sup> (Reference) 及其在参数传递过程中的作用。

#### 什么是引用？

我们曾言，变量名不是变量的本质。如果两个不同的变量名绑定了同一个地址，那么它们表达的信息就是相同的<sup>16</sup>。我们可以用其中一个名字来读取或修改它们的值，当然也可以用另一个名字。

引用的作用相当于变量的别名，定义一个引用的基本语法是

```
<类型> &<引用名> = {变量}; // 必须在定义之时初始化
```

这里的 `&` 不是取地址运算符的意思，而是表示我们在定义引用。一旦这个引用绑定了这个变量，它就充当了这个变量的别名，我们使用引用的效果与使用变量名等同。

```
int a {3};
int &ref {a}; // 定义一个引用，它充当 a 的别名
++ref; // 更改 ref 相当于更改 a
cout << a; // 输出 a 的值，观察结果
```

这个程序的输出结果将会是 4。这说明我们用 `ref` 和用 `a` 的效果是相同的。

如果你去输出一下它们的地址就会发现，`ref` 与 `a` 的地址是相同的。

```
cout << &ref << endl << &a; // 输出 ref 和 a 的地址，它们应当相同
```

我们还可以定义常量引用，它同样是一种引用，但 `const` 限制了我们通过这个引用来修改变量值的能力。

```
int a {3};
const int &ref {a}; // 定义一个常量引用
++a; // 没问题，a 是一个变量
++ref; // 错误！
//error: increment of read-only reference 'ref'
```

在这种情况下，我们可以用 `a` 来修改相应的值，但不能用 `ref` 来修改相应的值。这说明，“变量”和“常量”并不是直接体现在数据中的信息，而只是“变量名”对数据内容是否拥有修改权限的体现。

至于本来就定义成常量的数据，如果我们用普通引用来作为它的别名，编译器就不会允许。

```
const double std_gravity {9.80665}; // 这是一个常量
double &refgravity {std_gravity}; // 错误！
```

<sup>15</sup>泛讲篇中不讲解右值引用，这里提及的所有引用都指左值引用。

<sup>16</sup>当然，这两个变量名必须是同一类型的，因为类型会影响内存信息的解释方式。

```
//error: binding reference of type 'double&'  
//to 'const double' discards qualifiers  
const double &ref_gravity {std_gravity}; //正确
```

总而言之，我们只能用常量引用来绑定常量。而我们可以使用普通引用或常量引用来绑定变量。如果使用常量引用来绑定变量，我们可以用变量名来修改变量的值，但不能使用引用。

## 引用参数传递

还记得我们在讲指针时介绍的 `exchange` 函数吗？如果要使用引用来接收实参，那么我们就相当于在 `exchange` 函数中创造了这个实参的别名，它和 `main` 函数中的变量共享了相同的内存地址。于是我们可以直接用这个“别名”来修改实参的值了。这种方式又被称为按引用传递（Passing by reference）。

```
void exchange(int &a, int &b) { //接收两个引用参数  
    int tmp {a}; //临时变量  
    a = b;  
    b = tmp;  
}  
  
int main() {  
    int a {3}, b {4};  
    exchange(a, b); //传入实参  
    cout << a << ' ' << b; //输出，检验结果  
    return 0;  
}
```

最后的输出结果也合乎我们的预期，正是

---

4 3

这说明我们的确可以用它来完成数值交换的操作。

另外你可能还记得，我们在前面介绍了一个设置了默认参数的 `input_clear` 函数，用以清除错误输入。它的定义中就使用了引用参数：

```
void input_clear(istream &in = {cin}) { //不提供参数时使用默认参数cin  
    in.clear(); //清除错误状态  
    while (in.get() != '\n') //清除本行输入  
        continue;  
}
```

为什么要使用引用参数呢？这是因为，我为了清除 `cin`（或其它 `istream` 对象）的错误状态，必须要对实参作出修改。如果按值传递的话<sup>17</sup>，我修改的只能是 `in` 这个副本的状态。

引用参数传递还有一些其它的功用。很多时候我们希望把某个类型的对象（变量）传入函数中，我们不需要修改它的值，所以按值传递也可以。但是这个对象可能占用非常大的内存空间（通常是某种数据结构或类），如果在函数调用时要为它建立一个副本的话，那就既浪费内存空间，又浪费算力和时间。这时我们会选择这样定义函数：

```
void func(const Type &obj) {接收Type类型的常量引用  
    //...
```

<sup>17</sup>实际上这是不可能的，`istream` 类已经删除了拷贝构造函数，所以这个语法根本不能通过编译。

```
}
```

这样只是为传入的实参创建了一个别名而已，并不需要浪费大量的时间和空间来作复制工作。如果要防止误修改实参，我们也可以把它定义成常量引用，这样就不会出问题啦。

## 引用作为返回值

来看一看这个语句，它初看上去可能有点费解：

```
(num *= num) %= 11; //这是在做什么？
```

让我们用前面学过的运算符的有关知识来分析一下吧：

首先，`%=` 运算符把 `(num*=num)` 与 `11` 隔开。而在 `num*=num` 表达式中，`*=` 的作用是乘赋值，于是这个表达式的作用是把 `num` 变成 `num` 的平方。

下一步，因为赋值语句返回的是左操作数本身，所以 `(num*=num)%=11` 的作用相当于 `num%=11`。

我们可以试着写一两个函数来模拟这个语句的行为，比如说，一个叫乘赋值 `mul_ass`。它接收的第一个参数是引用，这是为了能够改变它的值；第二个参数是常量引用，这是因为我们无需用 `b` 来修改它的值。

```
int mul_ass(int &a, const int &b) {
    a = a * b;
    return a;
}
```

这样一来我们就可以用 `mul_ass(num, num)` 来实现和 `num*=num` 一样的功能了。

接下来我们应该把它的返回值扔到 `rem_ass` 当中了。

```
int rem_ass(int &a, const int &b) {
    a = a % b;
    return a;
}
```

这样我们就可以用 `rem_ass(num, 11)` 来实现和 `num%=11` 一样的功能了。

但是如果我们这么写，编译就会出现问题：

```
int num {5};
rem_ass(mul_ass(num, num), 11);
//error: cannot bind non-const lvalue reference of type 'int&'
//to an rvalue of type 'int'
cout << num;
```

编译器报错信息的意思是：“不能把 `int&` 型的左值引用绑定到 `int` 型的右值上。”关于左值右值的问题，我们暂不讨论；但是这个问题，我们需要解决。

问题的关键在于，函数返回的返回值，其实是一个“副本”，而不是 `return` 所跟的变量本身！因此我们是在对着一个不应该取引用的内容<sup>18</sup>按引用传递参数，那当然就会产生问题了。

这和我们在按值传递参数的过程中面临的窘境如出一辙。而解决方法也很相似，就是使用引用返回值。

```
int& mul_ass(int &a, const int &b) {
    a = a * b;
    return a; //返回值按引用传递，就不会再创建副本了。
```

<sup>18</sup>我们在精讲篇中会更详细地探讨此类问题。简而言之，不是所有数据都可以取地址的，也不是所有数据都可以被引用的。

```

}

int& rem_ass(int &a, const int &b) {
    a = a % b;
    return a; // 同上
}

int main() {
    int num {5};
    rem_ass(mul_ass(num, num), 11); // 现在它能正常运行了
    cout << num;
}

```

## 引用的类型与 `is_same`

我们发现，引用类型与基本数据类型有太多相同之处。比如说，我们可以把引用完全当作变量名来看待，从而我们可以读取或修改内容。还有，我们可以为引用再取一个别名，其效果相当于为原来的变量取一个别名：

```
int num, &ref {num}, &rref {ref}; // 效果等同于&rref{num}
```

再看地址，`num` 和 `ref` 的地址也永远相同，它们的内存大小可以用 `sizeof` 求得，这也是相同的。

看了这么多，我们发现，引用好像是一个分身，或者是真假美猴王那样的关系，我们根本分辨不清谁是本体，谁是别名。

那么，变量与引用的类型一样吗？其实是不一样的。`num` 是 `int` 类型无疑，而 `ref` 和 `rref` 都是 `int&` 类型的。这几个名字看似一模一样，但正主还是原变量，六耳也终究不是孙悟空。<sup>19</sup>

那么如何检验类型呢？我们可以用 `type_traits` 库的 `is_same` 来检验之。它是一个类模版，可以接收两个模版参数，并检验它们是否是同一类型。如果相同话，其静态成员 `value` 的值就是 `true`；如果不同的话，其静态成员 `value` 的值就是 `false`。<sup>20</sup>

```
// 需要包含头文件 type_traits
cout << is_same<int, int>::value << endl; // int 与 int 相同，故输出 1
cout << is_same<double, long double>::value; // 不同，故输出 0
```

提醒读者，`cout` 输出 `bool` 类型的值时，会默认以整数的形式输出。我们也可以用 `cout.setf(ios_base::boolalpha)` 来让它以布尔值的方式输出。

但是这里有另一个问题：尖括号（Angle brackets）中只能接收类型信息，我们不能直接把一个变量，或者引用，或者指针塞进去。这时我们就要用到 `decltype` 了。`decltype` 是一个编译时操作，它会解释出一个表达式的类型。

```
int num;
cout << is_same<decltype(num), int>::value; // 将输出 1
```

因此我们可以用 `decltype` 配合 `is_same` 来判断类型的差异了。

先来看一下 `num`, `ref` 和 `rref` 是不是同一个类型。

```
int num, &ref {num}, &rref {ref};
cout << is_same<decltype(num), decltype(ref)>::value << endl;
cout << is_same<decltype(ref), decltype(rref)>::value;
```

<sup>19</sup> 其实从汇编代码中看，C++ 中的引用全都是通过指针实现的。所以说，在定义引用的过程中，程序上会创建一个我们看不见的指针，包括传引用参数等操作，本质上都是传指针。关于这里的细节，我就不多谈了。

<sup>20</sup> 这里出现了很多新概念，比如类模版，模版参数、静态成员等。读者无须知道细节，我们会在后面慢慢道来。

这两句的输出分别是 0 和 1，说明 `ref` 和 `rref` 都是同一个类型的，它们都和 `num` 不是同一个类型的。

接下来我们具体看一下它们三个分别是什么类型。

```
cout << is_same<decltype(num), int>::value << endl;
cout << is_same<decltype(ref), int&>::value << endl;
cout << is_same<decltype(rref), int&>::value << endl;
```

这三个输出的结果全部为 1，说明 `num` 是 `int` 类型的，而 `ref` 和 `rref` 是 `int&` 类型的，它们并不相同。

`is_same` 是一个很实用的类型判断工具，我们在后面也会用到它的。

## 5.4 一维数组

在前面的章节中，我们用各种数据类型、运算符和选择结构，可以处理一些小规模的问题。循环结构允许我们用少量的代码处理一些规模较大的问题，但是我们所掌握的数据方面的知识太少，还没有跟得上这样突飞猛进的处理能力。试想，如果我们需要 100 个变量来存储不同信息，并在我们需要时对它进行处理，那么“如果把这 100 个变量存下来”就成了我们的难题。我们要定义 100 个变量吗？这样写起来好像也很麻烦吧。而且更可怕的是，如果我们也不能确定需要多少个数据（可能 100 个，或者 100000 个），那怎么办？我们要怎么搞到这么多变量名啊？

且慢。我在本章中一再提醒读者，变量名并不是变量的本质。如果要读取或者存储某个变量的信息，我们是可以不需要变量名的。假如说现在有三个 `float` 型的变量（每个变量占 4 字节内存），我暂且用 `a`, `b`, `c` 来称呼它们，如果它们在内存中的地址是挨着的，比如图 5.10 中的这种情况，我们可以定义一个 `float*` 类型的指针 `pf=&a`，让它指向 `a`。

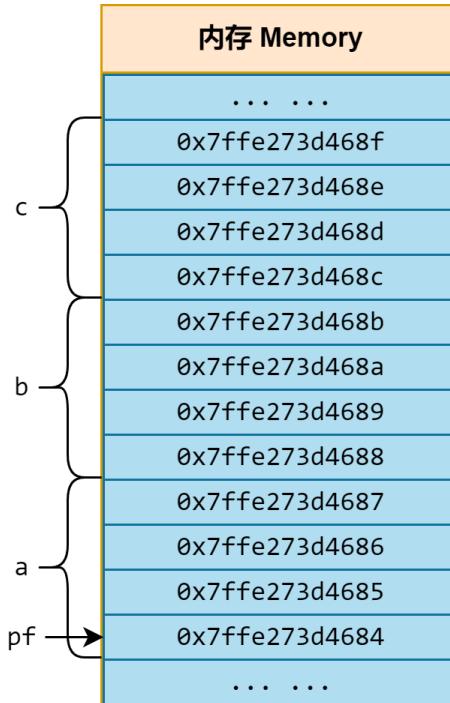


图 5.10：如果三个变量在内存中的地址是挨着的，我们可以用一个指针来访问这三个数据

接下来我们就可以用这一个指针访问这三个数据。还记得我们讲过指针的运算吧，`pf+1` 相对于 `pf` 偏移了一个数据（而非一个字节）。既然 `pf` 是 `a` 的地址，那么 `pf+1` 就是 `b` 的地址，`pf+2` 就是 `c` 的地址。

试想一下，如果有 100 个同类型数据，它们都是紧挨着的，那么我们用一个指针 `p` 指向第一个元素，那接下来不就可以用 `p+index` 表示第 `index+1` 个数据的地址了吗？<sup>21</sup>

不过 C/C++ 标准从来就没有保证过“连续定义的若干变量在内存中必须是紧挨着的”。所以我们需要一点点手段来确保这个前提——这就是数组（Array）。

本节我们先来介绍一维数组，高维数组留待后面再讲。

## 一维数组的定义和使用

数组是一大类复合类型，它可用于批量定义数据。定义一个数组的基本的语法是

```
<类型> <数组名>[<数组长度>] = {<初始化列表>};
```

其中的长度是一个 `size_t` 类型的常量表达式<sup>22</sup>。话虽这么说，但因为整型数据都可以隐式类型转换成 `size_t` 类型，所以我们直接用整型的常量表达式来定义就可以。

以下是一些正确的定义方式：

```
double array1[3] = {1e5, .3, -1}; // 正常格式，可省略=号
int array2[4] {1}; // 省略部分或全部内容，省略的部分使用默认值(0)
float array3[] {0.1f, 0.2, 0.3d}; // 省略长度，数组长度会被定为{}内的数据个数
```

而以下的定义方式是错误的：

```
int array4[3] {1, 2, 3, 4, 5}; // 初始化列表中的数据个数超过长度限制
int array5[5] {1, , , ,1}; // 不能略过中间的数据，定义后面的数据
int array6[] {}; // C++ 标准禁止使用长度为0的数组
```

数组是什么类型呢？看上去好像很神秘，但如果我们试着输出这个数组的名字，就会发现，它输出的东西很像是一个地址值<sup>23</sup>。

```
cout << array1; // 输出 0x7fff32b2044c 或者别的，总之很像一个地址
```

事实上，在我们日常使用数组的过程中，大部分情况下数组类型都会隐式类型转换成对应类型指针。刚才的输出语句就是一例，在输出之前，`double[3]` 类型的 `array` 会被隐式类型转换为 `double*` 类型，然后作为一个地址值输出。

在涉及加减法时亦如此。`array2` 原本是 `int[4]` 类型的，而在与整型数据进行加减时，就会先隐式转换成 `int*` 类型，然后作为一个指针来进行加减。

不难发现，数组正好有我们在本节开头中所希望的那种性质：其一，一个数组包含了若干连续数据，它们紧挨着排布；其二，数组名可以视作一个指针，我们可以用 `p+index` 的形式来读取或修改它的值。

那么如果我要定义一个大小为 100 的 `int` 数组，并把每个值都变成 0，我可以如何做呢？

```
int array[100] {}; // {} 的中省略了全部的初始化，这样每个数据都会初始化为0
```

如果我要定义一个大小为 10000 的 `int` 数组，并把每个值都变成 1，我又该如何做呢？这里不太一样，我们无法用简洁的语法直接把所有数据都初始化为 1，所以需要另想方法。这里我们可以用 `for` 循环来批量处理这些数据，这时就能体现出循环语句的强大了。

```
int array[10000]; // 无需初始化，反正待会就要赋值
for (int i = 0; i < 10000; i++) // 注意循环的起止条件
    *(array + i) = 0; // array+i 即是第 i+1 个数据的地址，再取内容，就可以修改了
```

<sup>21</sup>注意，`p+0` 是第一个数据的地址，所以以此类推，`p+index` 是第 `index+1` 个数据的地址。

<sup>22</sup>现代编译器大都支持数组定义时使用非常量的长度，此时编译器会用一套比较复杂的机制来创建这个数组。但是我仍建议读者在需要使用动态大小的数组时选择动态内存分配语法。

<sup>23</sup>字符型的数组依然是一个例外，我们会在后面讲到它。

注意，`array` 是第一个数据的地址，所以 `array+1` 是第二个数据的地址，……，以此类推，第 10000 个数据的地址应该是 `array+9999`。因此在这里我们需要控制循环的起始和终止条件，让 `i` 从 0 开始，到 9999 终止<sup>24</sup>。

用 `*(array+i)` 这样的方式来访问显得有点麻烦了。在 C/C++ 中，我们还有一种方式可以访问数组的数据，那就是用下标运算符 `[]` 来实现这个功能。在 C/C++ 中，如果 `arr` 是一个数组类型，那么我们可以用 `arr[i]` 的方式来替代 `*(arr+i)`，效果相同。这个语法还可以扩展到指针层面，如果 `p` 是一个指针，我们也可以用 `p[0]` 的方式来替代 `*p`。<sup>25</sup>

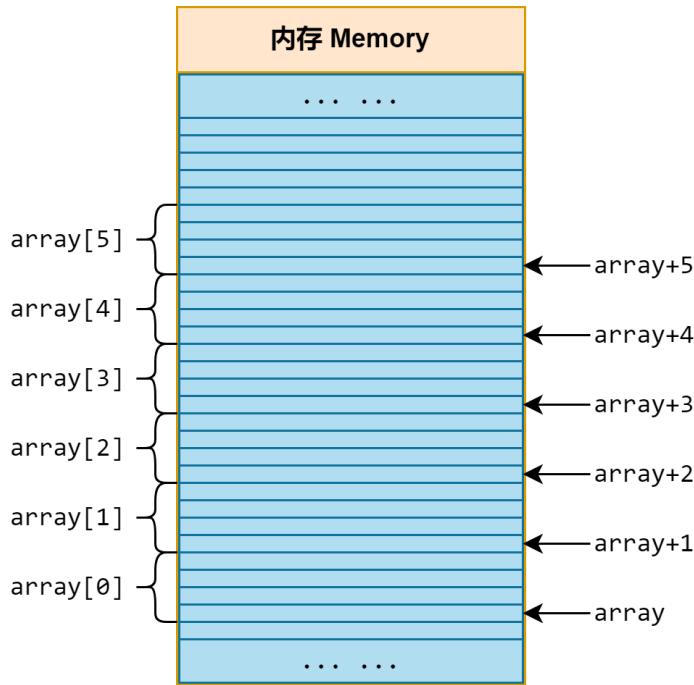


图 5.11: 用 `array[i]` 效果等同于 `*(array+i)`

有了数组之后我们就可以只需要一个名字 `array`，然后通过下标来读取或修改大量数据了。当然，不要忘记数组下标是从 0 开始的，所以 `array` 数组的第一个数是 `array[0]` 而不是 `array[1]`。下面是一个对数组中各数字进行求和的例子，我们需要用户先输入 10 个数，然后求出它们的和。

```
int arr[10], sum {0}; // 定义一个数组arr和int数据sum, sum初始化为0
for (int i = 0; i < 10; i++) // 注意条件
    cin >> arr[i]; // 输入arr[i]的值, arr[i]是arr数组第i+1个数
for (int i = 0; i < 10; i++)
    sum += arr[i]; // 用加赋值运算符, 把arr[i]的值加到sum中
cout << "总计: " << sum;
```

我们可以试一下这段代码的运行效果：

---

**1 2 3 4 5 6 7 8 9 10**

总计: 55

---

它的运行结果也很符合我们的预期。

<sup>24</sup>虽然我们可以访问 `array+10000`，但我们无法保证这个空间是安全的。换言之，使用 `array+10000` 会带来类似于野指针的问题。

<sup>25</sup>实际上，对于数组名或指针来说，`p[i]` 形式的表达式都会被编译器当作 `*(p+i)` 来处理，所以这么写只是会让我们自己方便一点而已，对于编译器来说没会么区别。

## 数组参数传递

我们在前面写了一个 `maximum` 函数模版，它可以求两个数的最大值。现在我们想要求大量数据的最大值，再用 `maximum` 重载的方法就显得太不合适了（即便我们有一个能传递任意数量参数的函数，我们也要衡量一下调用的时候是不是要把一个个参数全都写进去）。为了解决这个问题，我们需要考虑能否用数组来实现这个功能。

在 C++ 中，我们可以传递一个数组的首地址，作为函数的参数。这里“数组的首地址”，指的就是数组在内存空间中的第一个字节对应的地址。从地址值上讲，它就等于 `&array[0]`——我们之后再来谈它们在类型上的区别，目前读者这样理解就足够了。

以下是一个实现此功能的代码，可以接收 5 个浮点数输入，然后给出其中最大的数。读者可以先试着运行它，然后我将分别对语法和语义进行讲解。

```
template<typename T>
T maximum(T arr[], unsigned n) {
    T res {arr[0]}; // 先假设最大值为 arr[0]
    for (int i = 1; i < n; i++)
        if (arr[i] > res) // 如果 arr[i] > res, 说明有更大的数
            res = arr[i]; // 那么目前能找到的最大值就是 arr[i]
    return res; // res 就是返回值
}

int main() {
    double arr[5]; // 定义一个数组 arr 和 int 数据 sum, sum 初始化为 0
    for (int i = 0; i < 5; i++)
        cin >> arr[i];
    cout << maximum(arr, 5); // 传入 arr 数组，并告诉这个函数 arr 有多大
    return 0;
}
```

从语法上，我们可以看到，`maximum` 是一个函数模版，它可以接收一个 `arr` 数组和一个 `n` 无符号整数作为参数。你可能会好奇，为什么我们要把形参写成 `arr[]` 的形状呢？写成 `arr[5]` 不行吗？

可以，但没有意义。在数组参数传递的过程中，我们传递的不是数组本身，而是它的首地址——换句话说，传递的是一个指针。在编译器进行编译的时候，它会把 `T arr[]` 参数视同 `T *arr`，所以即便我们在 `arr[]` 中写了 5，也会被忽略，没有意义。

也正因为数组参数传递过程中只传递了指针，所以函数不能判断这个数组的大小，因此我们必须再加一个参数，来描述这个数组的大小。这就是 `n` 存在的意义。

再来看语义，`maximum` 函数中的这段代码是如何做到我们希望的功能的呢？其实就是用 `for` 循环读取 `arr` 中的每个数，然后逐一比较。如果发现了一个更大的数，那就说明最大值至少应该是这个数，所以 `ref=arr[i]` 赋值一下。如果之后又发现了一个更大的数，那就说明最大值至少应该是这个数，所以 `ref=arr[i]` 再赋值。这样一直找到最后，`res` 就是所有数的最大值了。读者若是还未明白，可以把自己想像成程序，以输入 `2,1,4,5,3` 为例，模拟一下这个过程。

有些时候，我们也不知道输入数据有多少个<sup>26</sup>，那么怎么办呢？为了不出问题，我们可以设计一个尽可能大的数组（比如，大小为 1000 的数组 `arr`），然后让用户输入尽可能多的数据。如果用户输入了字母 `q` 或者类似的不合理内容，就结束输入循环。这个思路与[代码 3.3](#)的判断方式很相似。

我们还需要考虑一种情况，用户的输入可能超过 1000 个，这时如果不加防范，程序会继续将输入存到 `arr[1000], arr[1001]` 这些地方。这会造成野指针问题，不得不防范！所以我们还需要在循环时判断一下，数组下标是否越过了边界。

---

<sup>26</sup>这种情况最常见于字符串的输入。

鉴于此，我们可以写一个输入函数。我们可以用 **for** 循环或者 **while** 循环，实现的方式也可以有细微不同，只要能以简单高效的方式达到我们的目的就好。以下是第一种写法：

```
double arr[1000]; //为了能接收任意数量的输入，预先设置一个比较大的数组
unsigned size = 0; //定义一个size变量，用来表示数组中目前的数据量
for (; size < 1000 && cin; size++) { //注意判断条件
    cin >> arr[size]; //输入到arr[size]中，然后size会自增
}
cout << maximum(arr, size); //现在是长度为size的arr数组，传入函数中
```

这里使用 **for** 循环，其中的判断条件是个逻辑与表达式。之所以用 `size<1000` 而不是 `size<=1000` 是因为，条件判断是在输入 `arr[size]` 之前进行的，而迭代是在输入之后进行的。如果某次 `size` 的值为 999，那么本次判断可能为 **true**，输入正常进行，输入了 `arr[999]` 之后，`size` 才迭代为 1000。而如果某次 `size` 的值为 1000 了，这时倘若用 `size<=1000` 来判断，那就会出现 `arr[1000]`，这就越界了！

我们还可以使用 **while** 语句来实现，思路大同小异。

我们也可以把这个输入功能做成函数，就叫 `input_arr`，它接收一个数组参数（其实是指针参数），再接收一个数组容量参数（这个数组最大的容量是多少）。考虑到用户的输入未必能达到最大值（“容量”与“实际储量”当然不一定相同），我们还需要知道这个数组实际装了多少数据。我们可以把这个数据作为 `input_arr` 的返回值。

这个思路比较清晰，我们可以直接写一个这样的函数出来：

```
template<typename T>
unsigned input_arr(T arr[], const unsigned max_size) {
    unsigned real_size {0}; //记录数组中数据的实际个数
    while (real_size < 1000) { //防止数组下标越界
        cin >> arr[real_size]; //输入arr[real_size]
        if (!cin) { //如果输入状态错误
            input_clear(); //先清理输入
            break; //再退出while循环
        }
        ++real_size; //然后real_size自增
    }
    return real_size;
}
```

这个函数略显复杂，注意事项也比较多，我们慢慢道来。

首先，它是一个函数模版，会根据我们的需要生成对应类型的函数实例。这里没什么可说的。

`max_size` 是数组最大容量。它只是一个副本，我们改动它也不会影响原变量，但是我依然建议使用 **const**，以避免在函数中误修改它的值。至于数组传递，它本身就是一种指针传递，所以我们直接在 `input_arr` 函数中修改 `arr` 数组的数据就行了，而不需要什么所谓“对数组的引用”。

在 **while** 语句中我们只判断 `real_size` 是否越界。而对于异常输入的情况，我们选择使用 `if(!cin)` 判断加 **break** 语句来退出循环。

语法正确还不够。你还需要考虑清楚 **while** 循环中各语句的执行顺序，以免出现语义错误。例如 `++real_size` 这一句，必须要在 `if(!cin)` 之后，这两句的顺序是不能调换的。这是因为，只有当 `cin` 正常的时候，`arr[real_size]` 才能得到有效输入，这时 `real_size` 才应该自增；否则，`arr[real_size]` 得不到有效输入，`real_size` 也不该自增，所以应当直接用 `if(!cin)` 中的 **break** 语句退出这个循环，避免执行 `++real_size`。

接下来我们可以使用这个函数，配合 `maximum` 来实现先输入，后求最大值的功能了。

```
int main() {
    int arr[1000];
    unsigned arr_size {input_arr(arr, 1000)};
    //向arr提供输入，并由arr_size接收返回值，记录当前arr数组中的有效数据个数
    cout << maximum(arr, arr_size);
}
```

如果我们发现用户实际输入的需求可能是几万个，而非几个到几百个，那我们就要考虑增加 `arr` 的容量。我们可以把定义句改为 `int arr[100000]`。

这就完了吗？不，还没有，我们还需要改 `input_arr(arr, 1000)` 这里的参数呐！把它也改成 `100000` 才行。

这就体现出单纯用字面量的缺点了，我们没法保证一次修改之后所有相关数据都改变，而人为操作的话，出现遗漏和失误的可能性又太大了。为了保持统一性，我们可以用常量或常量表达式，通过具名的方式把数组的容量描述出来。

```
constexpr int Maxsize {100000}; //用constexpr也可
int arr[Maxsize];
unsigned arr_size {input_arr(arr, Maxsize)}; //这样就好了
```

这样，如果我们要改 `Maxsize` 的值，只要修改一次，就可以把所有需要改的数据一并改完了，犯错的可能性大大降低。

关于输入和求最大值，我们还可以有一些更有技巧性的写法：

```
const int Maxsize {100000}; //用constexpr也可
int arr[Maxsize];
cout << maximum(arr, input_arr(arr, Maxsize)); //?
```

在这里，我们是把输入和求最大值并到同一步来写了。这句可能有点乱，但我们分析一下就会发现它很好懂。

首先，`maximum` 接收两个参数，一个是 `arr`，而另一个是 `input_arr(arr, Maxsize)`。这也就是说，我们需要把 `input_arr` 的返回值求出来，然后才能执行 `maximum` 函数。而在 `input_arr` 执行过程中，程序会先进行输入（这是 `input_arr` 函数的副作用），然后返回实际存储数据的个数。这个个数将作为参数被 `maximum` 函数使用。

下一步，`maximum` 知道了 `arr` 数组的位置和实际数据的个数，然后就会在对应的范围内求出最大的数。这时的 `arr` 也已经保存了用户输入的值（这是 `input_arr` 的副作用），所以整个程序就这样顺理成章地运行下去了。

代码 5.1: 输入数据求最大值.cpp

```
//这是一个输入若干数求最大值的程序，用户可以输入最多100000个数据，输入q结束输入
#include <iostream>
using namespace std;
//声明部分
void input_clear(istream& = {cin}); //在声明中就给出默认参数，注意类型是引用
template<typename T>
T maximum(T[], int); //T[]写成T*也没问题，不过习惯上还是写成数组形式
template<typename T> //注意：每次声明（定义）模板时必须写一个template
unsigned input_arr(T[], const unsigned);
//定义部分
```

```

int main() { //主函数
    const int Maxsize {1000000}; //用constexpr也可
    int arr[Maxsize];
    cout << maximum(arr, input_arr(arr, Maxsize));
    return 0;
}

void input_clear(istream &in) {
    in.clear();
    while (in.get() != '\n')
        continue;
}

template<typename T>
T maximum(T arr[], int n) {
    T res {arr[0]}; //先假设最大值为arr[0]
    for (int i = 1; i < n; i++)
        if (arr[i] > res) //如果arr[i]>res, 说明有更大的数
            res = arr[i]; //那么目前能找到的最大值就是arr[i]
    return res; //res就是返回值
}

template<typename T>
unsigned input_arr(T arr[], const unsigned max_size) {
    unsigned real_size {0}; //记录数组中数据的实际个数
    while (real_size < 1000) { //防止数组下标越界
        cin >> arr[real_size]; //输入arr[real_size]
        ++real_size; //然后real_size自增
        if (!cin) { //如果输入状态错误
            input_clear(); //先清理输入
            break; //再退出while循环
        }
    }
    return real_size;
}

```

这应该是本书迄今为止展示的最长代码了，但是关于它的每个部分，我都已经拆解过（除了 `input_clear`，这部分我们暂且不讲，读者知道怎么用即可）。当我们想要实现较复杂的功能时，长代码总是难以避免的。这也正是我们需要用函数来进行“模块化”的一个重要原因。

## 范围 `for` 循环

有的时候我们可能需要对一个数据的数据进行批量处理，这时我们一般用 `for` 循环来实现。好处也很明显，`for` 循环内部可以定义一个下标变量 `i`（或者用别的名字），终止条件可以通过比较运算来实现，而迭代操作刚好可以处理下标的变化。这样看来，`for` 循环仿佛是为数组处理量身打造的。

从 C++11 标准起，范围 `for` 循环（Range-based for loop）能让我们一次性读取（或修改）数组中所有数据的操作变得更简单。比如说，我们定义一个数组，然后输出它的所有数据，可以这样写：

```

int arr[] {1,2,3}; //定义一个长度为3的数组
for (int x : arr) //用范围for循环，变量x接收arr中的数据
    cout << x << 'u'; //输出x的值

```

这样一来，范围 `x` 就会循环接收 `arr` 中的每个数字，然后在循环体内进行操作。

如果我们要修改它的值呢？把 `x` 定义成 `int` 类型的变量是不行的，因为它相当于一个临时变量。因此我们需要使用引用来实现修改的功能。

```
int arr[] {1,2,3,4}; // 定义一个长度为4的数组
for (int &x : arr)
    x *= x; // 将x的值变为原数的平方
```

这个范围 `for` 循环看上去很好用，不过它也存在较多的限制：

- 范围 `for` 循环会对数组中的所有数据进行处理，我们无法人为改变它的处理范围。这也就意味着，如果某个数组中的数据不都是有效数据，那么这种方法可能会把无效数据一并处理。比如说，输出了不该输出的部分，或者是把不该平方的数据也给平方了。即便这样做不会带来任何危险，它也会在实际运行的过程中带来效率问题（因为我们把一些时间浪费在了不该处理的数据上）。
- 范围 `for` 循环不能处理那些“看上去是数组，但实际是指针”的东西。<sup>27</sup>比如说，函数中接收的“数组参数”，其实它们全都是指针。那也就意味着，代码 5.1 中的 `maximum` 和 `input_arr` 都是无法使用范围 `for` 循环的。我们在后面接触动态内存分配时也会发现这个问题，动态数组不能使用范围 `for` 循环来处理数据。

## 数组的类型

那么数组和指针之间到底是什么关系呢？为什么数组就可以用范围 `for`，指针就不能？为什么数组名可以隐式类型转换成指针？

如果我们用 `is_same` 来比较一个数组类型和一个指针类型，就会发现它们确实是不相同的。

```
int arr3[3], int *p;
cout << is_same<decltype(arr3), decltype(p)>::value; // 输出为0
```

那么我们试试比较一下两个长度不同的数组类型呢？

```
int arr3[3], arr5[5];
cout << is_same<decltype(arr3), decltype(arr5)>::value; // 输出也为0
```

哎，看起来这两个类型也不相同。

没错，数组类型的区分比指针更细。对于指针来说，指向 `int` 的指针当然和指向 `long long` 的指针是不同类型，但是任何两个指向 `int` 的指针都是同类型的。

数组不然，不光数据类型不同的数组不是同类型，连长度不同的数组也不是同类型了。这里的 `arr3` 的真实类型应该是 `int[3]` 型，而 `arr5` 的真实类型应该是 `int[5]` 型。

可惜 `is_same` 在这里能告诉我们的信息还是太少了，我们只能知道“某两个类型具体是什么”，但还不清楚“某个类型意味着什么”。所以我们还有另一个工具，那就是 `typeid` 运算符。

`typeid` 有点像 `sizeof`，它既可以作用于类型，又可以作用于对象（变量）。`typeid` 的成员函数 `name()` 可以返回一个 `const char*` 对象（这是一个字符串，详见下一节）。我们输出它，就可以看到相关的类型信息了。以下是一个示例：

```
int arr3[3], arr5[5];
cout << typeid(arr3).name() << endl;
cout << typeid(arr5).name() << endl;
cout << typeid(int).name() << ' ' << typeid(double).name() << ' '
```

<sup>27</sup>这是因为，指针相比于数组，缺少了“数组长度”这条信息，所以范围 `for` 循环根本不能确定数组的实际大小。详见后文。

```
<< typeid(unsigned).name() << ' ' << typeid(float).name() << endl
<< typeid(int*).name() << ' ' << typeid(char*).name();
```

需要提醒读者，`typeid(...).name()` 在不同编译环境下的结果可能不同<sup>28</sup>，因此本例中的结果取 Coliru 上运行的结果。读者也可以在 Coliru 上自行运行这段代码试试效果。它的输出是

---

```
A3_i
A5_i
i d j f
Pi Pc
```

---

其中 `A3_i` 格式的内容表示一个长度为 3 的 `int` 型数组（Array），`A5_i` 表示一个长度为 5 的 `int` 型数组。

而接下来的 `i, d` 等分别是 `int, double` 等类型的名字。

最后输出的 `Pi` 和 `Pc` 分别是两个指针类型的名字，其实就是在原类型名字前面加上了 `P`，代表指针（Pointer）。

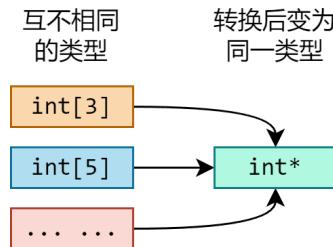


图 5.12: 数组到指针的类型转换

也就是说，数组比指针多出了一个“长度”信息。虽然我们一般看不出这个信息来，但是它确实存在。不信的话，你还可以用 `sizeof` 取数组的内存空间大小看一下，它是不是与数组长度挂钩。

```
int arr3[3], arr5[5];
cout << sizeof arr3 << endl << sizeof arr5;
```

而指针不是这样的。指针也是变量，它在内存空间中也有对应的存储位置<sup>29</sup>。我们可以用 `sizeof` 计算一下。

那么来到数组参数传递这里——我们也可以用 `typeid` 取“数组形参”的类型，或者用 `sizeof` 来取它的内存空间大小，就会发现它的表征和指针是一样的：

```
void fun(int arr[]) {
    cout << typeid(arr).name() << ' ' << sizeof arr;
    // 输出 Pi 8, 说明 arr 是一个指针!
}
int main() {
    int arr[1000];
    fun(arr);
}
```

<sup>28</sup>实际体验是，在 MSVC 开发环境中给出的结果最为清晰。

<sup>29</sup>因此我们还可以取一个指针的地址，这是后话。

所以说数组参数传递看似是在传递数组，其实是在传递指针。而在传递的过程中，数组原本所含有的“长度”信息就丧失了，因此我们在传递数组的过程中，最好要另外加上一个参数，说明一下数组的长度，否则函数也不能确定数组的长度，就难免会发生下标越界问题。

## 5.5 字符串

现在我们来看一种特殊类型的一维数组：字符串。字符串也是一大类，不过它们的道理很相似，所以我们只研究其中一个部分：`char` 数组。

定义一个 `char` 数组的方法和定义一般的一维数组差不多：

```
char str1[5] = {'J', 'a', 'c', 'k', '\0'}; // 正常格式，注意末尾最好用 '\0'
char str2[10] {"A", "m", "y"}; // 省略部分内容，省略的部分使用默认值 ('\0')
char str3[] {65, 66, 67, '\0'}; // 除非末尾用 '\0'，否则不建议这么写！
```

唯一的例外是，我们可以用字符串字面量来初始化字符数组。

```
char str4[5] {"Jack"}; // 效果等同于上述 str1 的定义
```

这是一种语法糖<sup>30</sup>，实际上编译器会把它解释成和 `str1` 相同的初始化方式。不过呢，这种针对于字符数组的语法糖让我们初始化内容变得更简单了。然而，这种语法糖仅在初始化时有效。到了赋值的时候，我们不能这样写：

```
str1 = "Bob"; // 错误！
```

要想改变字符串的内容，我们只能通过给每个元素赋值的方式来进行：

```
str1[0] = 'B';
str1[1] = 'o';
str1[2] = 'b';
str1[3] = 0; // 不要忘记！
```

但是这种逐个赋值的操作可能出问题，我们很容易漏掉 `str1[3]=0` 这句。所以我们推荐使用 `cstring` 库中的函数 `strcpy` 来实现。我们暂时先不讲这个，留到后面再说。

```
strcpy(str1, "Bob"); // 正确
```

字符串的使用和其它类型的数组很相似，但又有它的特点。所以我们在里先来介绍一下关于字符串的最基本概念。

### 字符串是什么？

我们最熟悉的字符串就是我们一直在使用，但是尚未讲解过的字符串字面量。很多时候我们都要输出字符串字面量。

```
cout << "HelloWorld!";
```

这里的 `"HelloWorld!"` 就是一个字符串字面量。

字符串字面量不同于基本数据类型的字面量，它是一个数组，我们可以像其它数组名那样去输出它的地址。但是当我们用 `cout` 输出一个字符串字面量的时候，它输出的并不是地址，而是这个字符串的内容啊！

---

<sup>30</sup> 语法糖（Syntactic sugar），是指某种编程语言中的某些语法，它们对语言本身的功能没有影响，但是更方便程序员使用。语法糖可以让程序更简洁，有更高的可读性。

类似的问题，我们在指针中也遇到过。当我们试图输出一个指针的时候，实际输出的可能是任何东西，但是怎么看都不像个地址值。这是因为，`<<`对于`ostream`和`const char*`类型有特殊的重载。在遇到`char*`及其变体的类型时，会把它按照字符串的方式来输出，而不是输出它的地址。

为了看到它的地址，我们可以通过显式类型转换，把`char*`类型转换成`void*`类型然后输出，这样就能看到地址值了。

```
cout << (void*)"HelloWorld!"; // 输出 0x4006a4
```

字符串字面量存储在数据段（Data section），这里的内容都是常量，所以字符串是一个常量`char`数组。我们可以用`typeid`检验。<sup>31</sup>

```
cout << typeid("HelloWorld!").name(); // 输出 A13_c
```

或者是用`is_same`检验。但是这里出现了新问题，我们直接写成下面这样是行不通的：

```
cout << is_same<decltype("HelloWorld!"), const char[13]>::value;
// 输出 0, 怎么回事?
```

具体的情况比较复杂，这涉及到`decltype`的问题<sup>32</sup>，读者不必深究。我们只需改成这样就好：

```
cout << is_same<decltype("HelloWorld!"), const char(&)[13]>::value;
// 输出 1
```

那么既然字符串字面量是一个字符数组，我们岂不是可以用下标运算符来访问它的某一个元素？

```
cout << "HelloWorld!"[0]; // 将输出 H
```

## 字符串是如何构成的？

细心的读者可能会发现另一个问题：这个`"HelloWorld!"`带空格和标点也只有 12 个字符，为什么它的长度是 13？这就不得不提到字符串的构成了。

在 C/C++ 中，所有的字符串都应当以`'\0'`，也就是 ASCII 码值为 0 的字符结尾。这是我们判断一个字符串有效信息长度的最佳标准。为什么这么说呢？

当我们定义字符串时，因为我们也不确定它的有效信息有多长，所以我们可能要给它比较大的容量。

```
char name[33]; // 难免会有名字很长的人，所以容量要大一点
```

但是这里就会存在一个问题：绝大部分情况下，这个字符串真实存储的信息长度肯定很少，比如说“Amy Smith”，这个名字只有 9 个字符，那么我们就还需要一个类似于`size`的变量，来说明`name`的有效信息长度。问题在于，这种方法比较浪费空间，我们还需要另外定义变量来存储它的长度；而且也很不方便使用，我们每次传参都要额外加一个变量。

C/C++ 有更好的解决方法，那就是用`name`数组内的信息来表示有效长度！具体的方法是，我们在有效信息的末尾，把下一个字符设置成`'\0'`。这样，无论哪个函数拿到了这个字符串，它只需要从`name[0]`开始数一数，看数到哪里`name[i] == '\0'`，那么自然就知道了这个字符串的长度，而不需要用另外的参数来记录了。

举个例子，现在我记录一个名字`"Alexander Alexandrovich Stepanov"`。这个名字碰巧有 32 个字符（含空格），加上末尾的结束符共有 33 个。现在我们把它存入`name`中。

```
char name[33] {"Alexander Alexandrovich Stepanov"};
```

这时它的 33 个字节全部都是有效信息，如图 5.13 第一行所示。

<sup>31</sup>MSVC 中能看出它是`char const [13]`类型，但 GCC 中只有`A13_c`，看不出常量性，只能知道它是一个`char`数组。`is_same`的检测相对更细致，不过它也有它的问题，见下文。

<sup>32</sup>按标准，`decltype("HelloWorld!")`返回的类型是一个对`const char[13]`的左值引用类型。

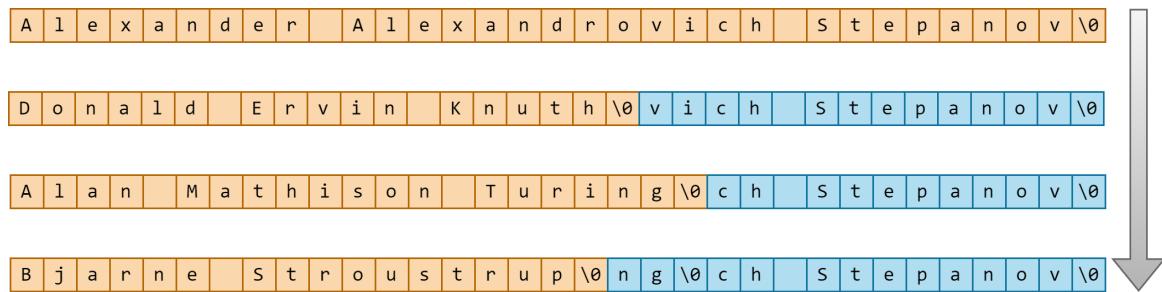


图 5.13: 字符串 name 中信息的变化情况

那么接下来我通过 `strcpy`<sup>33</sup> 把另一个名字复制到 `name` 中，这样做会修改原来字符串的信息。

```
strcpy(name, "Donald\u00D7Ervin\u00D7Knuth");
//strcpy 会把第二个参数中的有效部分复制到第一个参数起的内存空间中
```

现在 `name` 的内存空间应该如同图 5.13 第二行所示。注意，只有第一个 '`\0`' 及以前的部分是有效信息（橙色），而后面的部分不是有效信息（蓝色），我们也没有浪费时间去修改它的必要，让它们维持原状就好。

接下来我再次通过 `strcpy` 把另一个名字复制到 `name` 中。

```
strcpy(name, "Alan\u00D7Mathison\u00D7Turing");
//strcpy 会把第二个参数中的有效部分复制到第一个参数起的内存空间中
```

现在 `name` 的内存空间应该如同图 5.13 第三行所示。下一步我再修改 `name` 的值，就应该得到图 5.14 第四行的情况了。

```
strcpy(name, "Bjarne\u00D7Stroustrup");
```

这就是字符串的基本结构。如果简单点说就是，字符串总是要有一个 '`\0`' 结束符。那么如果我们要输入一个长度为 `n` 的字符串，考虑到末尾结束符的存在，我们就至少需要容量有 `n+1` 的字符串才行。

了解了这些之后，我们就可以自己写一些简单的字符串处理函数了。

## 字符串的处理

知道了字符串的结构之后，我们就可以写一些简单的函数来进行字符串的处理了。

`cstring` 库中有一些常见的字符串处理函数——有些我们也可以写，比如 `strlen`。这个函数可以计算一个字符串有效信息的长度（不包含结束符）。比如说，`sizeof("Amy")` 得到 4，这是它占用内存的大小；而 `strlen("Amy")` 得到 3 这是它外表展现出来的长度。

为了避免名字冲突，我们就写一个 `StrLen` 函数，它可以接收 `char` 指针<sup>34</sup>，并返回一个无符号的整数，作为长度。

来考虑一下这个函数的外在特征。我们只需提供这个字符串就可以了，无需更多信息。不过我们只想求值，不需要改变它，所以干脆设成指向常量的指针，稳妥一些。至于返回值，`strlen` 内部使用的是 `size_t` 类型作为返回值，其实我们用 `unsigned` 或者 `unsigned long long` 也行。

```
unsigned StrLen(const char*); // 声明完毕
```

<sup>33</sup>MSVC 会默认阻止我们使用 `strcpy` 和 `strncpy` 函数。为了解决这个问题，我们可以更改设置，或直接在源代码的最前面加一行 `#define _CRT_SECURE_NO_WARNINGS 1`。

<sup>34</sup>一般来说，我们在涉及其它类型的数组参数时会使用数组形式的形参，以便告诉用户它接收的是数组（虽然实际情况还是转换成指针了）信息；而在涉及字符串时，我们更习惯于把参数设为指针类型。

接下来考虑怎么求出它的长度。其实很简单，我们从 `str` 指向的第一个字符开始找，直到找到 '`\0`' 为止，走过了多少个字符，就说明它的长度是多少啦。所以我们可以用一个循环结构来实现这个功能。

```
unsigned StrLen(const char* str) {
    for (unsigned i = 0; ; i++) { // 条件留空将默认为 true
        if (str[i] == '\0') // 如果 str[i]==0, 说明字符串结束了
            return i; // 直接返回 i 的值
    }
}
```

写完了之后一定不要得意忘形，马上在主函数中写一个例子验证一下，看看是否合乎我们的预期。比如 `StrLen("Amy")`，它的值就应该是 3，我们可以输出一下看看对不对。

接下来我们再写一个稍难的函数。`strcpy` 函数可以把一串内容复制到另一个字符串中，效果等同于“字符串间的赋值”，这样可以免于让我们用逐个字符赋值的麻烦方法来改变整个字符串的值。它接收两个参数，第一个参数 `dest` 是目标字符串，第二个参数 `src` 是内容源。这个函数不需要考虑 `dest` 的容量是否足够，只管往里复制就行<sup>35</sup>。

换句话说，我们只需要把 `src` 串中的有效部分逐个赋值给 `dest` 的对应部分就行。比如说，`dest[0]=src[0]`，然后 `dest[1]=src[1]`，诸如此类。终止条件是 `src[i]=='\0'`，这时我们就可以停止了。但是最后不要忘了人为地在后面补上一个 '`\0`' 结束符，否则这个字符串就没有结束。

所以我们就来模仿赋值语句的方式来写一个 `StrCpy` 函数。它的第一个参数是 `char*` 类型的，第二个参数是 `const char*` 类型的（因为不需要修改，那么就从定义层面防止修改），而返回值是 `char*` 类型的，它的值（地址）应该等于第一个参数的值（地址），这是为了实现类似于赋值运算符的那种连续赋值语法 (`StrCpy(str1,StrCpy(str2,str))`)。

```
char* StrCpy(char*, const char*);
```

定义部分看上去好像很难，但我们分析了一通之后会发现，那也只不过是一种逐个赋值的语法而已，所以用 `for` 循环简单写一写就足够了。

```
char* StrCpy(char *dest, const char *src) {
    for (unsigned i = 0; (dest[i] = src[i]) != '\0'; i++) { ; } // 还能这样？
    return dest; // 返回 dest 的值
}

int main() { // 主函数
    char str[10] {"AbCdEfG"}; // 定义 str，并预先用其它内容占据
    cout << StrCpy(str, "Bob"); // 如果输出 Bob，说明代码正确
    return 0;
}
```

读者可以运行一下这段代码。如果不出意外的话，结果会输出 `Bob`。

这里的 `StrCpy` 的逻辑写得比较有技巧性，我们来拆解一下。在 `for` 循环中，我们先初始化了 `i=0`，这是循环的起始条件。接下来，我们进行判断。判断语句 `(dest[i]==src[i])!='\0'` 将会先执行 `dest[i]=src[i]` 这个赋值，其返回值 `dest[i]` 会与 '`\0`' 比较。如果不相等，那就说明没有遇到结束符，循环继续。主体部分只有一个分号（甚至可以什么也没有），它表示什么也不做。既然什么也不做，那就直接运行迭代语句 `i++`，进入下一轮循环判断就好。如此，我们也可以省了单独为 `dest` 数组添加一个结束符的麻烦——因为在判断它是不是 '`\0`' 之前，我们就已经做了赋值操作了。

<sup>35</sup>实际上，它也没办法顾及 `dest` 的容量够不够，因为字符串作为数组类型，在传递参数时就会变成指针类型，在此过程中它的长度信息就已经丧失了，除非我们人为地把长度信息也传入其中。

## 字符串的输入

现在我们再来讲讲字符串是如何输入的。

最简单的输入方式莫过于用 `cin <<` 来进行输入。

```
char str[33];
cin >> str; //直接用cin向字符串中输入
```

但是这样就可能会遇到问题。如果我们要向字符串中输入两个英文名，比如 Bjarne Stroustrup 和 Alan Mathison Turing，我们可能就会遭遇这样的困难：

```
char name1[33], name2[33]; //定义两个字符串用于输入人名
cin >> name1 >> name2; //连续输入两个人名
cout << name1 << endl << name2; //输出第一个人名，再换行，再输出第二个人名
```

如果我们真的运行一下，就会发现情况好像不太对劲。

### Bjarne Stroustrup

(刚按下回车键，准备输入第二个人名，结果程序就开始输出了)

Bjarne  
Stroustrup

这是怎么回事？

这是因为，用 `cin <<` 输入字符串时，它断句的方式是在每个空白字符<sup>36</sup>处断开。所以在输入 `name1` 时，它只会读取到空格之前的内容，而把 Stroustrup 这个输出留给下一个输入，即 `name2`。这也就解释了为什么当我们按下回车键的时候它就开始输出了，这是因为程序已经把两个字符串读完，接下来当然就是执行输出了。同时这也是输出 `name1` 时得到 Bjarne 而输出 `name2` 时得到 Stroustrup 的原因。

怎么解决呢？我们可以用 `cin` 的成员函数 `get` 或者是 `istream` 类的成员函数 `getline` 来解决。先来讲讲 `getline`。

### getline 成员函数

`getline` 成员函数的定义语法有如下两种，它们是重载关系：

```
istream& getline(char_type *s, streamsize count); //重载1
istream& getline(char_type *s, streamsize count, char_type delim); //重载2
```

其中的 `s` 是待输入的字符串，而 `count` 表明这个字符串的容量（不同于 `strcpy`，这里必须提供容量参数）。对于第一个重载来说，它的含义就是：读取一整行输入，直到遇到回车符<sup>37</sup>，才停止输入；或者是 `count` 长度的字符串容纳不了用户输入的内容量，这时输入也会停止，同时 `cin` 会进入“fail”状态（我们需要用 `input_clear` 来恢复 `cin`，并清理输入流）。

接下来我们可以用 `getline` 来输入一些有空格的名字了。程序不会在空格字符处断句，只会在回车字符处断句。

```
constexpr unsigned NameLen {33}; //把名称字符串容量设为33，便于统一
char name1[NameLen], name2[NameLen]; //定义两个字符串用于输入人名
cin.getline(name1, NameLen).getline(name2, NameLen); //getline可以连续使用
cout << name1 << endl << name2; //输出第一个人名，再换行，再输出第二个人名
```

<sup>36</sup>空白字符（Whitespace character），在 ASCII 字符区内包括 '\t', '\n', '\v', '\f', '\r', '\u'。如果要判断某个字符是否是空白字符，可以用 `cctype` 库中的 `isspace` 函数来处理，如 `isspace('a')`，返回值是 0。

<sup>37</sup>回车符会被提取，但不会存入 `s` 字符串中。

本程序的运行结果是

---

```
Bjarne Stroustrup
Alan Mathison Turing
Bjarne Stroustrup
Alan Mathison Turing
```

---

这就合乎我们的预期了。

至于第二种重载，无非就是把分隔符从换行符换成了自定义的字符而已，没什么特别的，这里就不再赘述。

### get 成员函数

`istream` 类的 `get` 成员函数的功能比 `getline` 更丰富。以下是 `get` 函数的一部分重载：

```
int_type get(); // (1)
istream& get(char_type &ch); // (2)
istream& get(char_type *s, streamsize count); // (3)
istream& get(char_type *s, streamsize count, char_type delim); // (4)
```

重载 (1) 和 (2) 的作用都是读取输入流中的单个字符。我们在 `input_clear` 中就用过 (1)。

```
void input_clear(istream &in) { // 默认参数 cin, 已在定义中注明
    in.clear();
    while (in.get() != '\n') // get() 的作用读取下一个字符
        continue;
}
```

`get()` 的返回值是读取的字符所对应的 ASCII 值，我们可以拿它与 '`\n`' 作比较。如果不相等，就说明还没有读取到回车符，循环继续；一旦读取到回车符，`while` 循环即终止，函数结束。

重载 (3) 和 (4) 与 `getline` 十分相似，但它们有着细微的区别，在于如何处理结束符。`getline` 会把结束符一并提取（形象点说，吞掉）；而 `get` 不会把结束符一并提取，结束符仍然保留在输入流中，下一个输入还会从这个字符开始。所以这样写就不行：

```
cin.get(name1, NameLen).get(name2, NameLen); // name1 正常，但 name2 为空
```

这是因为，`name1` 输入完成后，`get` 函数遇到回车符但没有吞掉它，回车符仍在输入流中，于是输入 `name2` 的时候，程序第一眼看到 '`\n`' 就把输入给结束了，`name2` 当然就会什么也没有了。

为了把输入流中的回车符吞掉，我们可以这样写：

```
cin.get(name1, NameLen).get(name2[0]).get(name2, NameLen);
// 先用 name2[0] 吞一下回车符，再输入 name2，没影响
```

或者干脆点，直接用 `getline` 就好。

## 5.6 指针与数组的复合类型

在前面的几节中我们可以看到，指针和数组之间有着千丝万缕的联系，它们虽然类型不同，但都与内存地址密不可分。数组类型还可以转换成指针类型，这种情况在参数传递中普遍存在。

本节我们来研究一些更复杂的结构：二维数组、指针数组、指向数组的指针和指向指针的指针（二阶指针）。它们的分析方法和我们分析一维数组和一级指针的方式大体相同，不过在这里我们要更多地关注类型信息。

## 二维数组

定义二维数组的基本语法如下：

```
int darr1[2][3] = {{1,2,3},{4,5,6}}; //正常格式，注意是外层2个，内层3个，勿搞反
int darr2[3][3] {{7,8},{10}}; //每个花括号内可分别省略内容，省略的部分初始化为0
int darr3[] [4] {{1,2,3,4},{5,6},{}}; //只能省略第一维长度，不能省略第二维长度
```

初学者很容易在二维数组的下标问题上犯糊涂。比如在定义 `darr1` 的时候，究竟是写成  `{{1,2,3},{4,5,6}}` 还是写成  `{{1,2},{3,4},{5,6}}`？还有，`int[2][3]` 和 `int[3][2]` 到底有什么区别？为什么在定义 `darr3` 的时候只能省略第一维的长度而不能省略第二维的长度？

要解决这些问题，我们就要去探寻二维数组的本质。

### 二维数组是什么？

我们已经对一维数组十分熟悉了。在我们已有的理解中，一维数组是内存中连续的一串数据，每个元素的类型都是基本数据类型。比如说 `int arr[3]`，它就是一个一维数组，类型为 `int[3]`；而它的某个元素，比如 `arr[0]`，就是一个数据，类型为 `int`。我们可以把它叫做“由数据构成的数组”。

那么我能否定义一个“由数组构成的数组”呢？假设我们想定义一个由 4 个 `arr[3]` 数组构成的数组，那么我们岂不是应该定义成 `int (darr[3])[4]`？

这就错了——其实这也是初学者常常会出现的理解误区。为了便于我们清晰地理解指针/数组复合类型的定义，请读者记住一条重要原则：“**定义的语法**要与**使用的语法**保持一致。

以刚才说到的二维数组为例。我们希望定义一个长度为 4 的二维数组 `darr`，这个数组的每个元素都是一个长度为 3 的数组。那么在我们使用的时候，`darr[0]` 就是一个长度为 3 的数组吧！同样的，`darr[1]`, `darr[2]` 和 `darr[3]` 都是一个长度为 3 的数组。

那么，如果我们还要再读取更内层的信息（这次就是数据了），我们就需要把 `darr[i]` 当作一个整体，再用一次下标运算符，也就是 `(darr[i])[0]`, `(darr[i])[1]` 和 `darr[i][2]`（可以省略括号，不影响结果）。

那么回顾一下 `darr[i][j]` 这个语法，其中 `i` 的合理范围是 0~3，而 `j` 的合理范围是 0~2，所以该定义成什么样，就一目了然了吧？

```
int darr[4][3]; //从darr到darr[4]，再到darr[4][3]
```

接下来我们可以用 `typeid` 或者 `is_same` 来检验它的类型。在 Coliru 中，我们可以运行以下代码：

```
int darr[4][3]; //从darr到darr[4]，再到darr[4][3]
cout << typeid(darr).name() << endl; //输出darr的类型
cout << typeid(darr[3]).name() << endl; //输出darr[3]的类型
cout << typeid(darr[3][2]).name(); //输出darr[3][2]的类型
```

这段代码的运行结果是

---

```
A4_A3_i
A3_i
i
```

---

我们从 `typeid` 的结果中也可以看出，`darr` 的输出 `A4_A3_i` 说明它是一个由“`int` 数据构成的 3 长度数组”构成的 4 长度数组。这个话有点绕，读者多琢磨一下，想必就能搞通啦。

所以为什么我们在定义二维数组时只能省略第一维的长度而不能省略第二维的长度呢？这点想必也是一目了然的，因为一个 `int[4][3]` 二维数组本质上是一个由 `int[3]` 一维数组构成的数组！

所以我们在定义数组类型的时候，必须把类型 `int`[3] 告诉编译器，否则编译器就不知道这个数组的类型了。但 4 作为这个数组的长度，是可以省略的。

```
int darr4[3][3]={{} ,{} ,{} ,{} } ; // 四个括号说明它由4个数组构成，所以4可以省略
```

## 二维数组在内存中是什么样的？

我们说过，一维数组就是在内存中连续存储的一些数据。既然二维数组本质上就是一维数组构成的一维数组，那么我们也不难想见，二维数组也是内存中连续的一段，如图 5.14 所示。

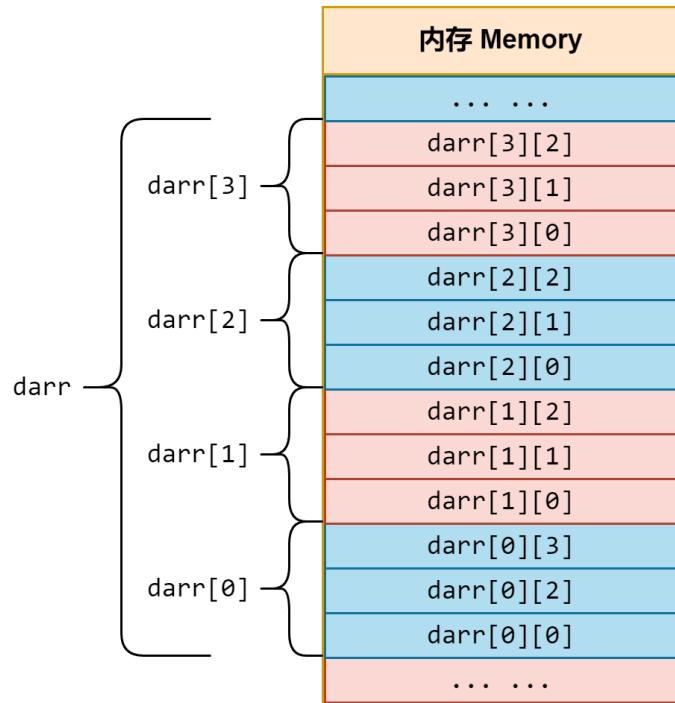


图 5.14: 二维数组在内存中的布局  
本图的每个蓝/粉色框代表一个数据，而非一个字节

一些资料会在讲解二维数组时把它画成矩阵的形状，这可能会给读者造成一种误解，即二维数组在内存中是以行列的方式排布的。这是一种偏见。实际情况是，数据在内存中的排布是线性的，比如 `darr[0][3]` 的下一位是 `darr[1][0]`，而不是 `darr[1][3]` 或者别的什么。

敏锐的读者可能会想，既然 `darr[0][3]` 的下一位就是 `darr[1][0]`，那么我们能否通过 `darr[0][4]` 这样的语法来访问 `darr[1][0]` 呢？这当然是可以的；但是如果我们只用 `darr[0][i]` 来访问数组数据的话，为什么我们要定义成二维数组呢？直接用一维的不是更方便吗？

## 二维数组的常见应用

既然二维数组数据在内存中的排布方式与一维数组一样，那么我们之所以会使用二维数组，肯定是因为二维数组在语法上有过人之处。这里我就来介绍一下它的几个常见应用：

- 矩阵。设想我们需要存储一个  $5 \times 5$  大小的矩阵。如果用一维数组的话，我们需要用一个大小为 25 的矩阵；在表示“第  $n$  行，第  $m$  列”数据的时候，我们需要写成 `arr[(n-1)*5+(m-n)]`（别忘记数组下标从 0 开始）。但是我们可以用一个大小为  $5 \times 5$  的二维数组直接存储；表示时直接用 `darr[n-1][m-1]` 就行了。这样就很方便吧！
- 多个字符串。字符串本身就是一个 `char[N]` 类型的数组。那么当我们需要批量处理字符串时，当然就要用“数组的数组”啦！

- 数据统计。有些时候我们需要记录两组数据并作统计分析，例如对“售价”-“利润”数据进行存储。我们当然可以使用两个一维数组来实现这个功能，但是用一个二维数组会在传参时更方便一点！我们可以直接传递一个二维数组，而不是传递两个一维数组。<sup>38</sup>

这里提及到了二维数组参数传递的问题，我们留到待会儿讲指向数组的指针时再讲解。

## 指针数组

二维数组的表象看上去好像很复杂，其实经过解析之后我们就会发现，那只不过是一种“以数组作为构成元素”的一维数组罢了。指针数组也是这个道理，它是一种“以指针作为构成元素”的一维数组！

指针也是一种数据类型，它一经定义，就在内存中有自己的一席之地。假如说有一个场合，需要我们批量定义指针，那么**指针数组（Array of pointers）**就派上用场了。

定义指针数组的基本语法如下：

```
const char *ap1[3] = {"Alice", "Bob", nullptr}; // 正常格式
int *ap2[10] {}; // 省略的部分将初始化为nullptr
float *ap3[] {nullptr, nullptr, nullptr}; // 可以省略指针数组的长度
```

初学者很容易把指针数组的定义语法与下文要讲的指向数组的指针搞混，我们在这里还是按照刚才说的原则来解释：“定义的语法”要与“使用的语法”保持一致。

## 指针数组是什么？

这里要提问读者：`*ap1[0]` 是什么？根据 C++ 运算符优先级的知识，我们知道 `[]` 的优先级高于 `*`，所以它应该被理解为 `*(ap1[0])`。那么 `ap1[0]` 是什么？按照我们的理解，它应该是指针数组的第一个元素——一个指针。那么 `*ap1[0]` 就相当于再对这个指针取内容。而 `ap1[0]` 指向的是字符串面量 `"Alice"`，那么我们不难想见，`*ap1[0]`，或者等效于 `(ap1[0])[0]`，就是字符 `'A'`。

我们同样可以用 `typeid` 来检验它们的类型。

```
cout << typeid(ap1).name() << endl; // 输出 ap1 的类型
cout << typeid(ap1[0]).name() << endl; // 输出 ap1[0] 的类型
cout << typeid(*ap1[0]).name() << endl; // 输出 *ap1[0] 的类型
```

这段代码的运行结果是

---

A3\_PKc

PKc

C

---

其中的 `PKc` 意为“指向常量 `char` 的指针”，总之是一个指针就对了；而 `A3_PKc` 当然就是一个指针数组<sup>39</sup>。

## 指针数组在内存中是什么样的？

指针数组在内存中的结构非常简单，它就是一个普通的一维数组而已，只是元素类型换成了指针。指针类型的数据在内存中也有其存储空间，我们之后就会来研究相关问题。

<sup>38</sup>需要注意一点，用二维数组的前提是这个二维数组中的所有数据都是同一类型。如果涉及到“名字”-“性别”-“年龄”这样不同数据类型同时存在的情况，我们就需要用结构体，详见第六章。

<sup>39</sup>其实我们可以分得更细，它是一个“指向常量的指针数组”（*Array of pointer to constant*），不过初学阶段的读者就不必深究了。

## 指针数组的常见应用

我们在前面已经看到了指针数组的一个常见应用，那就是存储字符串字面量。我们此前已经提过，字符串字面量不同于其它类型的字面量，它是有地址的！既然它有自己的地址，那么我们没必要用一个二维数组去存储（既浪费存储空间，又浪费存储所用的时间），直接用一堆 `const char*` 来指向这些内容就行，什么时候需要就拿出来用。当然这样做的前提是不需要修改这些字符串字面量的值。如果修改这些值，就可能发生未知错误（这也就是我们使用 `const` 的原因）。

指针数组还可以用于动态内存管理。在涉及动态内存分配时，我们可以把一个二级指针通过动态内存分配变为指针数组，然后对其中的每个指针再进行动态内存分配，诸如此类。我们等到下一节再谈此话题。

在传递参数时，我们也可能用到指针数组。这和我们用数组类传递参数有着异曲同工之处——如果参数太多，那么用一个数组来传参就要经济得多；如果要传递的指针太多，那么用一个指针数组来存储这些指针，把它作为实参传递给函数，也要经济得多。我们留到待会儿讲二阶指针的时候再讲解。

## 指向数组的指针

**指向数组的指针** (**Pointer to array**, 简称**数组指针**)，对初学者来说更是一个难题。很多初学者搞不清楚什么是指向数组的指针，什么是指针数组，并且经常在定义的时候搞混。这里我们不仅要讲清楚指向数组的指针是什么，还要理清它和指针数组之间是什么关系。

定义一个指向数组的指针的基本语法如下：

```
int (*pta1)[5] = {nullptr}; // 正常格式，等号可省略
double arr[3] {};
double (*pta2)[3] {&arr}; // 3不可省略，并且需要和arr的长度保持一致！
```

不要小看定义语句中的 `()`，它是指向数组的指针区别于指针数组的关键。

### 指向数组的指针是什么？

顾名思义，这个指针是指向数组的。我们之前遇到的那些指针，它们都是指向数据的。现在我们遇到了指向数组的指针，不用怕，我们把数组也当成一种数据类型就好。（还记得吗，如果我们把一维数组当成一个数据类型的话，二维数组其实就是“一维数组构成的一维数组”）

如图 5.15 所示，现在有一个二维数组 `int darr[4][3]`。它的每个元素都是一个长度为 3 的 `int` 数组 (`int[3]`)，那么我们可以定义一个指向 `darr[0]` 的指针，如下所示：

```
int (*pta)[3] {darr[0]+0};
```

在这里我们要解决两个问题。第一，怎么理解它的定义语法？第二，二维数组和指向数组的指针究竟有什么关系？为什么我们可以用 `pta` 指向 `darr[0]+0`？

首先，让我们来考虑一下使用的语法。`(*pta)[0]` 是什么？这里 `(*pta)` 套了括号，所以应该先算它。鉴于 `pta` 是一个指向数组的指针，那么 `(*pta)` 就是一个数组了。所以 `(*pta)[0]` 就是这个数组的第一个元素，就这么简单。现在读者可以再回看一下指针数组的定义，并比较它们“使用的语法”，大概就可以分清它们的定义语法了吧！

然后再来看看二维数组和指向数组的指针间的关系。我们之前说过，`T` 类型数据构成的数组 `T[N]` 也可隐式转换成指向 `T` 类型的指针，即 `T*`。对于二维数组也是如此！既然一个二维数组就是一维数组构成的一维数组，那么它就可以转换成指向一维数组的指针！

那么读者也可以猜一下，`pta+1` 会指向什么呢？既然指针的加减法讲究的是数据的偏移量而非地址的偏移量，那么我们就应该想到，`pta+1` 应该指向下一个数组，也就是 `darr[1]`。同理，`pta+2` 应该指向下一个数组，即 `darr[2]`，以此类推。图 5.15 给出了这种关系，读者可以作为参考。

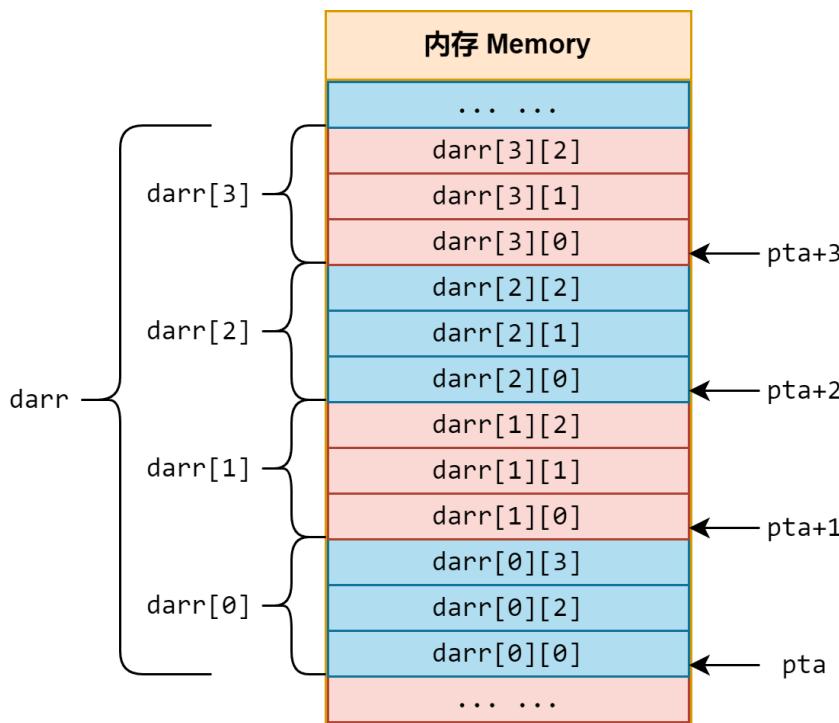


图 5.15: 指向数组的指针 `pta` 指向了数组 `darr[0]`

### 指向数组的指针在内存中是什么样的？

指向数组的指针只是一个指针。如果你用 `sizeof` 取它的内存占用，会发现它的大小和指向 `int` 的指针相等。实际上，无论指针指向什么，它永远都只是负责存储那个事物的地址而已，所以寥寥几个字节的空间足够！

这也就是我们喜欢用指针或引用<sup>40</sup>传递参数的原因——无论要传递的信息有多大，一个指针那么大的数据就可以说明一切了！

### 指向数组的指针的常见应用

要说数组指针最常见的应用，那当属二维数组参数传递了。

我们说过，所有的数组参数传递在本质上都是指针传递。对于二维数组也是这样。所有的二维数组在进行参数传递时都会被隐式类型转换成指向一维数组的指针，然后传递参数。

```
double MatrixSum(double matrix[][n], int rows);
```

这个函数就可以用来求一个矩阵中所有数字的和。

一个数组也有它的内存地址。比如说有一个一维数组 `arr`，我们可以输出它的地址，也可以直接把它当作指针来输出。

```
int arr[3];
cout << arr; //arr会转换成int*型然后输出
cout << &arr; //直接输出一个int(*)[3]，这就是指向数组的指针类型
```

它的输出将会是这样的：

---

0x7ffd12a60744

<sup>40</sup>C++ 中的引用是通过指针实现的，只是它在代码上隐藏起来了。我们进行引用传递的过程其实都是在进行指针传递。所以你可以认为引用是一种语法糖。

---

0x7ffd12a60744

读者可能会感到奇怪：这两个地址一样啊！

没错，这两个地址的值是一样的，但它们的类型完全不同。对于 `arr` 来说，它是 `int[3]` 类型，而在输出的时候会转换成 `*` 类型。而对数组取地址将会得到 `int(*)[3]`，这是一个“指向 3 长度 `int` 数组的指针”类型。我们可以用 `typeid` 或者 `is_same` 来验证它。

```
cout << is_same<decltype(arr), int[3]>::value << endl; // 输出 1
cout << is_same<decltype(&arr), int(*)[3]>::value << endl; // 输出 1
```

还记得我曾经说过吗？对于一个需要用多个字节来存储的类型来说，它的地址就是存储区域中第一个字节的地址。`arr` 的第一个字节也正是 `arr[0]` 的第一个字节，地址为 `0x7ffd12a60744`，所以自然会输出相同的值，读者不必感到奇怪。如果输出了不一样的值，我们才要感到不对劲呢。

那么我们是不是还能想到一种输出字符串字面量地址的新思路？

```
cout << &"HelloWorld!"; // 将输出地址值
```

这里的 `&"HelloWorld!"` 也是一个指向数组的指针类型，具体地说，是 `const char(*)[13]`。

```
cout << is_same<decltype(&"HelloWorld!"), const char(*)[13]>::value;
// 输出为 1
```

## 二阶指针

我们在前文中说过，一个指针也有它的内存地址。那么我们可以通过取地址运算符来取出它的地址吧。

```
int num {3};
int *p {&num};
cout << &p << endl << &num;
```

这个程序的运行结果是

---

0x7ffd7e034998  
0x7ffd7e034994

---

不同于我们在前面看到的例子，这里的 `&p` 和 `&num` 是不同的。

它们当然不同啦，又不是同一个数据，也没有包含关系，当然应该存储在不同的单元中。

那么我们能否用另一个指针来指向 `p` 呢？这是可以的。我们可以定义一个二阶指针，或者说，指向指针的指针，来实现这个功能。

定义一个二阶指针的基本语法如下：

```
int **pp1 = {&p}; // 正常格式，等号可省略
char **pp2 {}; // 省略将默认初始化为 nullptr
```

这个语句的定义就很容易理解了，不会涉及什么误解。

## 二阶指针是什么？

初学者容易在自己尝试的时候写出 `&&num` 这样的语法，试图一步到位，得到一个二阶指针。这样写是错误的，而且也是没有意义的。这是因为，二阶指针描述的是一个指针的地址。但是 `&num` 只是一个临时的返回值<sup>41</sup>，不是内存中我们可以获取到的实体，所以对它取地址是没有意义的！

<sup>41</sup>更确切地说，右值。我们会在精讲篇中探讨这个问题。

但是我们可以对二阶指针两次取内容，这是因为，只要它是一个地址，它就可以取内容。二阶指针存储了某个一阶指针的地址值，而一阶指针又存储了某个数据的地址值，所以我们可以对二阶指针连续取地址。

```
cout << typeid(pp1).name() << endl;
cout << typeid(*pp1).name() << endl;
cout << typeid(**pp1).name() << endl;
```

这段代码的运行结果是

```
PPi
Pi
i
```

很好理解，我就不多说了。

### 二阶指针在内存中是什么样的？

同上文，它是一个单独的变量，在内存中有它自己的一席之地。我们甚至还可以取二阶指针的地址，并把它存入三阶指针当中！

```
int ***ppp {&pp1}; //这是一个三阶指针，存储了一个二阶指针的地址
```

还有更高阶的指针，不过我们在这里就不多说了。

### 二阶指针的常见应用

我们知道，一个二维数组可以隐式转换成指向数组的指针。而一个指针数组也可以转换成一个二阶指针。其实这点也符合我们的规律，因为它是一个“由指针构成的数组”向“指向指针的指针”的转换。所以如果我们要向某函数传递一个指针数组，我们就需要用二级指针形参来接收了。

这四种类型的转换关系可以用图 5.16 来描述。注意，二维数组是不能直接转换成二级指针类型的，数组指针也是不能直接转换成指向数组的指针类型的。这方面更详细的讨论，我们放在精讲篇。

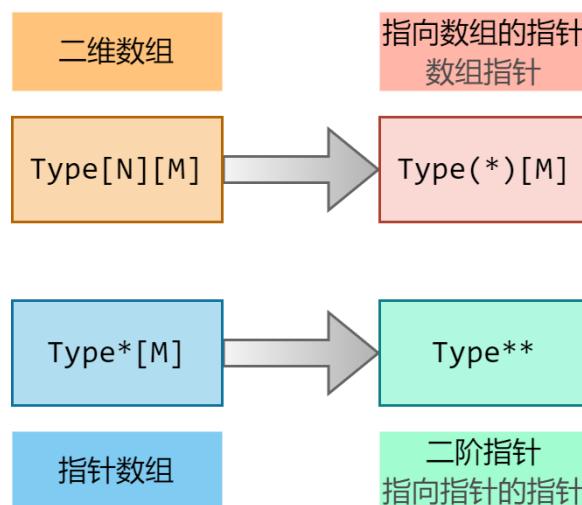


图 5.16: 四种指针/数组复合类型的类型转换关系

## 5.7 动态内存分配

在本章前几节的例子中，我们每次使用一个数组时，都会把它定义成一个长度为常数的类型。但在实际使用的时候，我们可能不知道用户需要多大规模的数组。如果预先定义一个十分大的数组，就免不了会浪费大量的内存空间，这在实际使用时是很不经济的。

我们能否定义一个“在运行时确定长度”的数组呢？早期的编译器不允许这种语法，不过许多较现代的编译器支持这种语法。

```
unsigned n; // 长度变量
cin >> n; // 由用户输入想要的长度
char str1[n]; // 这就是一个n长度的数组，n是刚才用户输入的值
cin >> n; // 由用户再次输入一个长度变量
char str2[n]; // 这就是一个n长度的数组，这里n的值可以不同于上次
```

但是我不推荐这样定义数组，有以下几个原因：

- C++ 标准从未对变长度数组<sup>42</sup>的使用作出明确规定规定。C++ 标准中只允许了使用常量表达式<sup>43</sup>来作为数组长度。因此它本身就是一种未定义行为。
- 这种语法只是部分编译器对 C++ 标准的扩充，但并不是所有编译器都能做到！比如说，这段代码可以在 GCC 或 Clang 中通过编译，但在 MSVC 中就不行。
- 变长度数组的类型是模糊的。我们提过，不同长度的数组属于不同的类型，它们之间不能等同。那么这里的 str1 和 str2 到底是什么类型呢？这就造成了类型上的困惑。

正因如此，我非常不推荐使用这种方法来定义“在运行时确定长度”的数组。我们权当这种语法不存在就好。

那么我们还有什么别的办法吗？这就要谈到动态内存分配（Dynamic memory allocation）技术了。

### 动态内存分配是什么？

我们此前所接触的数组定义方式被称为“静态分配”，简单说就是，一个数组的大小在编译期就决定好了。具体的内存分配肯定是在运行时进行，但是它也只能遵照代码中的规则，代码中要求它分配多少内存，它就分配多少内存。

这种方式很简单，也容易实现。但它的问题在于太不灵活。为了使数组有足够的容量，我们不得不用饱和式的方法来分配内存，把数组设计得越大越好。

动态内存分配则不同，它可以根据运行时的实际需要来分配内存空间。如果运行时需要存储 100 个 **double** 数据，那我们就使用一个大小为 800 字节 ( $100 * \text{sizeof}(\text{double})$ ) 的内存块来存着就好。如果运行时需要存储 10000 个呢？那我们就使用一个大小为 80000 字节的内存块来存着就好。这就是“动态分配”，我们可以在运行时再决定使用多少内存，而无需提前在代码中决定。这是一种按需分配。

动态内存分配需要使用关键字 **new**，它会在内存的堆段找到一段未被使用的空间。我们需要用一个指针来存储这段空间（第一个字节）的地址，否则我们就不知道它在哪里了（这也是指针的重要之处）。

<sup>42</sup>变长度数组（Variable-length array），是指大小在运行时确定的数组。这里特指如 **char** str1[n] 这样用变量长度来定义数组的情况。

<sup>43</sup>或者是在编译时可确定的常量。总之，需要是一种能够在编译时确定的信息。

```
int *pobj = new int {3}; //new在动态内存中创建一个对象（数据）并初始化为3
double *arr = new double[4] {.1, .2};
//new[]在动态内存中创建一个长度为4的数组，初始化列表中留空的部分为0
```

`new` 和 `new[]` 是两个运算符，前一个用来动态分配单个对象（数据），而后一个用来动态分配数组。

在实际编程中，这两个运算符的作用差不多，而且一般也不会同时出现，所以我们通常把它当成是一样的东西。但是在回收的时候我们要注意它们的对应关系，否则就会出错。我们待会儿再谈。

`new/new[]` 的返回值是一个 `void*`，它会隐式类型转换成我们需要的指针类型。比如说 `new int` 的返回类型就是 `int*`，而 `new double[4]` 的返回类型就是 `double*`。这个返回值可以被指针存储，此时这个指针就指向了此段动态内存。

对于 `new` 分配的内存空间，我们可以把它当成一个“指向对象的指针”来用，比如说 `*pobj`（当然，也可以是 `pobj[0]`），这就相当于一个对象（变量），它的用法如同对象名（变量名）那样。

对于 `new[]` 分配的内存空间，我们可以把它当成一个“数组”来用，比如说 `arr[i]`（也可以是 `*(arr+i)`）。总之，只要你会用静态的变量/数组，那你自然就会用动态的。

## 内存泄漏问题及动态内存的回收

动态内存与静态内存的另一个区别在于它们的生存期。我们讲过静态数据的生存期，它们在定义完毕时生存期开始，而在作用域结束时生存期结束。

```
{ //某个作用域
    int x {3}; //x的生存期开始
    { //内层作用域
        int y {x}; //y的生存期开始；x对本作用域可见
    } //y的生存期结束，此后y这个名字没有意义
    { //另一个内层作用域
        int y[5]; //y的生存期开始，虽然它和前面的y重名，但它们没有任何必然联系
    } //y的生存期结束
} //x的生存期结束
```

这些局部静态变量存储于栈段（见图 5.1）。

而经过 `new/new[]` 分配的内存空间存储于堆段（见图 5.1），它们的特征与静态变量不同。它们在 `new/new[]` 之后生存期开始，此后就会长期存在，无论作用域怎么变化，都不会影响它们的生存期。

```
{ //某个作用域
    int *p; //p的生存期开始，p仍是静态生存期的
    { //内层作用域
        p = new int[3]; //p指向的位置处，有一个动态数组的生存期开始
    } //作用域结束，但动态数组的生存期并未结束
    p[0] = 3; //动态数组仍然可用
} //p的生存期结束，但动态数组的生存期依旧没有结束
```

现在我们遇到了一个棘手的问题：`p` 的生存期结束了，但用 `new[]` 分配的动态数组的生存期还没结束。结果就是，这个内存空间已经不能用了（因为 `p` 已经不能用了，我们就不可能知道这段动态内存的地址的），但是它仍然存在，变成了占据着内存空间但起不到任何作用的僵尸！这就是一个典型的内存泄漏（Memory leak）问题。

在个人电脑上，内存泄漏问题是很容易解决的。比如这段代码

```
int main() {
```

```

while (true)
    new long double;
return 0;
}

```

这是一个内存泄漏器，只要跑起来就会不停地吃内存，直到内存空间没有一点余量，程序才会以 `bad_alloc` 异常结束。读者可以打开自己的任务管理器，看一下内存占用的情况。

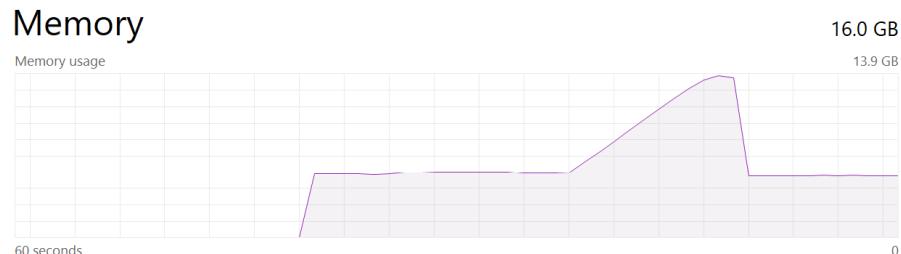


图 5.17: 开始运行内存泄漏器和结束运行时，内存占用的变化

图中可以看到，在我们开始运行这个程序的那一刻，内存占用就迅速增加，直到接近最大值。而在我们结束这个程序的时候，内存占用马上又回到了原来的水平。所以对于个人电脑来说，内存泄漏是无关紧要的小问题，结束一下进程，它就解决了。

但是不是任何时候都可以靠结束进程来解决问题的。对于需长期运行的服务器来说，贸然结束进程可以意味着很大的损失。所以在我们日常写代码的时候就要养成良好的习惯，不要写出会导致内存泄漏的代码来。

那么怎么预防内存泄漏呢？唯一的方法就是及时回收（或者叫释放）那些我们不需要的内存空间。

回收内存空间需要用到 `delete` 或 `delete[]` 运算符。其中 `delete` 运算符用于回收 `new` 分配的内存空间，而 `delete[]` 运算符用于回收 `new[]` 分配的内存空间。它们两个是不同的，一定不要混用！

```

int *pobj = new int;
delete pobj; // 使用 new 分配的动态内存要用 delete 回收
pobj = nullptr;
double *arr = new double[4];
delete[] arr; // 使用 new[] 分配的动态数组要用 delete[] 回收，[] 内无需加数字
arr = nullptr;

```

读者需要注意的是，`delete/delete[]` 只是回收了这个指针所指向的动态内存，它不会影响指针的值。那么在动态内存回收之后，这个指针指向的就是无效区域——换句话说，它是野指针！那么为了防范野指针的出现，我们最好在回收内存空间之后，让这个指针指向 `nullptr` (`nullptr` 是对所有指针都绝对安全的避风港)。

真实情况下的内存泄漏往往不是由这么低级的“忘记回收”导致的，而是一些“我们自己都没想到需要回收”的情况，比如

```

int* alloc(int value) { // 注意返回类型是 int*, 它意味着返回一个地址值
    return new int {value}; // 直接返回由 new 分配好的动态空间，已经用 value 初始化
}
int main() {
    const int N {10};
    int *p[N]{};
    // 定义由 N 个指针构成的指针数组
}

```

```

p[0] = alloc(5); //让将 p[0] 指向 alloc 分配好的动态内存空间
//在经历了很多操作之后
delete p[0]; //最后记得回收！
return 0;
}

```

我们很容易在复杂的代码中忘记了回收动态内存空间，尤其是在这种“`new` 出现在一个函数中，而使用这个动态内存却是在另一个函数中”的情形下（类似的例子，我们会在面向对象编程时见得很多）。所以动态内存分配的灵活性与危险性是并存的，我们稍有不慎，就可能会造成内存泄漏的问题。

## 二维数组的动态内存分配

现在你已经学会了如何动态分配一个一维数组，那么如何动态分配一个二维数组呢？我们想，二维数组也只不过是一个“一维数组类型的一维数组”，所以我们用一维数组的动态内存分配语法就可以分配一个二维数组了。

```

int n;
cin >> n;
int (*pta)[10] = new int[n][10];

```

在这里，`new[]` 的返回值是一个 `(int*)[10]`，也就是指向数组的指针，我们可以定义一个 `pta` 来存储这个返回值。这样，我们就用一个指向 `10` 长度 `int` 数组的指针，指向了一个 `n` 长度的 `10` 长度 `int` 动态数组。这话可能有点绕，读者多琢磨一下想必就可以理解。

回收这个数组的方式很简单，直接 `delete[]` 就行。

```

delete[] pta; //回收 pta 指向的动态内存
pta = nullptr; //最好这样

```

但是这样也有一个限制——我们在这里定义的是一个 `n*10` 大小的数组，还是有一个维度需要静态分配的。我们能不能让两个维度都是变量，比如定义一个 `n*m` 动态数组？其实是可以的。

我们的思路是，先用一个二阶指针，分配一个动态指针数组：

```

int **pp = new int*[n] {}; //全部初始化为nullptr

```

在这里，`new` 的返回值是一个 `int*[n]`，也就是长度为 `n` 的指针数组。那么其中随便一个 `pp[i]` 都是一个 `int*` 指针，工我们岂不是可以直接为每个这样的指针分配一个长度为 `m` 的动态数组？

```

for(int i=0; i<n; i++) { //别忘了 pp 下标的范围是 0~n-1，而不是 1~n
    pp[i] = new int[m] {}; //对其中的每个指针 pp[i] 都分配一个长度为 m 的数组
}

```

至于这种二维数组的回收，那就有讲究了。我们首先为 `pp` 分配了一个 `int*` 指针数组，然后又为其中的每个 `int*` 指针分配了一个 `int` 数组，所以这些内容都要回收（仅回收 `pp` 指向的空间不代表回收了每个 `pp[i]` 指向的空间）。但是怎么回收就是个问题了——

```

delete[] pp; //回收 pp 指向的空间
for(int i=0; i<n; i++) {
    delete[] pp[i]; //错误！
}

```

我们仔细想一遍过程就知道为什么了：当 `pp` 的空间被回收之后，`pp` 就变成了一个野指针。这时候再访问它的内容就不合理了，所以 `pp[i]` 这个用法也就有问题了！

所以我们回收动态内存的时候也要讲求一个顺序。应该先回收各个 `pp[i]` 的空间，然后再回收 `pp` 的空间才对。

```
for(int i=0; i<n; i++) {  
    delete[] pp[i]; //先回收各个pp[i]的空间  
}  
delete[] pp; //再回收pp的空间
```

# 第六章 自定义类型及其使用

迄今为止，我们遇到的各种类型都是 C++ 提供给我们的。要么是基本类型，要么是指针、数组、引用这样的复合类型。类型（Type）与数据（Data）之间是类与对象的关系。一个数据只不过是内存中若干 0 和 1 的排列组合，倘若没有类型，程序就不知道如何把这些信息解释成有效内容。而对于同一串 0/1 来说，如果类型不同，那么解释出来的信息也是不一样的。以下是一个例子：

```
int n {0x63794b37}; // 这是一个 16 进制字面量，其 10 进制为 1668893495
cout << (char*)&n; // 以字符串形式输出
```

在某个环境下<sup>1</sup>，它的输出是

---

7KycT¤\$¤¤

---

为什么把 `int*` 类型转换成 `char*` 类型之后再输出就会得到这么奇怪的内容呢？原因就出在类型上。

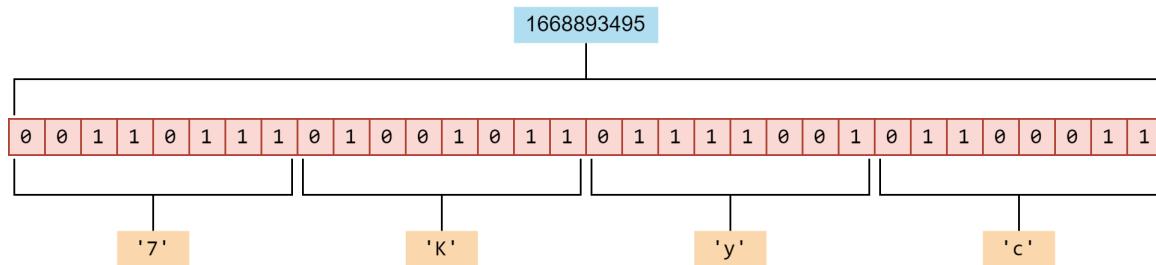


图 6.1: 对同一段比特串，不同类型会解读出不同内容

内存中的 32 个比特如图所示。如果用 `int` 类型去解读它，就会得到 1668893495；而如果用 `char` 类型去解读它，就会分别得到四个字符 '7', 'K', 'y', 'c'。而当我们以 `char*` 类型输出时，因为这里没有结束符 '\0'，所以它还会继续输出后面内存中的无意义内容，直到碰到结束符为止。

所以我们可以认为，类型就是信息与二进制编码之间的一套转换方案。同样一串二进制编码，在不同类型下会解释出不同的信息——这也就是为什么我从第一章起就在强调“类型”。

那么言归正传（刚才的内容看不懂也没关系）。在本章，我将讲解如何利用基本数据类型和上一章中讲到的复合数据类型，来自创类型。我们不需要研究到比特这个层次，更不需要讲什么编码，我们拿 C++ 现成的类型来组合就可以了。

## 6.1 枚举常量 `enum`

定义一个枚举常量的基本格式如下：<sup>2</sup>

---

<sup>1</sup>不同环境下给出的结果不尽相同

<sup>2</sup>本节只介绍无作用域枚举，关于有作用域枚举，请见精讲篇。

```
enum <枚举名> {<枚举项>=<整型常量表达式>, <枚举项>=<整型常量表达式>, ...};
```

枚举名不是必须的。如果不使用枚举名，那么这些枚举项的类型就是匿名枚举，这样做相当于定义了一些整型常量。如果我们规定了名字，那它们就是具名枚举，有丰富的作用。

枚举常量都是基于整型的，它的枚举项可以隐式类型转换为整型来输出。

```
enum {A=1, B=3, C=5}; //不具名枚举
cout << B; //输出3
```

我们可以省略其中的部分枚举项的值，这时枚举项的值是基于前一个值递增的。

```
enum {A=-2, B=4, C, D}; //C为递增为5, D递增为6
enum {E, F, G}; //第一项E不设值则为0, 后面F,G分别为1和2
```

如果我们需要一次性定义大量整型常量，那么除了 **const** 及 **constexpr** 语法之外，我们还可以选择 **enum**。

如果这个枚举有枚举名，那么我们还可以把它作为一个类型来用！具名枚举类型的优点是：

- 它的值可以用标识符来表示，而不是用单纯的字面量——我们提过，常量表达式比字面量的一大好处在于方便在源代码层面做统一修改。
- 具名枚举类型可以隐式转换为整型常量，所以我们使用起来很方便。
- 整型常量不能隐式转换为具名枚举类型，而且具名枚举类型数据的取值范围有很大限制，我们可以很防止这类数据取一些不允许的值。

举个例子，想必读者就可以理解了。关于服装的尺码，不同国家/地区都有各自的规定。以男士西装为例，欧洲/俄罗斯、英国/美国、日本、韩国都有自己的尺寸对照表。我们可以用枚举常量来表示各个尺寸。

```
enum SuitsSize {
    XXS=40, XS=42, S_1=44, S_2=46, M_1=48, M_2=50,
    L_1=52, L_2=54, XL=56, XXL=58, XXXL=60
}; //这是欧洲/俄罗斯标准
```

接下来我们可以定义一些 **SuitsSize** 类型的变量，比如说就定义成对应的人名吧。

```
SuitsSize Erik {XXL}; //Erik的值就是XXL, 不是58, 只是它可以隐式转换成58
SuitsSize Jacques {XS}; //Jacques的值就是XS
SuitsSize Carlos {M_1}; //M尺码有两个, 为了区分而分别写成M_1和M_2
```

当我们想要知道枚举项的具体值时，我们可以直接以 **int** 类型输出。

```
cout << static_cast<int>(XL); //或者直接cout<<XL, 会隐式类型转换
```

如果某个人的尺码发生了改变，我们可以直接赋值来改变，非常方便。

```
Erik = XXXXL; //胖了
```

但是我们赋值也不能乱来；我们只能赋那些已经规定好了的项。

```
Eirk = XXXXXXXL; //错误！
```

既然 **SuitsSize** 也是一个类型，那么我们当然还可以定义这个类型的常量数据。

```
const SuitsSize Giovanni {XXXL}; //Giovanni的值一经定义, 就不可改变
```

总而言之，我们可以把具名枚举的各项看作是整型数据，但是我们应该先把它当成一种特殊的自定义类型，它的值就是标识符本身，只不过可以转换为整型而已。作为类比，我们可以想一下 **bool** 类型，它的值就是标识符 **true** 或 **false** 本身，只是它可以隐式转换成整型而已。

每个枚举项的内存占用默认与 **int** 相同，这种做法有些时候有点浪费空间。比如说，**SuitsSize** 的所有枚举项的值都在 40 到 60 之间，用一个字节足够存储得下了。为了节约内存空间，我们可以改变它的枚举基，也就是它们基于哪种类型。

```
enum SuitsSize : char { //... }; //省略
```

这样，**SuitsSize** 类型的内存占用就变成了一个字节，读者可以用 **sizeof** 检验之。

除了 **char** 之外，我们还可以用 **bool** 以及所有的整型类型。例如要表示一个人的性别，我们可以用一个 **bool** 型数据来实现。我们可以记 **true** 表示男性，**false** 表示女性。不过用 **true/false** 的表示方法不够直接，我们何不自定义一个具名枚举呢？

```
enum Sex : bool {male, female}; //以bool为枚举基
```

在这里，我们当然也可以指定 **male/female** 的布尔值；但是这样做没太大的意义，因为我们真正要表示的信息其实是性别而不是那个布尔值。在应用的时候，我们可以直接这么用：

```
Sex group[5] {male, male, female, male, female}; //定义
for(auto individual : group) { //范围for循环
    if(individual == male) { //用individual==male 作为判断条件
        //...
    }
    else {
        //...
    }
}
```

## 6.2 结构体 **struct**

我们之前讲过的各数据类型，除了数组以外，都是只能表示单个数据的。比如说一个 **double** 数据，虽然它有 8 个字节，但是它的 64 个比特全都用来表示单个数据了。但是我们很容易想到，现实中的很多东西不是单纯用一个数据就能描述清楚的。比如我要描述一个长方体的信息，我需要三个数据：长、宽、高。这三个数据最好放在一起，作为一个整体存在，所以“定义三个变量”的思路就有点太原始了。

所以我们自然会想到定义数组。

```
int cuboid[3];
```

这样当然可以，但是它也有几个缺陷，不太方便解决：

- 数组类型没有排他性。它就是一个 **int[3]** 类型，但是有很多东西都是 **int[3]** 类型的，比如说三维空间坐标，或者是颜色的 RGB 值。如果说这些东西都算是同一类型的话，那未免有点牵强。
- 各个维度的数据没有明确的含义。**cuboid[0]**，这个数据到底是长度，还是宽度，还是高度？这会造成困惑，所以我们需要事先约定第几个数据代表什么。这样就增加了理解成本，也容易犯错。如果我们可以直接给每个数据命名呢，岂不美哉？<sup>3</sup>

<sup>3</sup>这点也可以通过枚举常量作下标的方式来强行实现，不过就请读者自行尝试吧。

- 数组对数据的包装不够彻底。它们归根到底还是三个数据，我们很难把它当成真正的“整体”来对待。比如说，函数的返回值只能是单个（整体）数据，但很明显函数不能直接返回一个数组。（我们可以让它返回一个指向数组的指针，或都是数组引用，但是那会非常麻烦）这就说明它的集成度还不够，我们需要集成度更高的方案。
- 数组只能存储同一类型的数据，这是一大硬伤。试想，如果我们描述一个人的特征，我们可能需要很多类型的数据放在一起，做成大杂烩：描述名字要用字符串类型，描述性别要用 **bool** 类型（或者以 **bool** 为枚举基的 **Sex**），描述身高体重要用 **float** 类型（如果对精度要求不高），描述年龄要用 **unsigned short** 类型（用 **int** 也行），这么多类型，肯定是无法放在一个数组里的。

结构体能很好地解决以上困难，它是一个有排他性，集成度更高，类型支持更丰富的解决方案。有了结构体之后，我们就可以自定义数据类型，并存储我们想要存储的信息了。

## 定义、声明、初始化和使用

结构体的定义不同于函数的定义。函数不能嵌套定义<sup>4</sup>，但我们在函数内或在函数外定义结构体，也可以嵌套定义结构体。

不过就我们的习惯而言，把结构体定义在函数外——也就是全局范围内的情况更普遍。

声明/定义一个结构体需要用到 **struct** 关键字。比如我们要定义一个长方体的信息，我们就需要把长、宽、高三条信息都包装到这个结构体中，所以我们可以这样写：

```
struct Cuboid { // 这个类型的名字就叫Cuboid了
    int length; //length部分数据，用int型
    int width; //width部分数据
    int height; //height部分数据
}; //注意末尾的分号！
```

注意，这里的 **length**, **width** 和 **height** 不是“枚举项”，它们不能单独存在，必须是作为 **Cuboid** 数据的一个部分存在的。我们也把这些部分称为成员（Member）。

如果我们要声明，直接写成这样就行：

```
struct Cuboid; // 声明Cuboid类型
```

这样之后我们就可以定义结构体的对象（数据）了。我们来看一下如何初始化它们。

```
Cuboid cub1 {1,2,3}; //length为1, width为2, height为3
```

也就是说，在花括号 {} 内的初始化数据会一一对应到 **Cuboid** 的三个成员中。那么如何使用它们呢？我们要用到成员访问运算符 **.**。

```
int volume1 {cub1.length * cub1.width * cub1.height}; //计算其体积
```

在这里，**cub1.length** 就是 **cub1** 的 **length** 成员，它的值是 1；同理，**cub1.width** 的值就是 2，**cub1.height** 的值就是 3。所以最后会算得 **volume1** 的值是 6。

我们知道，同一个类型的不同数据可以存储不同的值，这是因为在内存中有各自的存储空间，互不干扰。对于结构体的对象来说也是如此，我可以定义若干个对象，并给它们不同的值，这时它们是互不干扰的。

```
Cuboid cub2 {3,5,7}, cub3 {4,6,5}; //再定义两个Cuboid类型的对象
```

这就意味着 **cub1**, **cub2** 和 **cub3** 有着各自的存储空间，互不干扰。我们可以用取地址运算符 **&** 来返回它的地址——也就是它存储位置中第一个字节的地址。

<sup>4</sup>Lambda 除外，这个留到精讲篇再谈。

```
Cuboid cub1 {1,2,3}, cub2 {3,5,7}, cub3 {4,6,5};
cout << sizeof (Cuboid) << endl //输出Cuboid类型的内存占用
<< &cub1 << endl << &cub2 << endl << &cub3 << endl; //分别输出地址
```

程序的运行结果如下<sup>5</sup>:

---

```
12
0x7ffc5cc022c0
0x7ffc5cc022d0
0x7ffc5cc022e0
```

---

从这个运行结果中我们能看出，Cuboid 类型的内存占用是 12 个字节，但是我定义的三个 Cuboid 类型的对象分别位于 ...2c0, ...2d0 和 ...2e0 位置上，每两个地址相差 16 字节——也就是说，它们在内存中不是紧密排布的，相互之间间隔了 4 个字节。

其实我之前在讲一维数组时就提过，C/C++ 标准从来就没有保证过“连续定义的若干变量在内存中必须是紧挨着的”，这就是一个绝佳的例证。也正因如此，`&cub1+1` 这种语法就是错误的，因为我们不能保证它还指向有效信息。如果我们希望让它紧密排布，那么我们应该怎么做呢？很简单，定义一个数组。

```
Cuboid cubs[3] {{1,2,3},{3,5,7},{4,6,5}}; //定义一个数组
for (Cuboid cub : cubs) { //范围for循环
    cout << "长度" << cub.length
    << ", 宽度" << cub.width
    << ", 高度" << cub.height
    << endl; //输出长、宽、高，然后换行
}
```

这个程序的运行结果就是

---

```
长度 1, 宽度 2, 高度 3
长度 3, 宽度 5, 高度 7
长度 4, 宽度 6, 高度 5
```

---

## 结构体与函数

结构体把数据包装得更好，这样我们就可以把它作为一个完整的单元，传给函数作为参数，或者是作为函数的返回值。举个例子，我们要写一个 `rotate_horizontal` 函数，来水平方向旋转这个长方体，把长度和宽度颠倒过来。

```
void rotate_horizontal(Cuboid &cub) { //引用传递
    swap(cub.length, cub.width); //调用标准库中的swap函数，注意需要utility库
}
```

在这里我们可以直接对 `cub.length` 和 `cub.weight` 成员进行交换，因为是引用传参，所以这样就可以直接修改传入的实参。

```
Cuboid cubs[3] {{1,2,3},{3,5,7},{4,6,5}}; //定义一个数组
for (Cuboid &cub : cubs) { //在范围for循环中，如果要修改cubs，应该用引用
```

<sup>5</sup>输出的地址值和内存占用量可能因设备而异。总而言之，运行结果不唯一。

```

rotate_horizontal(cub); //引用传参，将会修改实参cub
cout << "长度" << cub.length
<< "， 宽度" << cub.width
<< "， 高度" << cub.height
<< endl; //输出长、宽、高，然后换行
}

```

这个程序的运行结果就是

---

```

长度 2, 宽度 1, 高度 3
长度 5, 宽度 3, 高度 7
长度 6, 宽度 4, 高度 5

```

---

看起来非常完美地实现了我们的目标。

我们还可以做一些其它的功能，比如说写一个函数，每次调用它时就用 `new` 来创建一个新的长方体，并返回它的地址，这样我们就可以用一个指针来接收它。

```

Cuboid* create_cuboid(int l, int w, int h) { //返回值是指向Cuboid的指针类型
    return new Cuboid {l, w, h}; //创建动态对象，并用l, w, h初始化。
}

```

于是我们就可以使用它了。

```

Cuboid *pcub {create_cuboid(5, 12, 13)}; //用create_cuboid创建一个新长方体
//...
delete pcub; //不要忘记！

```

我们发现，输出 `Cuboid` 对象信息要写很长一串代码，我们也可以写一个函数来实现这个功能。这样我们就不需要每次很麻烦地写这么多代码了，直接调函数来就好。

```

void output_cuboid(const Cuboid &cub, ostream &out = {cout}){
    out << "长度" << cub.length
    << "， 宽度" << cub.width
    << "， 高度" << cub.height
    << endl; //输出长、宽、高，然后换行
}

```

这里我们用 `const Cuboid &cub` 的原因是，传引用一般要更节省内存空间<sup>6</sup>。而我们在这里不需要修改 `cub`，所以把它设成 `const` 可以防止篡改它的值。

至于 `out`，我们为它设计了一个默认值 `cout`。如此，如果我们想用 `cout` 来输出的话，就不需要写第二个参数了。其实，更合理的参数列表写法是先 `ostream&` 再 `const Cuboid&`；但是鉴于默认参数必须设置在列表右侧，所以这样是不得已而做出的设计。

## 结构体成员的类型

刚才的例子比较简单，`Cuboid` 的三个成员都是同一类型的。实际上我们可以用不同类型的数据，把它们组织到同一个结构体中。

例如，如果要表示一个人的基本信息，我们可能需要用字符串表示名字，用 `Sex`（上一节中自定义的枚举类型）表示性别，用 `double` 身高、体重，用 `unsigned` 表示年龄。那么我们可以这样写：

<sup>6</sup>并不总是如此，比如说对于 `char` 类型来说，传值只需要 1 个字节的空间临时变量就行，但传引用需要 4 或 8 个字节的临时指针（传引用的本质是传指针）。

```

enum Sex : bool{male, female}; //枚举基为bool
struct PersonalInfo { //一个结构体，表示个人信息
    char name[33]; //字符串，表示名字
    const Sex sex; //性别一般是不会改变的，所以设置成const
    double height;
    double weight;
    unsigned age;
};

```

接下来我们可以定义一些函数，比如这个函数可以用来输出某个人的个人信息：

```

void output_info(const PersonalInfo &person, ostream &out = {cout}) {
    out << person.name << ", " //输出字符串
        << (person.sex == male ? "男" : "女") << ", " //条件表达式
        << person.age << "岁" << endl //换行
        << "身高" << person.height << ", "
        << "体重" << person.weight << endl;
}

```

然后我们就可以在主函数中写一些代码来测试它的行为了。

```

int main() {
    PersonalInfo group[3]{
        {"John Doe", male, 175.5, 70.2, 30},
        {"Jane Smith", female, 162.3, 55.8, 25},
        {"Bob Johnson", male, 180., 80.5, 35}
    }; //定义一个PersonalInfo[3]，分别为它们初始化
    for (PersonalInfo person : group) {
        output_info(person); //在范围for循环中输出每个人的信息
        cout << endl; //为了区分，每两人的信息之间多换一行
    }
    return 0;
}

```

这个程序的运行结果如下：

---

John Doe, 男, 30 岁  
身高 175.5, 体重 70.2

Jane Smith, 女, 25 岁  
身高 162.3, 体重 55.8

Bob Johnson, 男, 35 岁  
身高 180, 体重 80.5

---

读者可能注意到 `person.sex==male?"男":"女"` 此段中我们使用的条件表达式。如果 `person.sex==male` 为 `true`，那么就会返回 "男"；否则返回 "女"。

看上去无论是内置类型还是自定义类型，我们都可以把它放到 **struct** 当中，构成一个结构体。那么有什么是不可以放入其中构成结构体的呢？那就是这个结构体本身！在函数定义中我们见过递

归定义，但是结构体是不允许递归定义的。

```
struct Data {
    int num;
    Data next; // 不允许
};
```

这是因为，如果我们递归定义的话，那么程序就不知道这个类型占用的内存空间有多大了——这个类型的大小等于这个类型的大小加上一些杂七杂八的东西，这是不合理的！

但是这个类型的成员中可以有指向这个类型的指针。

```
struct Data {
    int num;
    Data *next; // 可以
};
```

`Data*` 与 `Data` 可不是同一个类型，而且 `Data*` 是一个指针，它占用内存空间的大小是确定的，所以程序当然知道 `sizeof(Data)` 是多少，所以在我们定义 `Data` 对象时也就知道要使用多大的内存空间了。

基于这个用法，我们可以写一个简单的单链表，用来存储任意量的数据。我们将会在下一节中介绍相关内容。

## 6.3 实操：用结构体实现单链表

### 什么是单链表？

单链表是一种线性的数据结构<sup>7</sup>。它的数据存储方式是这样的：每个数据单元有两部分，其中一部分存储数据；而另一部分是一个指针，它指向下一个单元所在的位置。下一个单元也是两部分，其中一部分存储数据，而另一部分是一个指针，它指向下一个单元所在的位置。它和数组的区别在于，数组不需要单独的指针来存储下一个数据的位置，因为下一个位置就是取地址加一；而链表的各个单元可能分布于内存的各个区域，所以只有通过指针才能找到下一数据的位置。图 6.2 展示了这种关系。

那么我们为什么要用链表而不是更容易实现的数组呢？这取决于我们的需求。在实际操作中，如果我们需要对一个数组的内容进行频繁的插入、删除操作，那么我们就需要在插入数据之前把对应位置之后的所有数据都向后挪一位；而如果要删除某个数据，我们就需要在删除数据之前把此位置之后的所有数据都向前递补。这是非常麻烦的。

而链表就能很高效地完成数据的插入和删除操作，如图 6.3 所示。如果要插入一个单元，那就把前一个指针指到新单元的位置上，并且让新指针指向下一个单元，于是插入操作就完成了。如果要删除一个单元呢，那就把前一个指针绕过本单元，直接指到后一个单元上，于是待删除的单元自然就可以抽身而去。

不止如此，单链表更大的优点在于可以方便地插入或删除一段数据，或者是把原链表中的一段转移到其它链表中。图 6.4 展示了片段转移的过程。无论这个片段有多长，我们都可以只通过两个起止点把待转移的片段摘下来，然后插入到另一个链表的某单元之后。

那么了解了链表的基本概念之后，我们就可以试着自己来写一个链表，并配以相应的函数，来实现链表单元的插入、删除和整段转移操作。

---

<sup>7</sup> 数据结构（Data structure），是一种组织、存储和管理数据的方式，它把一些数据通过某种特定的方式组织起来，以便于我们高效地访问和处理数据。

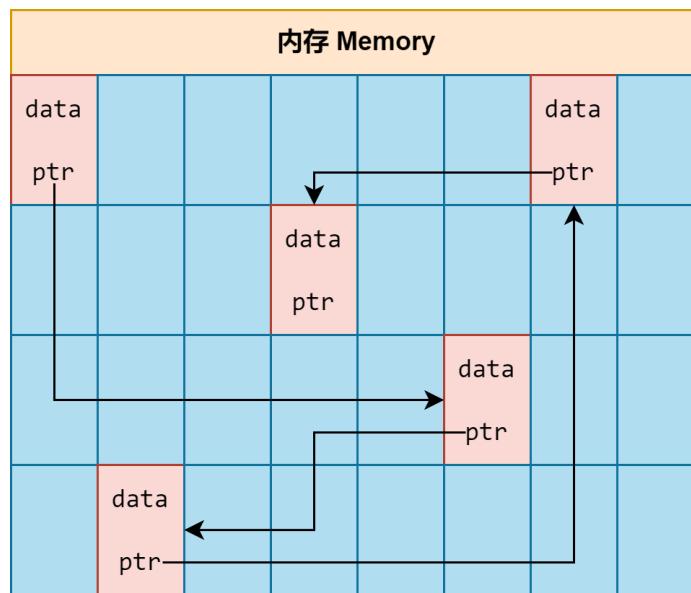
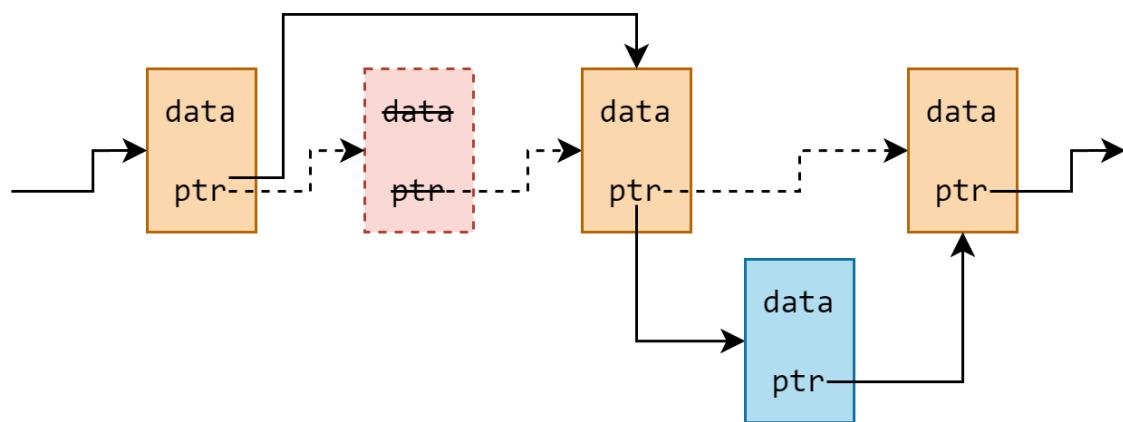
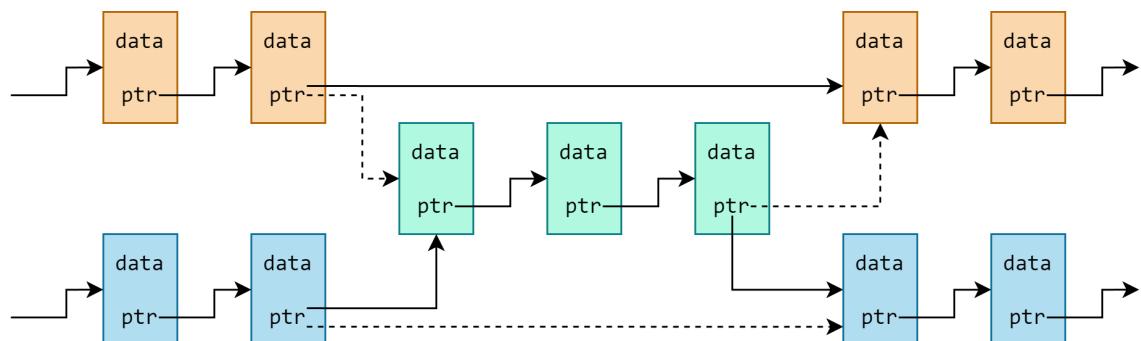


图 6.2: 链表单元可以在内存中分散于各处

图 6.3: 链表的插入和删除操作  
蓝色的单元是待插入单元，红色的单元是待删除单元图 6.4: 链表的整段数据转移操作  
将绿色段从黄色链表中转移到蓝色链表中

## 链表的构建与内存回收

链表的构建方式并不唯一，具体支持哪些功能也取决于我们的实际需求。在这里，出于教学目的，我们删繁就简，写一下链表的基本功能就行。

链表的每个单元都包含两部分——一个数据和一个指针，其中的指针要能指向某个单元。所以我们自然选择结构体的方式来实现它。

```
struct Data {
    int num;
    Data *next;
};
```

我们可以定义一个不起存储作用的链表头 `head`，它只负责指向本链表的第一条有效数据，比如这样：

```
Data head {0, nullptr}; // 链表头，指向nullptr以免出现野指针
head.next = new Data {5, nullptr}; // 在链表末尾插入一个单元，用new分配
(*head.next).next = new Data {8, nullptr}; // 再插入一个单元
```

这里的 `head.next` 就是指向第一个有效数据的指针。如果我们要指向第二个有效数据，我们就要先用 `*` 运算符取内容，然后再用成员访问运算符 `.` 找 `next`。这样也太麻烦了。

C++ 为我们提供了更方便的方式，我们可以用指针的成员访问运算符 `->` 来直接通过指针访问对应的成员，而不必麻烦再取内容了。

```
head.next->next = new Data {8, nullptr};
//head.next是一个指针，可以用指针的成员访问运算符直接得到它指向的next
head.next->next->next = new Data {12, nullptr};
//head.next->next同样是一个指针，我们还可以用->访问其内容
```

最后别忘记一个问题——我们是用 `new` 来添加新单元的，这也就意味着我们要用 `delete` 回收。但是我们不能直接 `delete head.next` 了事。这是因为，虽然 `head.next` 被回收了，但是它指向的那些内存地址还没有被回收。同时，因为 `head.next` 已经被回收了，所以我们也不能再通过 `head.next->next` 之类的语法来回收剩下的内存——也就是说，还有一个顺序的问题。

所以我们要考虑清楚怎么在建立一个单链表后妥善地回收动态内存，不然就会出现内存泄漏。正确的思路是，先回收链表末端的单元，再逐个向前依次回收，最后回收 `head.next`。至于 `head`，它本来就是静态的，我们不需要回收，作用域结束之后它自然就没了。

但是另一个棘手的问题在于，我们不知道末端的单元在哪里，所以要从 `head.next` 开始一个一个找，直到找到某个“`next` 指向 `nullptr`”的单元为止。但是费了这么大劲才回收了一个单元，接下来我们还要再来这么好多轮，才能把所有单元都回收——这也太麻烦了。下面是这种思路的代码，它是一种最低效的方法，仅供读者了解。

```
while (head.next != nullptr) { // 如果head之后的内容还未清空
    Data *p = head.next; // 定义一个临时指针，准备从head.next开始向后寻找
    while (p->next->next != nullptr) { // 说明*p->next还不是末尾单元
        p = p->next; // p向后移动一位
    }
    delete p->next; // 现在*p->next就是最后一个单元，程序会回收这里的内存
    p->next = nullptr; // 现在*p是最后一个单元
}
```

这个方法既麻烦，又低效；而我们有更好的方法，一种是用删除的逻辑从头部开始清理，而另一种是用递归的方法从尾部开始清理。我们只讲第二种：

```
void clear_list(Data *head) { // 递归回收 *head 之后的所有动态内存
    if (head->next == nullptr) // 如果 *head 之后没有动态内存
        return; // 那就不需要回收什么了，直接 return；结束本次调用
    clear_list(head->next); // 先清理 head->next 之后的部分
    delete head->next; // 再回收 head.next
    head->next = nullptr;
    // 现在 head 之后没有动态内存了，我们可以放心地把 head->next 置为 nullptr
}
```

这个函数的设计目的是：任意给定一个头部单元指针 `head`，回收 `*head`（不含）之后的内存。而它的递归思路是：每次调用时，先用 `clear_list(head->next)` 把 `*head->next`（不含）之后的内存回收，然后回收 `*head->next`。



图 6.5: 递归回收链表的过程示意图

## 链表的基础功能

接下来我们考虑对单个单元进行插入和删除操作。为了简化，我们规定：插入操作只能插入到指定单元的下一位，而删除操作只能删除指定单元的下一个单元。这些操作都很简单，但是你需要非常小心——尤其是在语句的顺序安排方面。

我们先来看插入操作。这个操作的具体过程可以分为三步，见图 6.6 及代码中注释。

```
void insert_after(Data *head, int n) { // 在 *head 下一位置插入 n
    Data* p = { new Data{n, nullptr} }; // 分配动态内存，并初始化其 num 成员
    p->next = head->next; // 插入数据的下一位指向当前的 *head->next
    head->next = p; // head->next 指向 *p，注意顺序不要颠倒！
}
```

读者一定要注意这些操作的顺序安排。如果我把 `head->next=p` 放在前面，那就会出现一个问题，`head->next` 赋值之后，它原本所指向的单元就找不到了。

当你熟悉了语法之后还可以把这三步简化成一步：

```
void insert_after(Data *head, int n) { // 在 *head 下一位置插入 n
```

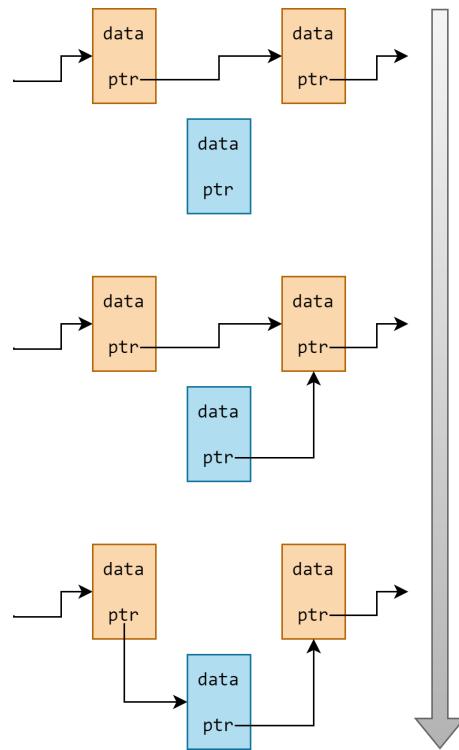


图 6.6: 插入数据的过程

```
head->next = new Data{n,head->next}; //三步并作一步
}
```

再来看删除操作，我们需要写一个函数，给定某个 `Data *head`，这个函数能删除它后面的一个单元。这个操作的过程可以分为两步，见图 6.7 及代码中注释。

```
bool delete_after(Data *head) {
// 删除*head下一位位置的数据，并返回true；若*head是末尾元素，删除失败，返回false
    if (head->next == nullptr) //说明*head就是末尾元素
        return false; /*head是末尾元素，无法再删除后面内容
    //这里应当用else块，不过也可以省掉
    //因为if块内会直接return，所以后面的代码肯定是在条件为false情况下运行的
    Data *p = head->next; //临时指针p指向*head->next
    head->next = p->next; //head->next指向更后一个元素
    delete p; //回收p指向的内存
    return true;
}
```

这里指得一提的是返回值。按理说删除操作只是 `delete_after` 函数的副作用，不要求值。但是有些情况下我们可能需要根据返回值的实际情况来调整其它值。例如，我可能有另外一个值用来记录这个单链表的长度，那么返回值就决定了我们要不要在删除操作之后让这个长度减一。

```
if (delete_after(ptr)) { //如果删除成功
    --size; //size自减
}
//否则size不变，什么也不做
```



图 6.7: 删除数据的过程

## 链表的进阶功能

接下来我们可以来试着实现一些链表的进阶功能：片段插入、片段删除和片段转移。其中的片段插入可以用片段转移的方式来实现，所以我们只需要关注后两个就行。我们先来看片段删除操作。

我们在这里要写一个函数，删除从某个单元（不包含）开始到某个单元（包含）为止。<sup>8</sup>

```
bool delete_after(Data *head, Data *tail) {
    // 删除 *head 之后（不包含）直到 *tail 为止（包含）的单元。若删除失败，返回 false
    if (head == tail) // 说明没有任何数据需要清理
        return false; // 删除失败
    if (tail == nullptr) { // 说明我们需要删除 *head 之后的所有内容
        clear_list(head); // 当然是用 clear_list 啦
        return true; // 删除成功
    }
    Data tmp_head {*head}; // 用 *head 来直接为 tmp_head 初始化
    head->next = tail->next; // 绕过待删除段，直接指向 *tail->next
    tail->next = nullptr; // tail 指向 nullptr，现在待删除片段就被剥离出来了
    clear_list(&tmp_head); // 以 tmp_head 为头，清理这段链表
    return true; // 删除成功
}
```

这段代码对于初学者来说可能有点复杂，所以看不懂也没关系。如果还可以勉强看懂一部分，那不妨再看一看下面的讲解：

这里的 `delete_after` 是一个重载函数，它既有 `(Data*)` 版本，又有 `(Data*, Data*)` 版本。当传入单个指针时，就意味着删除一个单元；当传入两个指针时，就意味着删除一串单元。这种重载思路在 STL 中比较常用。

在函数体部分，我们需要判断一种非法情况，也就是 `head==tail`。在这种情况下，`head` 到 `tail` 之间的片段是空，所以这种操作没有意义。

那么在 `tail==nullptr` 的情况下，我们可以直接删除从 `head`（不包含）开始的所有单元。这样的话，最佳选择就是直接使用内存回收函数 `clear_list` 解决问题。

现在我们再来考虑一般的情况——`head` 和 `tail` 都是这个链表中的单元。<sup>9</sup> 我们的思路是把这个

<sup>8</sup>关于“是否包含两端”的问题，其实有很多种写法，都可以实现。在此处我们选择使用此套方案，即左端不包含而右端包含。

<sup>9</sup>实际的情况远比这个复杂，比如说 `head==nullptr` 的时候怎么处理；`head` 和 `tail` 不在同一链表中的情况怎

片段给剥离出来，变成一个单独的临时链表，然后用 `clear_list` 直接清理这个单独的链表，如图 6.8 所示。

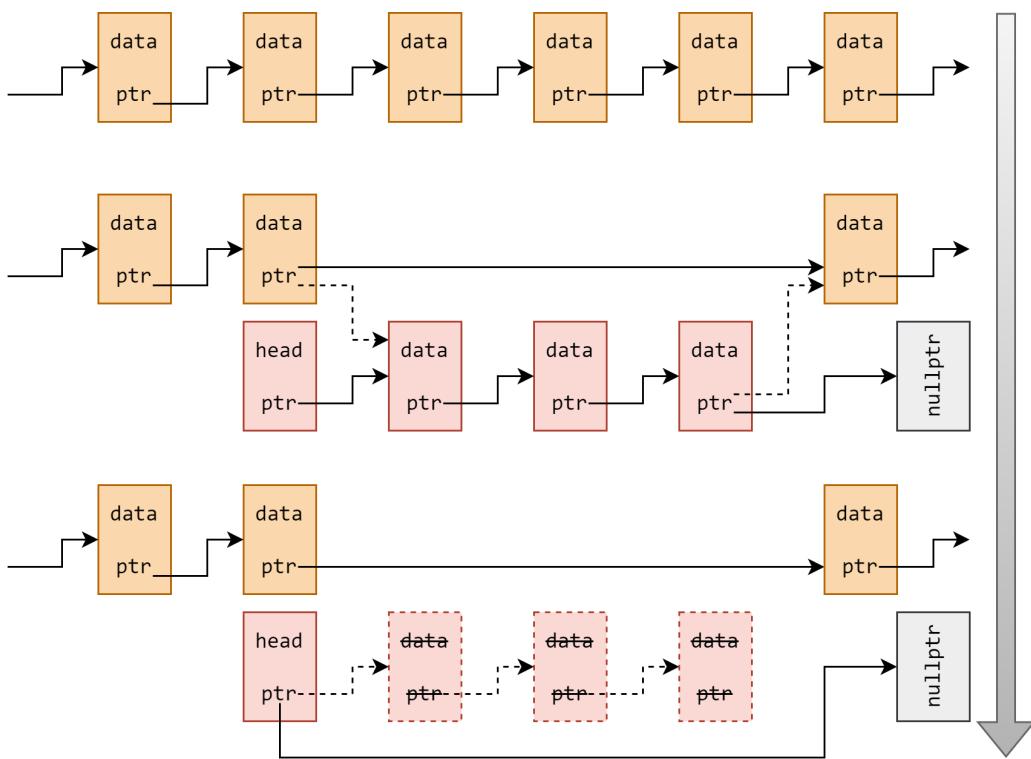


图 6.8: 链表的片段删除操作

在剥离操作中，我们定义了一个临时的链表头 `tmp_head`，并且用 `*head` 来初始化它<sup>10</sup>，这样 `tmp_head` 和 `*head` 就有了相同的值，只是它们地址不同。我们就打算把待删除的片段转移到以 `tmp_head` 为首的链表中。

在剥离的过程中我们也要注意顺序，否则就容易把信息搞丢：第一步，我们把原链表缝起来，这时 `tmp_head` 还能连到待删除段，并且 `tail` 也能找到需要缝起来的部分，所以一切安然无恙；第二步，我们把 `tail` 段也拆下来，这样就彻底断绝了两个链表的联系。

剥离完成之后我们直接清理 `tmp_head` 链表的动态内存就行，这里很简单。

那么我们再来看链表片段转移的操作。一个片段转移函数——我们起名 `transfer`，它应该有三个参数：前两个参数描述这个片段从哪里到哪里，后一个参数描述这个片段要转移到何方（目的地）。

读者可以延续我们刚才的思路，并参考图 6.4 的示意来自行完成这个函数。而这里我们选择另一种写法：

```
void transfer(Data *head, Data *tail, Data *dest) {
    // 片段转移，把*head之后（不包含）直到*tail为止（包含）的单元移到*dest之后
    if (head == tail || dest == nullptr) // 这两种情况下转移没有意义
        return;
    if (tail == nullptr) { // 这种情况不方便我们处理，我们把tail改成指向末尾单元
        for (tail = head; tail->next != nullptr; tail = tail->next) {
            ; // for 循环的逻辑是，从head开始找，直到tail->next==nullptr为止
        }
    }
}
```

么处理，诸如此类。不过我们在这里只是用代码来进行演示和学习，不需要像实际工程中那样做得面面俱到了。

<sup>10</sup> 在后面我们会讲到，这里用到了默认拷贝构造函数。

```

    if (head == tail) // 仍需当心 head==tail 的可能
        return;
}
swap(tail->next, dest->next); // swap 函数需要 utility 或 string_view 库
swap(head->next, dest->next); // 是一种技巧，读者不必强求掌握
}

```

这里再说明两点：

其一，在这里 `tail==nullptr` 的情况下我们要处理是比较麻烦的，因为当后面涉及到 `tail->next` 的时候我们必须得让 `tail` 有意义才行——`nullptr` 肯定不行。所以为了解决这个问题，我们要把 `tail` 改成指向末尾单元。怎么做呢？只能从 `head` 开始一个个找，直到找到一个“`tail->next==nullptr`”的情况，这时 `tail` 就是我们想找的末尾单元了。

我们可以用循环结构来找，本段代码就是这样操作的。除此之外还可以递归，参照这段代码来找也行：

```

Data* find_tail(Data* head) { // 从 head 开始找
    if (head->next == nullptr) // 如果 head->next==nullptr
        return head; // 那么它就是我们要找的 tail
    return find_tail(head->next); // 否则就从 head->next 开始找
}

```

其二，这里转移操作的写法很独特（也很简洁，不是吗）。这里用文字很难讲清楚，读者可以直接看图 6.9，很直观。在这里我们用标准库中的 `swap` 函数，它可以交换两个指针的值<sup>11</sup>，也就相当于交换它们的指向。于是我们可以通过先交换 `tail->next` 和 `dest->next`，再交换 `head->next` 和 `dest->next` 的方式来实现这个功能。

## 6.4 联合体 union

联合体初看上去很像结构体，它们的定义语法极其相似，但是它们是完全不同的事物。

```

union ValType {
    int val_i;
    long long val_ll;
    char val_c;
    double val_d;
    long double val_ld;
};
int main() {
    cout << sizeof(ValType); // 输出 8 或 16，总之不是 29 或者更大的数
}

```

一个结构体会把所有成员分别存储在不同的内存区域中，所以这个类型的内存占用不能小于所有成员的总和。以第 2 节中定义的 `Cuboid` 类型为例，我们可以用这样一段代码来观察某个对象各成员的地址：

```

Cuboid cub;
cout << &cub.length << endl << &cub.width << endl << &cub.height;

```

<sup>11</sup>实际上 `swap` 是一个函数模版，当我们尝试使用它时，程序会生成一个关于 `Data*` 类型的实例。我们会在第十一章中介绍相关的知识。

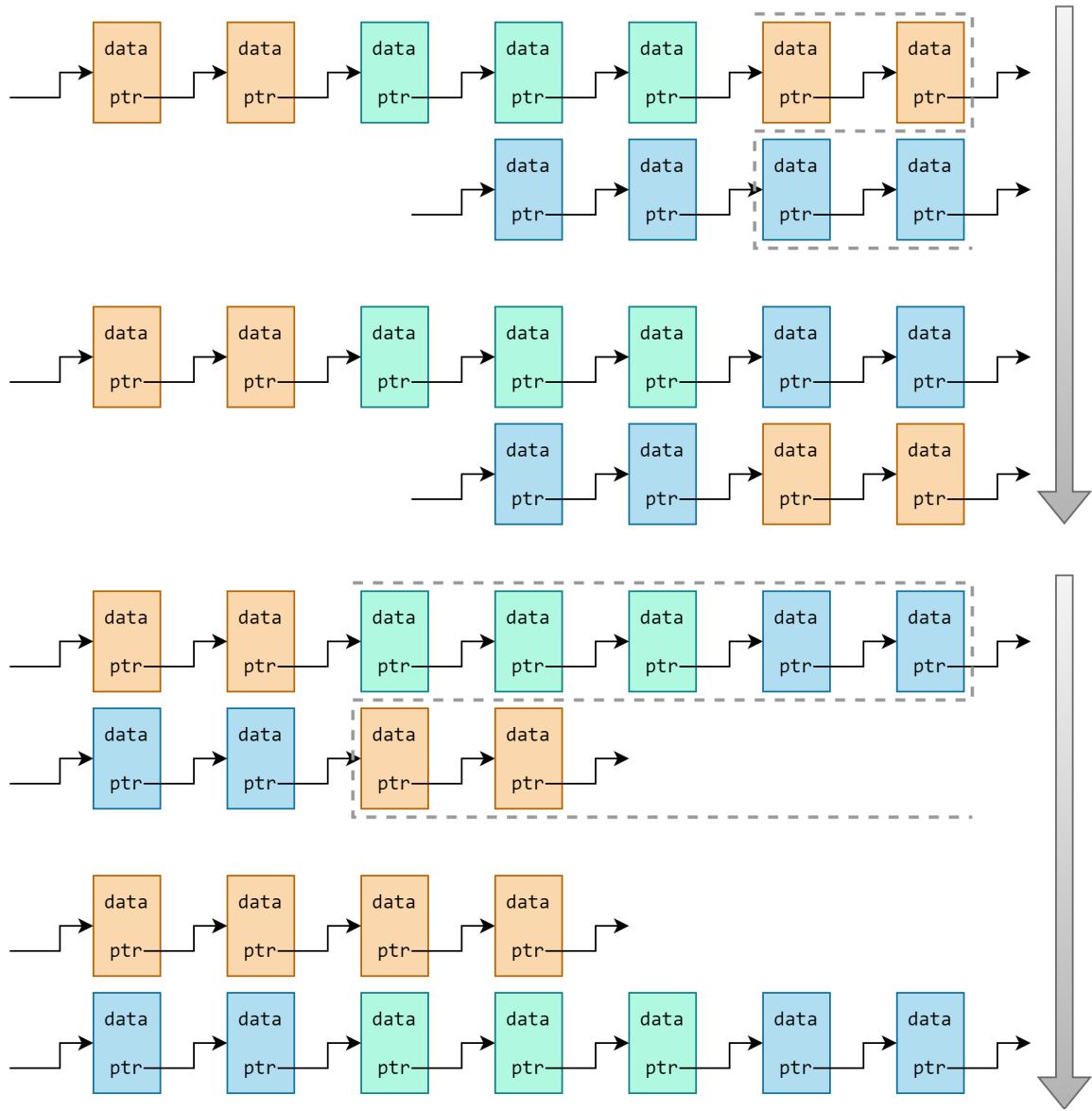


图 6.9: 片段转移操作的示意图

下面是一种可能的运行结果：

---

```
0x7fffea18b070
0x7fffea18b074
0x7fffea18b078
```

---

从结果中我们可以看到，`cub` 的三个 `int` 型成员分别在不同的内存空间中，`cub.length` 在 ...070 到 ...073 位置，`cub.width` 在 ...074 到 ...077 位置，`cub.height` 在 ...078 到 ...07b 位置。它们之间没有任何重叠，互不干扰。

但联合体不然，它会把所有成员存储到同一个内存区域中。

```
ValType a;
cout << &a.val_i << endl
    << &a.val_ll << endl
    << (void*)&a.val_c << endl //记得把char*转成void*输出
    << &a.val_d << endl
    << &a.val_ld << endl;
```

下面是一种可能的运行结果：

---

```
0x7ffc00a4d280
0x7ffc00a4d280
0x7ffc00a4d280
0x7ffc00a4d280
0x7ffc00a4d280
```

---

这说明联合体的所有成员都在同一个位置存着呢。

## 联合体如何组织内存？

正如我们才看到的那样，联合体会把所有的成员放在同一个内存区域中。所以 `ValType` 的内存占用不需要达到全部成员的内存占用之和，只需要等于它们之中最大的那个就行。在这里，如果你的开发环境中 `long double` 类型占用 8 字节，那么 `sizeof ValType` 的结果一般就是 8 字节；如果你的开发环境中 `long double` 类型占用 16 字节，那么 `sizeof ValType` 的结果一般就是 16 字节。

既然五个成员全都占用同样的内存空间，那么它们之间势必会存在冲突。回想一下章首的图 6.1 ——我们用不同类型去解释同一块内存空间中的内容，将会得到不同的结果。在这里也是如此。如果我们用 `a.val_i` 去修改这段内存中的内容，那么它将会按照 `int` 型对信息的组织方式来存储 0/1 串。在这种情况下，输出 `a.val_c` 或 `a.val_d` 往往是看不出什么有效内容的。

既然这些成员之间互相有冲突，那么我们为什么还要用联合体呢？这是因为，在有些情况下，一个联合体的成员是彼此互斥的，它们不会同时出现。还是以 `ValType` 为例，如果用一个结构体来表达它的话，需要 29 或更多字节才能容纳——其中有用的信息永远不超过 16 字节，那么剩下的空间就浪费了。通过联合体，我们可以把这些数据都存到这 8 或 16 个字节的空间当中，只要它们不同时出现，那就不会存在冲突了。举个现实一点的例子：像 Python 这样的编程语言支持动态类型，即，一个变量可以在某个时候是整型，而在之后变成字符型或浮点型。而 C++ 是静态类型语言，一个变量的类型在它定义的时候就确定下来了，不能更改。通过联合体，我们可以在一定程度上实现动态类型的的部分功能。

```
struct dynamic { //自定义动态类型
```

```

enum Types{integer,floating,string}; //枚举，用来规定可能的类型
Types type; //type用来标记当前的类型
union Value { //联合体，它的三个成员不会同时出现
    long long vll; //整型
    long double vld; //浮点型
    char str[16]; //字符串型
};
Value value; //定义Value类的对象value
};

```

我来解释一下这段代码：我们定义了一个 `dynamic` 类，这个类的对象可能以三种状态存在：整型状态，浮点型状态或字符串型状态。`type` 是一个枚举类型，它可以用来标记这个对象当前处于何种状态。

而在联合体 `Value` 中，三个成员 `vll`, `vld` 和 `str` 分别是整型、浮点型和字符串型。在任何时候，这个变量只能是这三种类型之一，所以它们不会同时出现，用 `union` 就很合理。

需要读者留意的是，在 `dynamic` 中定义 `Value` 的操作不会直接引入一个什么实体。如果我们不用 `Value value` 这句来定义一个 `Value` 类的对象，那么这个内嵌的定义就没有什么存在感了。

## 如何使用联合体？

那么我们就在主函数中定义一个 `dynamic` 类的对象，并设置它的类型和值。

```

dynamic number {dynamic::integer}; //integer在dynamic域中，所以用dynamic::
number.value.vll = 15; //修改dynamic.value的vll成员，把它变成15

```

这里需要注意，`integer` 是 `dynamic` 域中的枚举项。如果我们要在域外使用，就必须用 `dynamic::integer`。相关细节，我们留到第七章再谈。

在本段代码中，我们定义了一个 `number` 变量，并初始化它的 `type` 成员为 `integer`。接下来，我们用 `number.value.vll` 来为 `value.vll` 成员赋值。一旦为 `vll` 赋值，这时 `vll` 就是活跃成员，而 `vld` 和 `str` 都是不活跃成员。试图访问不活跃成员是未定义行为，会得到不确定的结果<sup>12</sup>。

下一刻，我们想把 `number` 改成浮点型。这个操作非常简单，只要为 `vld` 赋值，就可以把它变成活跃成员，顶掉 `vll`。

```

number.type = dynamic::floating; //修改number的类型，标记为浮点数
number.value.vld = number.value.vll; //vld将取代vll成为活跃成员
cout << number.value.vld; //会输出15

```

第一行的操作就是更改一下标签，不必多说。第二行的操作却有些费解——不是说同一个联合体的不同成员不该同时出现吗？为什么我们在这里可以用 `vll` 给 `vld` 赋值还不产生问题呢？

其实这就是“同时”的问题了。赋值运算符不是一下子就把右操作数的值赋给左值的。赋值操作的内部过程可以分成两步：第一步是处理右操作数，把右操作数的值转移给一个临时变量<sup>13</sup>；第二步是把这个临时变量的值转移给一左操作数，临时变量销毁。我们看，在第一步的时候 `vll` 是活跃成员，这时我们没有用到 `vld`，没有问题；而在第二步的时候 `vld` 是活跃成员，这时我们没有用到 `vll`，也没有问题，如图 6.10 所示。

试过了这些简单功能之后，我们可以写一些函数来实现它们。以下是一个输出函数，它根据 `type` 来判断要输出哪个成员。

<sup>12</sup> 约等于定义了局部变量但还没有赋值/初始化就开始使用它，会得到不确定的结果。

<sup>13</sup> 这是一个左值到右值的转换，可能还会伴随类型转换。

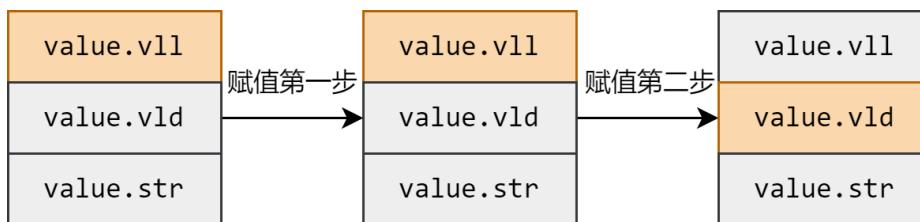


图 6.10: 赋值过程中, 活跃成员的变化

```

void output(const dynamic &number, ostream &out = {cout}) {
    switch (number.type) { //用switch来判断number.type的值
        case dynamic::integer: //如果是整型
            out << number.value.vll; //输出vll
            return; //直接用return返回; 或者用break也行
        case dynamic::floating: //如果是浮点型
            out << number.value.vld; //输出vld
            return;
        case dynamic::string: //如果是字符串
            out << number.value.str; //输出str
            return;
    }
}

```

这个函数仍然沿袭我们的思路, 以 `cout` 作为输出的默认参数。在函数中, 我们用 `switch-case` 结构来实现对 `number.type` 的判断。这个函数没有什么技术含量, 读者想必很容易就能看懂。

```

dynamic& assign_int(dynamic &number, long long ll) {
    number.type = dynamic::integer; //更改type
    number.value.vll = ll; //赋值给vll, 现在它是活跃成员
    return number;
}

dynamic& assign_float(dynamic &number, long double ld) {
    number.type = dynamic::floating; //更改type
    number.value.vld = ld; //赋值给vld, 现在它是活跃成员
    return number;
}

dynamic& assign_str(dynamic &number, const char *src) {
    number.type = dynamic::floating; //更改type
    strncpy(number.value.str, src, 16); //用strncpy为str赋值, 现在它是活跃成员
    return number;
}

```

在这里, 我们定义了三个函数, 分别用于给 `number` 赋特定类型的值。至于字符串的赋值, 我们提过, 字符串不能直接赋值, 要用 `strcpy` 或 `strncpy` 才行。这两个函数在头文件 `cstring` 中。`strncpy` 比 `strcpy` 更安全, 它可以规定第一个操作数最多能接收的字符数量, 以防字符串赋值时发生越界问题。

我们还可以借助联合体玩更多花样。不过相比于 `class/struct` 来说, 它的应用范围还是很窄的。接下来的章节中我们几乎不会再用 `union` 了, 所以这里就仅为读者开拓一下眼界。想要了解关于联合体的更多用法, 可以参考 [When would anyone use a union? Is it a remnant from the C-only](#)

days?-Stack Overflow。

## 6.5 类 `class` 初步

“一个设计良好的类能为用户提供简洁易用的接口<sup>14</sup>，并将其内部结构隐藏起来，用户根本不必了解其内部结构。如果内部结构不应该被隐藏——例如，因为用户需要随意改变类中的任何数据成员——你可以把这种类认为是‘普通的老式数据结构’。”<sup>15</sup>

——比雅尼·斯特劳斯鲁普

我们在第二节中所介绍的结构体更接近于 C 风格。它只能实现“把多个数据包装到一个对象中”的功能。而在使用时，我们可以任意地改变它的成员变量。这样做的好处是方便，简单，直接。如果我们要写一个结构体来管理单链表，我们完全可以这样写：

```
struct List {
    struct Data { // 在 List 内部定义 Data 类
        int num;
        Data *next;
    };
    Data head; // Data 类的成员 head
};
```

接下来我们只要用它就行了。

但是当我们要用这个单链表时，那就有些麻烦了：

我们必须知道这个链表是怎么实现的，并且亲自去修改它的成员。为了解决这个问题，我们尚且可以写一些函数来针对 `struct` 的成员进行修改。

结构体对外不封闭，程序员可以随意更改成员变量的值。这就会带来安全性问题，我们不能防止自己或别人因为各种原因（失误也好，有意操作也罢）乱改结构体的成员（尤其是乱改指针的值，指针可不是闹着玩的）。

C++ 大大扩展了 `struct` 的能力，并引入了关键字 `class` 作为 `struct` 的替代<sup>16</sup>。

### 类的成员访问权限

在普通的老式数据结构中，所有的成员都毫无保留地公开，外界可以任意地读取或修改它；而在类中，我们会把成员的访问权限分成三种：

- `public` 公有成员。它们是对外公开的，外界可以访问。
- `private` 私有成员。它们对外不公开，外界无法访问。只有类中的其它成员<sup>17</sup>才能访问或修改它。外界如果想要访问它，只能以 `public` 成员作为中介。

<sup>14</sup> 接口（Interface），在信息技术中指能将不同部件连接起来的媒介。通过接口，不同部件之间可以进行信息交换。例如，笔记本电脑有 USB 接口，如果手机通过数据线连到这个接口上，那么笔记本电脑就可以和手机交换信息。

<sup>15</sup> 原文：A well-designed class presents a clean and simple interface to its users, hiding its representation and saving its users from having to know about that representation. If the representation shouldn't be hidden - say, because users should be able to change any data member any way they like - you can think of that class as ‘just a plain old data structure’.

<sup>16</sup> `class` 与 `struct` 只有两个方面的不同：其一，`class` 的默认成员访问权限是 `private`，而 `struct` 的默认成员访问权限是 `public`；其二，`class` 的默认继承方式是 `private`，而 `struct` 的默认继承方式是 `public`。除此之外，`class` 和 `struct` 没有任何本质差异。

<sup>17</sup> 一般是成员函数，见下文。

- **protected** 受保护成员。它们对外不公开，但对派生类<sup>18</sup>公开。

**private** 成员是类与老式数据结构的关键区别。通过 **private** 成员，我们可以将类的一些内部结构隐藏起来，只有这个类中的其它成员能够访问。

举例来说，`istream` 就是一个类，我们一直在使用它的一个名为 `cin` 的对象。这个 `istream` 类的内部结构是怎样的？它如何把我们的输入信息解释出来，并且赋值给右操作数？它如何标记输入异常状态？这些内容纷繁复杂，C++ 自有它的实现机制。

但是我们不用考虑这么多！我们只需要在输入的时候写一个 `cin>>num` 就行了；如果要清理输入异常，我们直接调用成员函数 `clear()` 就够了！C++ 把底层的实现机制放到 **private** 成员中，而 **public** 成员则是它对外的接口，我们用的 `clear` 就是一个 **public** 成员<sup>19</sup>。图 6.11 体现了这两种对象的关系。

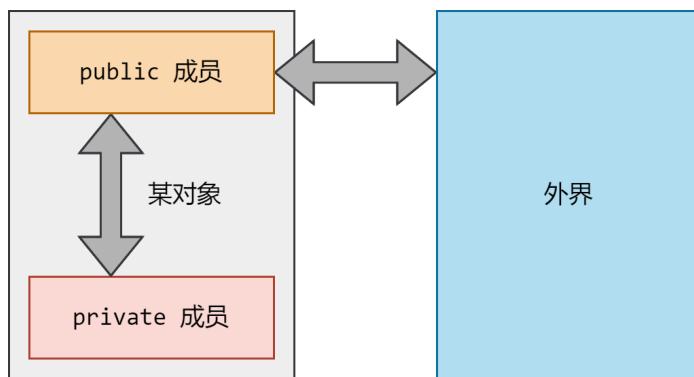


图 6.11: **public** 成员提供对外界的接口，而 **private** 成员隐藏了底层的细节

至于 **protected** 成员，在不涉及继承时，它和 **private** 成员没什么区别，我们可以暂时把它当作 **private** 成员来看待。

举个例子，`string` 库中的 `string` 类<sup>20</sup>提供了强大的字符串存储和处理功能，比如说它可以动态地改变字符串的容量，用 `=` 直接为字符串赋值，以及用 `+=` 拼接字符串等。

```

string str {"Bjarne\u00e5Stroustrup"}; // 需要头文件 string
str = "Donald\u00e5Ervin\u00e5Knuth"; // 可以用等号赋值！
cout << str << endl; // 将输出 Donald Ervin Knuth
str = "Alexander\u00e5Alexandrovich"; // str 会动态改变长度，不怕越界
cout << str << endl; // 将输出 Alexander Alexandrovich
str += "\u00e5Stepanov"; // 用 += 运算符拼接
cout << str << endl; // 将输出 Alexander Alexandrovich Stepanov
cout << str.length(); // 输出 str 当前的长度，应为 32

```

这里的赋值 `=`、加赋值 `+=` 和 `length` 都是在 `string` 类中定义的成员函数，它们是公有成员，我们可以直接使用；而这个类的底层实现是通过动态内存分配和各种成员变量来记录的，它们是私有成员，我们必须通过公有成员来找到它们。就以 `length` 成员函数为例吧，它是一个公有成员，在 `basic_string` 中的定义是这样的：

```

public:
    _NODISCARD _CONSTEXPR20 size_type length() const noexcept {
        return _Mypair._Myval2._Mysize;
    }

```

<sup>18</sup> 我们会在第九章介绍相关内容。

<sup>19</sup> 至于 `>>`，它不是 `istream` 的成员。它使用了友元，我们会在第八章介绍这些内容。

<sup>20</sup> 实际上它是 `basic_string` 类的一个实例，我们暂时不讨论这些细节问题，把 `string` 当成一个类即可。

函数体中给出了它的返回值 `_Mypair._Myval2._Mysize`, 这是什么呢? 再查, 会发现 `_Mypair` 定义在这里:

```
private:
    _Compressed_pair<_Alty, _Scary_val> _Mypair;
```

至于这个 `_Mypair` 具体是什么, 我们就不多追究了, 总之它的某个叫做 `_Myval2._Mysize` 的成员存储了这个长度信息。

再来看它们的访问权限。我们会发现, `length` 是公有(看到那个 `public` 了吗)成员, 而 `_Mypair` 是私有(看到那个 `private` 了吗)成员。这就说明, 我们可以调用 `length` 来返回 `_Mypair._Myval2._Mysize`, 但不能直接用这个私有成员。这样就非常安全, 外界不能任意篡改 `_Mypair`; 而且也非常方便, 我们根本就不需要知道 `_Mypair._Myval2._Mysize` 这个复杂的用法, 只需要调用成员函数 `length` 就够了。

而在调用其它成员函数, 比如赋值运算符时, 这个字符串的长度值也可能发生变化。下面是 `assign21` 函数的实现, 读者只需关注 `_Mypair._Myval2._Mysize=_Count` 这一步, 它就是一个改变长度的过程。

```
public:
    _CONSTEXPR20 basic_string& assign(
        _In_reads_(Count) const _Elem* const _Ptr, _CRT_GUARDOVERFLOW
        const size_type Count) {
    if (Count <= _Mypair._Myval2._Myres) {
        //...
        _Mypair._Myval2._Mysize = Count; // 在这里改变了长度值
        //...
    }
}
```

## 类的成员函数

以往我们的函数都是定义在全局范围内的。C++ 允许我们在类中定义函数, 这样的函数叫作**成员函数 (Member function)**。上文讲到的 `length` 就是 `string` 类的一个成员函数。

成员函数的特点有二: 其一, 成员函数只能由这个类的对象来调用; 其二, 成员函数可以访问这个类的 `protected`/`private` 成员。

假如说我们想要模仿 `string` 类的思路来写一个简易版本的 `String` 类, 我们可以这么写:

```
class String { //String类的定义
public: //从这里开始的都是公有成员
    unsigned length() { //返回值表示str的长度
        return _length; //String类的成员可以访问私有成员_length
    }
    void assign(const char *src) { //改变str字符串中的内容
        strncpy(_str, src, 100); //String类的成员可以访问私有成员_str
        _length = strlen(_str); //更新_length的值
    }
private:
    char _str[100]; //实际的string使用指针和动态内存分配, 这里简化成用char[100]
    unsigned _length; //私有成员_length, 用来记录现在str的长度
};
```

<sup>21</sup>实际调用赋值运算符时都要经过 `assign` 这一步, 我们就不多说了。

在这里我们定义了 `String` 类的部分功能，包括两个公有成员函数和两个私有成员变量。接下来我们解释一下它们的作用。

`_length` 私有变量用于记录现在 `_str` 中存储的有效内容的长度，这个不难理解。而 `length()` 函数用于返回这个值。这样我们定义的任何 `String` 类变量都可以通过 `length()` 成员函数来返回它的值。

```
String str1; //这里不能再用结构体那种初始化语法了，我们之后再谈
cout << str1.length(); //用成员访问运算符.访问length()成员函数
```

`_str` 是一个数组，我们可以用它来存储一串字符。但 `_str` 是一个私有成员，我们不能直接修改它，因此 `assign` 公有成员函数就派上用场了。

```
String str2; //再定义一个str2
str2.assign("BjarneStroustrup"); //由str2调用assign
```

在 `str2` 调用 `assign` 的时候，程序会把 `"BjarneStroustrup"` 作为 `const char*` 数据传给 `assign` 函数<sup>22</sup>。在函数调用期间，`strncpy` 会改变私有成员 `_str` 的内容，而 `strlen` 会计算 `_str` 的有效长度并以此改变 `_length`。

初学者，尤其是从 C 语言过渡到 C++ 的学习者，往往会觉得 `class` 这种设置访问权限的写法实在是太麻烦了，还不如 `struct` 那样一切默认用 `public` 的方法好。这种想法对于小规模的工程来说还好，但是对于大规模的工程来说，如果没有成员函数加以封装，没有私有访问权限加以保护，那么我们很容易在复杂的工程中写很多冗余代码，或者是犯一些低级错误。这就像是我们用函数——我们当然可以不用函数，但用了函数能让我们生活更轻松；或者像是我们用 `const`——我们当然可以不用 `const`，只要你能确保自己写代码时不犯误修改某个值的错误。但是我们还是会选用它们，也会选择用 `class` 代替 `struct`，道理也是一样的。等到读者有了更多的编码经验之后，想必也会体会到这一点。

## vector 简介与示例

C++ 中有一个动态数组类模版 `vector`，它定义在头文件 `vector` 中，是 STL 的一部分。我们暂时不讲 STL，但可以以它为例，讲一下类的成员函数的应用。它是一个类模版，我们需要指定具体的类型，比如 `int` 或者 `double`，从而让它变为一个类实例。为了避免过多无关信息干扰读者的思路，我们直接这么写吧：

```
#include <vector> //包含头文件vector
using vecint = vector<int>; //vecint相当于vector<int>类的别名
```

接下来我们可以直接用 `vecint` 作为 `vector<int>` 类的别名，所以 `vecint` 就成了一个类型。这样理解就好。

`vecint` 有一个成员函数 `push_back`，它允许我们每次向这个动态数组中添加一个元素；还有一个成员函数 `size`，它的返回值是当前数组的有效数字个数。

```
vecint v; //定义一个v，这时它的有效数字个数为0
for (int i = 0; i < 10; i++) {
    v.push_back(i); //每次向v中添加数据
    std::cout << v.size() << '\u00d7'; //每次输出v的有效数字个数
}
```

这个程序的输出如下：

---

<sup>22</sup>实际被传递的还有 `&str2`，作为隐藏参数 `this` 传递。详见第八章。

---

```
1 2 3 4 5 6 7 8 9 10
```

---

不难看出，这个动态数组的数据量是在变化的。

`vecint`有一套动态调整容量的机制，如果要存储的数据量超过了原有的容量（比如说，用 `push_back` 向已满的数组中添加新数据），它就会重新分配一块更大的内存空间。我们可以用 `capacity` 函数来输出 `v` 的容量。

```
vecint v; // 定义一个v，这时它的容量为0
for (int i = 0; i < 20; i++) {
    v.push_back(i); // 添加数据，如果数据量超过了容量，v将会自动扩容
    std::cout << v.capacity() << '\u200e'; // 输出v的当前容量。
}
```

这个程序的输出如下：

---

```
1 2 3 4 6 6 9 9 9 13 13 13 13 19 19 19 19 19 19 28
```

---

也不难看出，随着我们不断向 `v` 中添加数据，它的容量会以一种特殊的方式来扩充——不是每次加一（重新分配内存会很浪费时间）。

如果我们预先知道我们需要多少容量，我们也可以用 `reserve` 函数一次性分配这么大的容量。  
23

```
vecint v; // 定义一个v，这时它的容量为0
v.reserve(50); // 用reserve改变其容量
for (int i = 0; i < 50; i++)
    cin >> v.at(i); // 向数组中输入值
```

这里的 `v.at(i)` 有点像是数组下标访问的形式，它也是从 0 开始的，所以 `v.at(i)` 对应的是数组的第 `i+1` 个数。实际上 `vecint` 重载了下标运算符，它使得我们可以用更简单的方式来表达 `v.at(i)`。

```
for (int i = 0; i < 50; i++)
    cin >> v[i]; // 直接用下标来访问，就像真的数组一样
```

这些功能看上去让人眼花缭乱，但它们的思路都不难想，我们有了一定的基础之后也可以自己写一个低配版的 `vecint` 出来。在后面的章节中我们会实操一些相关方面的内容，比如实现 `vector`, `valarray`, `string` 和 `stack` 等——看上去很吓人，但实际上没有那么难。当读者把这些代码的构建全部练过一遍之后，读者对 C++ 的理解想必会更上一层楼。

不过莫急，我们先插叙一章，来讲讲代码工程的相关知识。

34

---

<sup>23</sup>`reserve` 的作用是，如果现在的容量小于 `reserve` 的参数中给定的值，就把它容量扩充到这个值；否则就什么也不做。

# 第七章 代码工程

当读者已经读到这里的时候，以你的知识，其实已经足够实现很多功能了。将来我们的代码会写得越来越长，花样和技巧也会越来越多，功能的多样性和复杂程度也会逐渐攀升。

在开始下一步的内容之前，我深感有必要向读者传授一些基本的工程知识，这样我们才能够更加清晰、简洁、有效地管理我们的代码，提高生产力，并且不致在许多恼人问题上浪费时间。

在本章之后，我也会逐步调整示例代码的风格，从新手友好型的 `using namespace std` 逐渐转向通篇 `std::` 的严谨性风格上去。读者也应当学会适应不同风格的代码，并最好形成自己最习惯的一套风格来。

希望读者能够在学完本章之后掌握一些代码工程的相关知识，这对我们后续的编程大有裨益。

## 7.1 跨文件编译

我们曾谈过，一个源代码的核心是主函数 `main`。为了突出重点，我们更倾向于把主函数的定义放在其它函数的定义之前。这样就会带来“找不到函数”的问题，因为编译器在遇到一个函数名的时候，它会先向前寻找。

为了解决这个问题，我们需要在函数的定义之前先声明。如果有默认值的话，也尽量放在函数声明的时候写出来，否则编译器也找不到它。

我们后来又学习了结构体、共用体和类。它们也遵循这样的规则，我们必须在使用它之前就给出声明。

```
class Data; //Data类的声明
void func(Data); //func(Data)的声明
//...
class Data {
    //...
}; //Data的定义
void func(Data d) {
    //...
} //func(Data)的定义
```

其中 `Data` 的声明和 `func(Data)` 的声明是不能颠倒的，否则编译器就找不到 `Data` 了。

麻烦在于，一旦我们的程序稍微复杂点——试想，我们为了实现某个功能，需要三个类和十余个函数——我们仍然需要写一大串的定义。这样还是会把 `main` 的定义挤到后面去。整个文件也会极度臃肿，既不方便写，也不方便看。

C++ 允许，也提倡我们把同一段长代码放在不同文件中，然后再通过编译器、链接器<sup>1</sup>等，生成一个可执行文件。读者可以回顾图 1.1，作为参考。

想想我们在每编译一个完整的代码时，是不是都要写一句这个呢：

<sup>1</sup>实际上，很多集成开发环境可以一步到位，把编译、链接等操作一键包揽，这样就能省却我们不少烦心事。

```
#include <iostream>
```

这里的 `iostream` 就对应着一个头文件<sup>2</sup>。`#include` 是一个预处理指令，它会在编译之前把 `iostream` 库中的内容复制到这个源代码中，所以我们才可以使用这个文件中声明的 `cin`, `cout` 等对象。

我们也可以自己写头文件。这样，无论我们需要多少声明（及定义），只需要把它们都放在头文件中，然后 `#include` 就足够了。

在大规模项目中，仅仅用一个头文件可能还是会把代码写得十分冗长，所以我们可以有选择地把某一类功能放在某一个头文件中，另一些功能放在另外的头文件中。C++ 就是这么做的。它给我们内置了许多头文件。当我们需要用到数学函数时，就会使用 `cmath` 库；当我们需要使用某个 STL 容器，比如 `vector` 时，就会使用 `vector` 库。诸如此类。

把定义全都写到头文件中也未必合适。定义的代码可能会非常长，如果我们有单独的代码文件来存储它的定义，而仅仅把声明放在头文件中，这样就会让头文件看上去更整洁——头文件中定义了什么函数，一目了然。所以我们通常用另外的源代码文件<sup>3</sup> 来实现头文件中函数的定义。

就以 `clear_input` 函数为例，我们可以写一个头文件，用来存储 `clear_input` 的声明；写一个源代码文件（要包含头文件），用来存储 `clear_input` 的定义；再写一个源代码文件（要包含头文件），其中有 `main` 函数，用来调用 `clear_input` 函数。

代码 7.1: Header.h

```
#pragma once
#include <iostream>
using namespace std; // 使用命名空间 std
void input_clear(istream& = {cin}); // 在声明中就给出默认参数
```

其中的 `#pragma once` 是另一种预处理指令，它保证这个头文件只被包含一次，而不致违反单一定义规则<sup>4</sup>。较现代的编译器一般支持这种语法；但如果您的编译器不支持 `#pragma once`，您也可以这么写：

```
#ifndef _Header_ // 名字随便起，但要防止撞其它的名字
#define _Header_
//... 这里是头文件的代码部分
#endif // 与#ifndef 配对
```

一个头文件也可以包含别的头文件，比如这里的 `Header.h` 包含了 `iostream`。接下来我们用一个源代码文件来存储 `input_clear` 的定义部分。

代码 7.2: Definition.cpp

```
#include "Header.h" // 它需要包含头文件 Header.h
void input_clear(istream& in) { // 声明部分已经有默认参数 cin，这里不应再写
    in.clear();
    while (in.get() != '\n')
        continue;
}
```

我们在这里使用了 `#include "Header.h"`。其实用尖括号 `<>` 也是可以的，不过对于我们自定义的头文件来说，用双引号要更好一些<sup>5</sup>。

<sup>2</sup> 在 Windows 中，一般是 `.h` 文件。

<sup>3</sup> 在 Windows 中，一般是 `.cpp` 文件。

<sup>4</sup> 单一定义规则（One definition rule, ODR），即任何变量、函数和自定义类型（包括枚举）在每个翻译单元中都可以有多个声明，但只能有一个定义。

<sup>5</sup> 如果我们用尖括号，编译器会先在标准库中寻找，如果找不到对应的头文件，再寻找自定义头文件；而如果用双引号，编译器会直接从自定义头文件开始寻找，这样就节省了编译的时间。

在 `Definition.cpp` 文件中, `#include` 指令会将对应头文件的代码拷贝到此源代码文件中。所以相当于我们在源代码文件中先声明了 `input_clear(istream& ={cin})`, 然后给了它定义。所以在定义的时候我们就不需要再设默认值了<sup>6</sup>。

接下来我们就可以在其它的源代码文件中使用它了。

代码 7.3: `Test.cpp`

```
#include "Header.h"
int main() {
    int n {1};
    while (n != 0) {
        cin >> n;
        if(!cin) //检测cin的状态
            input_clear(); //清理输入
        //...
    }
    return 0;
}
```

我们注意到,当包含了 `Header.h` 之后,在 `Test.cpp` 中就不需要包含 `iostream` 和 `using namespace std` 了。这同样是因为 `#include` 指令会把 `Header.h` 中的代码拷贝到这里, 包括 `Header.h` 中的各种文件包含与命名空间使用。

在这里我们没有给出 `input_clear` 的定义, 但程序仍然知道这个函数要做什么——这正是链接器的功劳。在编译之后, `Test.cpp` 和 `Definition.cpp` 分别会生成目标文件, 链接器将它们链接起来之后, 这个程序就完成了。

我们在上面所说的结构是一种最简单的跨文件编译例子。实际的结构可能更加复杂 (如图 7.1 所示), 但读者只要把握这几条就足够了:

- 一个头文件可以被其它的头文件或源文件包含。但是, 在任何时候, 我们都不应当在别的代码中包含其它源文件<sup>7</sup>。
- 当使用 `#include` 进行文件包含的时候, 编译器会把被包含的文件的内容拷贝到此处。效果等同于我们自己写了一遍同样的代码。
- 任何具有外部链接<sup>8</sup>的东西, 都必须提供一个且只能提供一个定义 (除非它是内联函数, 我们会在后文中提及)。
- 这些代码在编译之后会生成目标文件, 编译器会将它们链接起来。一般的 IDE 都会在编译之后帮助我们完成这些工作, 所以无需过分操心。

在实际的编程中, 我们提倡把下面的内容放在头文件中:

- 常量表达式及常量的定义
- 结构体和类的定义
- 全局函数和成员函数的声明
- 函数模版和类模版的定义<sup>9</sup>

<sup>6</sup>其实是不允许重复设默认值, 哪怕是相同的默认值。

<sup>7</sup>虽然这样是可以的, 但是这样做很没意义, 也容易引起误解。

<sup>8</sup>我们稍后就会讲到链接方式的问题。

<sup>9</sup>类模版比较特殊, 类模版的成员最好定义在头文件中。如果你觉得内容太多, 那就把它们组织到不同的头文件中。

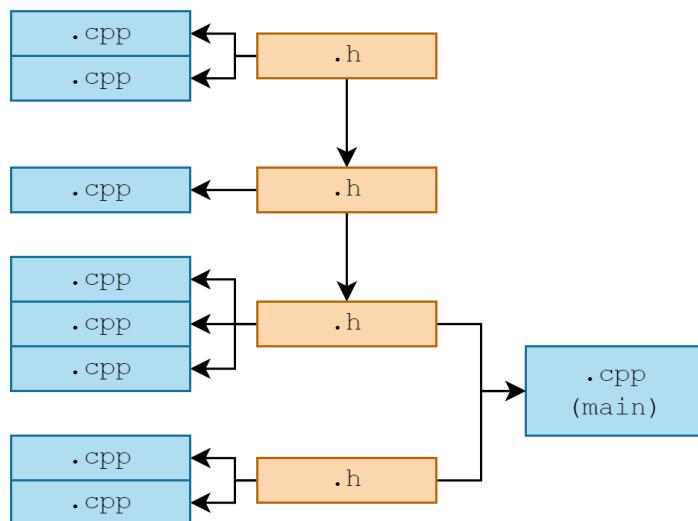


图 7.1: 一个跨文件编译的示意图  
箭头指向的方向有 `#include` 指令

- 一些很短的函数（我们可以以内联<sup>10</sup>形式直接定义在头文件中）

至于函数的定义，我们可以，而且最好把它们放在其它的源文件中。而 `main` 函数存放在一个单独的源文件中，这样我们就只需要在写好其它函数之后，关注主函数的功能即可。以 6.3 实操部分的代码为例，我们可以把它拆成这样三个文件：

代码 7.4: Header.h

```
// 本文件用于存放结构体的定义与函数的声明
#pragma once
struct Data { // Data 仍需放在前面
    int num;
    Data* next;
};
void clear_list(Data*); // 函数声明的时候可以不注参数名
void insert_after(Data*, int);
bool delete_after(Data*); // 重载
bool delete_after(Data*, Data*); // 重载
void transfer(Data*, Data*, Data*); // 这个函数需要用到 utility 库，但不是在这
```

代码 7.5: Definition.cpp

```
// 本文件用于存放 Header.h 中各函数的定义
#include "Header.h"
#include <utility> // transfer 函数用到了 utility 库中的 swap 函数
void clear_list(Data *head) { // 递归回收 *head 之后的所有动态内存
    if (head->next == nullptr)
        return;
    clear_list(head->next);
    delete head->next;
```

<sup>10</sup> 内联 (Inline)，即用 `inline` 把某个标识符声明为内联。对于函数来说，内联意味着某个具有外部链接的函数在不同翻译单元中都可以有一个定义。编译器会对它们进行优化，我们无需过分纠结。

```

    head->next = nullptr;
}

void insert_after(Data *head, int n) { //在*head下一位置插入n
    head->next = new Data{ n,head->next }; //三步并作一步
}

bool delete_after(Data *head) {
//删除*head下一位置的数据，并返回true；若*head是末尾元素，删除失败，返回false
    if (head->next == nullptr)
        return false;
    Data *p = head->next;
    head->next = p->next;
    delete p;
}

bool delete_after(Data *head, Data* tail) {
//删除*head之后（不包含）直到*tail为止（包含）的单元。若删除失败，返回false
    if (head == tail)
        return false;
    if (tail == nullptr) {
        clear_list(head);
        return true;
    }
    Data tmp_head {*head};
    head->next = tail->next;
    tail->next = nullptr;
    clear_list(&tmp_head);
    return true;
}

void transfer(Data *head, Data *tail, Data *dest) {
//片段转移，把*head之后（不包含）直到*tail为止（包含）的单元移到*dest之后
    if (head == tail || dest == nullptr)
        return;
    if (tail == nullptr) {
        for (tail = head; tail->next != nullptr; tail = tail->next) {
            ;
        }
        if (head == tail)
            return;
    }
    std::swap(tail->next, dest->next); //注意要用std::swap
    std::swap(head->next, dest->next); //除非using namespace std
}

```

代码 7.6: main.cpp

```

//主函数定义在本文件中
#include <iostream>
#include "Header.h"
using namespace std;

```

```

int main() { //仅作为示例
    Data head {0,nullptr}, *tail &head;
    while (cin) {
        int n;
        cin >> n; //输入n的值，直到出现异常输入（如q）
        insert_after(tail, n);
        tail = tail->next; //更新tail的指向
    }
    clear_list(&head);
    return 0;
}

```

请读者注意各文件之间使用 `#include` 的关系。因为在 `Definition.cpp` 中没有使用 `using namespace std`，所以我们在定义 `transfer` 函数时，调用 `swap` 函数必须要用 `std::swap` 这样的语法。

读者可能会好奇，我们不是在 `Main.cpp` 中用过了 `using namespace std` 了吗？不然，因为 `Definition.cpp` 和 `Main.cpp` 之间没有包含关系，所以 `Main.cpp` 中写下的代码不能用在 `Definition.cpp` 中，反之亦然。

## 7.2 命名空间

C++ 为我们提供了丰富的函数、类和对象，我们只需要若干 `#include` 指令就可以把我们需要的功能收入囊中。这很方便，但也可能导致一些问题。笔者有一次在调尝试调用自己写的 `swap` 函数：

```

void swap(int &a, double &b) {
    int tmp {a};
    a = b;
    b = tmp;
}
int main() {
    int a {3};
    double b {5};
    swap(a, b);
    cout << a << ' ' << b; //预期输出5 3
}

```

这段代码的调试结果合乎预期，而当我把 `double b{5}` 改成 `int b{5}` 的时候，却也输出了同样的内容。

但是不应该啊，按理说 `int` 类型的 `b` 会转换成 `double` 类型的临时变量，但这个临时变量是右值<sup>11</sup>，不可能被 `swap` 接收。为什么这个函数仍然正常运行并且给出了“正确”的输出呢？

再深入研究之后我才发现，原来程序根本就没有调用我自定义的 `swap` 函数，它调用的是标准库自带的 `swap`！

```

template< class T >
void swap( T& a, T& b ) noexcept( /* ... */ );

```

也就是从这个时期起，我逐渐改掉了使用 `using namespace std` 和 `#include<bits/stdc++.h>`<sup>12</sup> 这两个习惯。

<sup>11</sup> 我们将在精讲篇谈左值/右值的问题。

<sup>12</sup> `bits/stdc++.h` 是 GCC 编译器中允许的一个不标准的头文件。使用这个头文件可以让我们一次性包含许多常见头文件。这种用法被普遍认为是“不好的习惯”，详见 [Why should I not #include <bits/stdc++.h>? - Stack Overflow](#)。

几乎所有初学者都会无一例外地认为 `using namespace std` 是很方便的；而通篇累牍的 `std::cin` 和 `std::cout` 则显得十分麻烦。但是当我们的代码量非常大之后，我们难免会起一些与标准库中名字相冲突的变量、函数、枚举、结构和类。举几个例子吧：

- `list, map, array, queue, set, string, pair;`
- `copy, find, move, search, count, sample;`
- `next, begin, end, data, size;`
- `function, future, thread, yield;`
- .....

以上这些，有些是类模版，有些是函数模版，或者还有别的。这些名字你可能未必都知道，但是总会有些单词是你熟悉的吧！一旦我们不慎使用了这些名字，那么要么程序报错，你找了半天才知道问题所在；要么程序不报错，但至于究竟使用了哪个名字，做了什么，就不敢保证了。

试想，当你在用这些名字定义函数的时候，你是在重载它们；而当你在用这些名字定义类的时候，你会被编译器禁止（因为类不能重载）。被编译器禁止还是更好的选择，因为它在编译期就能帮你检查出错误来；但如果把这些错误留到运行期，那就很难说会发生什么了！

解决这个问题的方法很多样。我们可以自己有意识地避开这些名字（一般来说，只要你经验丰富，就不太容易在这个问题上翻车）；或者干脆点，不用 `using namespace std` 了<sup>13</sup>。那么究竟什么是 `namespace`，我们为什么要用它，又怎么用它呢？本节就来讲解这些问题。

## 什么是命名空间？

想象一下 Windows 电脑的文件资源管理器——就以 D 盘为例吧。我们可以直接把文件存放在 D 盘的主目录下，不过我们很少这么做；我们绝大多数时候是在 D 盘中建立一些文件夹，甚至是文件夹套文件夹，然后把具体的文件存放到这些文件夹中。简化起见，我们只在 D 盘中用两个文件夹。这两个文件夹中各有一份名叫 `Test.txt` 的文本文档<sup>14</sup>。它们虽然同名，但它们不冲突，因为它们在不同的文件夹下。修改其中一个 `Test.txt` 中的内容也不会对另一个 `Test.txt` 造成什么影响。

那么我们是不是可以这样认为：这两个文件虽然重名，但它们存在于不同的空间中，所以不会有冲突。如果我们想访问这两个文件，我们也肯定会通过不同路径来找。这样，一切可能的冲突都解决了。

命名空间也是这样的道理。在不同的命名空间中，我们可以按我们的需要来给变量、函数和类起名字，而不必担心与其它命名空间中的什么东西重名。

不过，实际上的命名空间还是与 Windows 资源管理器的结构不同的。接下来我们讲解一些有关命名空间的最基本规则。

## 命名空间的定义和使用

在所有作用域之外，也就是在函数、类、枚举等内容之外，有一个全局命名空间（**Global namespace**）。在全局命名空间中定义的名字，在其它任何地方都可以使用<sup>15</sup>。比如说，我们声明的那些函数，就在整个翻译单元的所有位置中可用。

<sup>13</sup> 不过我并不建议读者彻底杜绝 `using namespace std` 的使用。这种语法并不像 `bits/stdc++.h` 库或 `goto` 语句那样饱受诟病，仍然是许多程序员图省事的最佳选择。

<sup>14</sup> 它们二者之间不是快捷方式的关系，是真正意义上独立的文件。否则这个讨论就没什么意义了。

<sup>15</sup> 具体说来，指的是此源代码中，从该名称声明处开始的位置。头文件中定义的内容看似在整个源文件中可用，那只是因为我们把文件包含写在源文件开头所致。

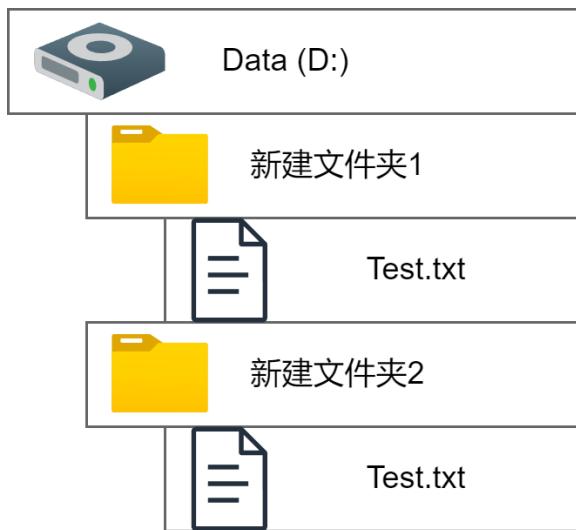


图 7.2: Windows 文件资源管理器中的重名文件

```

struct A {
    //...
}; // 该类定义在全局命名空间中，对后文全局可见
void fun(); // 该函数声明在全局命名空间中，对后文全局可见
constexpr double Pi {3.14}; // 该常量表达式定义在全局命名空间中，对后文全局可见
extern unsigned count; // 该变量声明在全局命名空间中，对后文全局可见
int main() {
    // 这里可以使用 A, fun, Pi, count 等名字，没有任何问题
}

```

我们可以使用 **namespace** 关键字来自定义命名空间。C++ 中使用了 **std** 命名空间，我们所熟知的 **cin**, **cout** 和 **endl** 等等，就在 **std** 命名空间中。

为了使用 **std** 命名空间中的名字，我们可以通过作用域解析运算符<sup>16</sup> **::**（两个紧密排列的冒号，这是“一个”运算符，不可以分开写）。比如说要用 **std** 命名空间中的 **cin** 对象，我们就可以写成 **std::cin**。

```

int main() {
    int number;
    std::cin >> number; // 使用 std 命名空间中的 cin 对象
}

```

我自己写代码时经常用自定义的 **cppHusky** 命名空间，并把我想要写的函数定义在其中。

```

namespace cppHusky {
    template<typename T>
    T max(T a, T b) {
        return a > b ? a : b;
    } // 定义函数模版 max
    template<typename T>
    T min(T a, T b) {
        return a < b ? a : b;
    }
}

```

<sup>16</sup>有些资料会把它叫作“操作符”而不是“运算符”，因为作用域解析更像是一种操作而非运算，其实它们在英文中都叫 Operator，这里可以区分也可以不区分。

```

    } // 定义函数模版min
}; // 注意分号结尾

```

在 `cppHusky` 命名空间中定义的 `max` 就不同于 `algorithm` 库中，在 `std` 命名空间中定义的 `max`。

```

std::cout << std::max(3, 5) << std::endl; // 注意 endl 也在 std 命名空间中
std::cout << cppHusky::max(3, 5) << std::endl;

```

这里我们使用的两个 `max` 函数分别位于两个命名空间中，互不冲突。

命名空间之间还可以嵌套定义，比如 `std` 命名空间中还嵌套定义了 `ios_base` 命名空间，它们二者的大致关系是

```

namespace std { // 命名空间 std
    // ...
    namespace ios_base { // std 中的命名空间 ios_base
        // ...
    };
}

```

在 `ios_base` 命名空间中有一些 `fmtflags` 常量表达式，比如 `boolalpha`<sup>17</sup>。如何获取到它呢？首先它在 `ios_base` 命名空间中，所以我们要使用 `ios_base::boolalpha`。而倘若我们没有直接 `using` 这个 `std` 命名空间，我们还需要通过 `std` 才能访问到 `ios_base` 命名空间，所以我们要写成 `std::ios_base::boolalpha` 才行。

```

bool judgement {2>3};
std::cout.setf(std::ios_base::boolalpha); // 设置成按布尔型输出
std::cout << judgement; // 将输出 false

```

如果我们需要使用全局命名空间中的名字 `x`，我们可以直接写成 `::x` 的形式。

```

int x {3}; // 在全局命名空间中
int main() {
    int x {5}; // 不在全局命名空间中
    std::cout << x << std::endl; // 将输出 5
    std::cout << ::x << std::endl; // 将输出 3
}

```

这里我们定义了两个 `x`，读者可能会感到困惑。不要担心，我们会在下一节中介绍它。

`namespace` 块有一个特点，它是可以扩展的。

```

// 以下是 Header.h
namespace cppHusky {
extern std::stringstream cin; // 声明，其中的 stringstream 需要 sstream 库
template<typename T>
constexpr const char* Tname(const T &object) noexcept{
    return typeid(object).name();
}
template<typename T>
constexpr const char* Tname() noexcept{
    return typeid(T).name();
}

```

<sup>17</sup> 我们曾讲过，`std::cout` 在输出 `bool` 型数据时默认按整数形式输出。如果我们要按布尔型数据输出，可以用 `cout.setf(ios_base::boolalpha)`，详见 2.2 节。

```

};

//以下是main.cpp
#include "Header.h"
int main() { /*...*/}
namespace cppHusky {
std::stringstream cin {"1024"}; //定义
};

```

我自己写代码的时候，常常会用这里的 `cppHusky::cin` 和 `cppHusky::Tname` 来进行测试。读者尚且不需要知道它们是用来干什么的，只需要关注整个代码的结构就可以了。

我们曾介绍过文件包含的机制。在源文件中，`#include "Header.h"` 指令会被预处理器替换成 `Header.h` 中的内容，所以我们可以认为，头文件中的指令被复制到了源文件中。那么在这里，就出现了两个命名空间的定义：

```

//等效于以下写法：
namespace cppHusky {
//在这里声明了cin并定义了Tname
};

namespace cppHusky {
//在这里定义cin
};

```

在第一次使用 `namespace cppHusky` 的时候，我们是在定义一个新的命名空间；而在第二次使用 `namespace cppHusky` 的时候，因为 `cppHusky` 已经定义，所以这里新增的内容只是对 `cppHusky` 命名空间的内容扩充。在这两段代码编译完毕之后，`cppHusky` 命名空间中就既有了 `Tname` 的定义，又有了 `cin` 的定义。我们还可以在其它地方继续通过 `namespace cppHusky` 添加新的内容。

之所以允许这样做，是因为我们会有“把不同头文件/源文件中定义的名字放在同一空间下”的需求。比如对于 `std` 命名空间来说，C++ 在 `iostream` 头文件中定义了 `std::cin`，在 `cstring` 头文件中定义了 `std::strlen`，在 `algorithm` 头文件中定义了 `std::max` 和 `std::min`，诸如此类……它们处于不同文件中，但只要用 `namespace std` 就可以把它们一同加到这个空间中来，这岂不是很方便？

## using 声明

如果每次都要靠作用域解析运算符 `::` 来访问非全局变量，那就显得太麻烦了。所以初学者一般很喜欢这样写，一劳永逸：

```
using namespace std;
```

这个语句的作用是，把 `std` 整个命名空间引入到此处。于是 `std` 命名空间中的所有类、函数和对象都在此空间中可用。如果我们把 `using namespace` 写在全局作用域中（我们一般这么做），那么这个命名空间中的内容就被引入到全局作用域中。

```

using namespace std; //把 \lstdinline@std@ 命名空间中的内容引入全局作用域
int main() {
    int num[3];
    std::cin >> num[0]; //标准写法
    ::cin >> num[1]; //既然引入了全局作用域，那cin就是全局作用域的一部分了
    cin >> num[2]; //会进行名称查找，进而找到全局作用域中的cin，详见下一节
}

```

当然，我们在上文中已经提到过，`using namespace std` 会把 `std` 中的所有内容都引入全局作用域，而其中恰好有些是容易重名的。

所以我们还可以有选择地指定把个别常用的名字引入全局作用域。这里也需要用到 `using` 语法。

```
using std::cin, std::cout, std::endl;
```

这样我们就可以有选择地把这三个常用且不易重名的对象引入到全局作用域，以后直接用它们即可。

关于要不要用 `using` 来引入命名空间，以及怎么引入的问题，不同的人有不同的见解。大致说来，如果你要追求严谨、准确、无歧义的代码，那么你就要忍受使用诸如 `std::` 这种写法造成的麻烦；如果你要追求简洁、方便的代码，那么你就需要有意识地防范出现名称冲突的问题——总之有利有弊。

## 7.3 作用域、存储期和链接

在前面的章节中我们已经介绍过作用域的概念。我曾说，作用域的存在是为了控制变量的生存期，其实这是很片面的见解。更准确的说法是：作用域（Scope）和存储类说明符（Storage class specifiers）二者共同作用，控制一个标识符<sup>18</sup>的生存期（Storage Duration）和链接方式（Linkage）。本节我们就来详细讲讲相关知识。

### 作用域

作用域像是程序中的一片领地或者范围，作用域之间相互嵌套，则有点像是行政区划。一个外层作用域嵌套了若干内层作用域，它对自身和内层作用域都有管辖权。而内层作用域仅仅能管理自己的一亩三分地，对外界没有任何作用。

最外层的作用域叫作全局作用域（Global scope）。我们在全局作用域中定义的类、函数或变量，对所有内层作用域都是可见的<sup>19</sup>。

当我们声明一个函数时，这个函数就引入了一个函数形参作用域（Function parameter scope）。

```
void func1(int a); //a位于函数形参作用域中
void func2(int a); //这两个a并不相同
```

函数形参作用域会在分号之后结束。所以一旦离开了这个函数，`a` 就变得不可见了。如果是函数定义的话，那么函数形参作用域会延续到函数定义的末尾。

```
constexpr double Pi {3.1415926}; //Pi在全局作用域中
double circ_area(double r) { //在定义之时，函数形参作用域开始
    return Pi * r * r; //函数形参作用域延续到函数定义中；全局作用域中的Pi也可见
}
```

当我们定义一个结构体、共用体或类时，它们就会引入一个类作用域（Class scope）。如果我们在其中定义了别的类或者成员函数，那么它们也会引入新的作用域。

```
class dynamic { //struct引入一个类作用域，这是较外层的作用域
    enum Types {integer,floating,string}; //这个枚举不会引入新的作用域，但有些会
    Types type; //Type成员位于类作用域中
    union Value { //union引入一个类作用域，这是较内层的作用域
        long long vll;
        long double vld;
```

<sup>18</sup>未必是变量，也可以是函数和类等

<sup>19</sup>作用域是针对编译时的概念，这决定了编译器会怎样解释代码。而在运行时，编译器早已解决代码如何解释的问题，因而“作用域”这个概念也就不再需要；而我们讨论作用域时，当然也不需要思考某段代码是如何运行的。

```

    char str[16];
};

//Value类的定义结束，作用域随之结束
Value value;
}; //dynamic类的定义结束，作用域随之结束

```

这是我们在介绍联合体时讲过的例子。这里我们在 `struct dynamic` 内部定义 `union Value` 的写法，就是一种嵌套定义。

定义枚举时，我们可以人为地引入一个枚举作用域（**Enumeration scope**）——这个取决于我们的需求。有的时候我们希望使用的名字太普遍了，容易出现重名问题，这时我们就可以定义一个有作用域枚举（**Scoped enumeration**）。

```

enum class Bluetooth : bool {on,off}; //蓝牙是否打开（枚举基为bool）
enum class Wifi : bool {on,off}; //Wi-fi是否打开
enum struct AirplaneMode : bool {on,off}; //用struct或class，效果相同

```

以上这些都是有作用域枚举。当需要它们时，我们不能像以往那样直接用 `on/off` 了——这些 `on` 和 `off` 未必是同一种类型！为了区分，我们需要使用作用域解析运算符 `::` 来操作它们。

```

struct Device{
    //在这里定义Bluetooth等，所以
    Bluetooth bluetooth = Bluetooth::off; //（出厂设置）
    Wifi mobilehotspot = Wifi::on; //这里都要用到作用域解析运算符
    AirplaneMode airplanemode = AirplaneMode::off;
};

```

在老式数据结构的成员中，这样设置默认值是可以的，而且也很简便。但是在类中，我们一般不推荐这么做，而是建议用构造函数来实现数据的初始化。关于这方面的内容，我们会在第八章中介绍。

命名空间也会引入一个命名空间作用域（**Namespace scope**）。命名空间作用域不像块作用域那样封闭，我们可以使用作用域解析运算符来找到它；或者干脆使用 `using` 把它带到全局作用域中来。第二节已经作过讲解，想必就不用我再多作解释了。

而最经典，也是我们最熟悉的作用域，还是要数块作用域（**Block scope**）了。一对花括号，它们套住若干语句<sup>20</sup>，会引入一个块作用域。在这个作用域中定义的名字，无论是变量、类，还是函数<sup>21</sup>，都只对这个作用域可见。

函数定义时也会带来一个块作用域<sup>22</sup>。还有 `for`, `while`, `if`, `else`, `switch` 等结构会自带一个块作用域。我们在 `for` 中定义的内容，在紧随其后的执行部分可见，但在执行部分结束后就不可见了。

对于 `switch` 语句来说，我们还有一个注意事项——如果在不同的 `case` 下需要定义局部变量的话，那么直接这样写是会出错的：

```

switch(num) {
    case 1:
        int x;
        //...
        break;
    case 2:
        int x; //错误
//error: redeclaration of 'int x'

```

<sup>20</sup>也称为复合语句（Compound statement）。实际上，绝不是所有“由花括号套住”的内容都能算作复合语句，比如统一初始化中所用的语法。

<sup>21</sup>我们可以在函数内部定义 Lambda 表达式，这是一种函数对象。我们会在精讲篇中讲到它。

<sup>22</sup>C++ 标准中有函数作用域（Function scope）这个概念，但它与函数定义关联不太。只有标签（Label）拥有函数作用域，其它“在函数中定义的命字”都是函数形参作用域或块作用域，不是函数作用域。

```
//...
break;
}
```

我们试图使用 `switch` 语句来实现分支，而在某些 `case` 下我们可能有定义局部变量的需要——这时候我们就会发现有问题——编译器给我们的报错信息是“再次定义了 `x`”！

出现这个问题的根本原因在于，`case` 本身只是一个标签，它不会自行引入一个内层作用域。换句话说，如果我们不自行引入一个作用域的话，所有的操作都是在 `switch` 的作用域下进行的，那么出现“再次定义了 `x`”这样的报错信息也就不足为怪了！

解决这个问题的方法也很简单，就是人为地引入一个块作用域：

```
switch(num) {
    case 1:{ //人为地用花括号会引入一个更内层的作用域
        int x;
        //...
        break;
    }
    case 2:{ //这样就不会引起冲突了
        int x;
        //...
        break;
    }
}
```

## 名称查找

C++ 允许我们在不同的作用域中使用相同的名字。毕竟，我们划分命名空间和作用域的一个主要目的正是避免名称冲突。在互不相干的作用域中自然可以使用同样的名字，比如我们定义一些函数时就经常用 `a`, `b` 这样的名字。

```
template<typename T> //这是一个函数模版
T max(T a, T b) { return a > b ? a : b; } //在这个作用域中使用a和b
template<typename T>
T min(T a, T b) { return a < b ? a : b; } //在另一个作用域中也用a和b
```

在这里，编译器绝不会搞混这两组 `a` 和 `b`。

而在互相嵌套的作用域中，我们也可以使用同样的名字。举个例子吧：很多初学者在熟悉了用 `i` 作为单层 `for` 循环的枚举变量之后，就很容易在多重 `for` 循环中不慎把多个变量都用成 `i`。

```
for (int i=0; i < 10; ++i)
    for (int i=0; i < 10; ++i) {
        //某些操作
    }
```

但是这样写并不会被编译器报错，反而可以通过编译，继续运行——但具体的运行结果会如何，那就难说了。

其实这是名称查找（Name lookup）的问题。每当我们使用一个名字的时候，编译器都会从此处开始向上查找这个名字的定义。只有找到了这个名字的定义，编译器才能知道这个名字意味着什么（它是变量、还是函数、还是类？什么类型的/接收什么参数/怎么定义的？）。

关于名称查找的规则，简单来说就是一句话：编译器会从这个作用域开始，一步步向外找，直到全局作用域为止，只要找到了一个，编译器就会认为这个名称就是这样定义的；如果没找到，就会认为这个名字没有定义，于是报错。

那么就以上述的双重 `for` 循环为例，如果我们在“某些操作”中使用了 `i`（比如说，输出它的值），这时编译器会如何解读呢？它会向上寻找，当找到内层 `for` 循环中的 `int i` 时，编译器就认定这就是它要找的 `i` 的定义。那么这样一来，“某些操作”中使用的 `i` 全都是内层 `for` 循环中定义的那个 `i`；而外层 `for` 循环中定义的那个 `i` 对它毫无影响。

那么接下来请读者做一个练习，以此深化对名称查找的认识，来看这段代码：

```
int a; // (1)
int b; // (2)
{
    int c; // (3)
    {
        int b; // (4)
        int c; // (5)
        cout << c; // (6)
    }
    cout << b; // (7)
    cout << a; // (8)
    int a; // (9)
}
```

这段代码是编译无误的，所以请读者放心。接下来请问读者：(6)(7)(8) 三句中所使用的变量 `c`, `b`, `a` 分别对应哪个定义？

先来看 (6)。我们在 (3) 和 (5) 当中都定义了 `c`，但编译器只会选择“它第一个遇到的”定义，所以在寻找过程中当然会先找到 (5) 了。

再来看 (7)。我们在 (2) 和 (4) 当中都定义了 `b`，而且看上去 `b` 离得更近，但是请注意，编译器只会从当前的作用域开始向外找，而当遇到一个内层作用域时就会跳过，所以 (4) 被自动忽略，编译器将选择 (2)。

最后来看 (8)。(9) 显然和 (8) 在同一个作用域中，而 (1) 就要远一层了，所以很多人自然会认为是 (9)。但是不要忘记，这些代码的执行是有顺序的！我们不可能先输出 `a` 再定义 `a`，所以编译器进行名称查找的方向都是向前面的代码去寻找，而不能向后！所以最终的答案是，编译器向前寻找到了 (1)，然后认定它就是这个 `a` 的定义。

这样的名称查找规则也能帮助我们理解“为什么当函数定义写在调用之后时，我们要声明这个函数”。这是因为，编译器在查找这个函数时，只会向前进行查找。所以我们最好把需要定义的那些函数声明在先，这样才能让编译器知道它的存在。

## 对象（变量）的生存期（存储期）

在前面的章节中，我们谈到，当一个作用域结束时，本作用域中定义的对象<sup>23</sup>的存储期也就终止了。这就是一种自动存储期的例子。而有些对象，比如 `std` 命名空间中的 `cin`, `cout` 和 `endl` 等，不仅作用域很宽，而且存储期也很长，不会因为离开哪一个作用域就终止，它们都是静态存储期的。

在 C++ 中，所有的对象都具有以下四种存储期（Storage duration）类型之一：

- 自动存储期（Automatic storage duration）。这种存储期的对象在定义之时开始分配内存空间<sup>24</sup>，

<sup>23</sup>包括变量、自定义类的对象、枚举项、函数对象等。我们已经了解了其中的部分内容。

<sup>24</sup>虽然我们习惯把它叫作静态分配，但是它和下文要提到的“静态存储期”并不是同类概念。

而在作用域结束时就自动释放（不需要手动，这是它与动态存储期的显著区别）。我们定义的局部变量多为自动存储期变量。

- 静态存储期（Static storage duration）。静态存储期的对象在整个程序运行过程中都存在，它会一直保留下来，直到程序结束时释放。<sup>25</sup>
- 动态存储期（Dynamic storage duration）。动态存储期的对象由动态内存分配运算符（如 `new/new[]`<sup>26</sup> 等函数来实现）分配产生，它不受作用域的影响，只有当我们手动回收动态内存空间时，它们才会被销毁；否则可能产生内存泄漏问题。
- 线程存储期（Thread storage duration）。这是 C++11 标准中添加的新内容。本书不打算讲解有关并发的知识，所以就不介绍线程存储期了。

我们在局部作用域内定义的变量都有自动存储期，除非我们用 `static`, `extern` 或 `thread_local` 说明符来指定它的类型。其中 `thread_local` 属于并发相关的知识，本书不介绍；我们稍后就来讲 `static` 和 `extern`。

而在命名空间作用域（如全局作用域，或 `std` 这样的作用域）中定义的对象，都具有静态存储期<sup>27</sup>。`std::cin` 就是一个静态存储期变量，所以我们可以在 `input_clear` 函数中先清理 `std::cin`，然后再在主函数中继续使用它，没有任何问题！

至于动态存储期的对象，我们只能通过动态内存分配来开始，也只有通过动态内存回收才能终止，所以它的存储期也是在运行时动态改变的，未必能事先预知。

```
if (/*...*/) { // 谁能预言这个条件是 true 还是 false 呢
    delete[] arr; // 回收，此时 arr 指向的动态内存存储期结束
    arr = nullptr; // 置空，防止野指针出现
}
```

## 对象（变量）的链接方式

在多文件编译过程中，每个源文件都是一个翻译单元（Translation Unit）。在每个翻译单元中，我们定义了同名的变量，它们是否是“同一个东西”呢？这就涉及到链接（Link）的问题。

链接这个概念解决的是变量重名的问题。对于两个重名变量，编译器会根据链接方式来判断它们是否是同一个东西。C++ 中的变量有三种链接方式<sup>28</sup>：无链接（No linkage）、内部链接（Internal linkage）和外部链接（External linkage）。我们分别简单介绍一下这三者。

无链接的含义是，只要离了这个作用域（或者是重新进入这个作用域），这个名字对应的变量就会被销毁。

```
for (int i = 0; i < 5; i++) {
    int x {0};
    ++x;
    cout << x << ' ';
} // 将输出 1 1 1 1 1
```

这个很好理解，因为每次循环结束，`x` 的生存期都会终止，于是下一次我们定义的 `x` 还是从 0 开始自增一次，进而得到输出 1。也即，在每次循环中出现的 `x` 彼此之间没有关系。

但是当我把它改成内部链接时，情况就不一样了。

<sup>25</sup>静态存储期变量一般存储于图 5.1 中所示的数据段或 BSS 段，而自动存储期变量一般存储于栈段，这也是与它们的存储期有关的。至于相关细节，我们就不多说了。

<sup>26</sup>`new/delete` 的确是我们最常用的动态内存分配方式，但它们不是全部。我们还可以使用 `malloc`。

<sup>27</sup>还可能是线程存储期，但是本书不再深究了。

<sup>28</sup>C++20 起增加了“模块链接”方式，但本书基于的标准是 C++17。

```

for (int i = 0; i < 5; i++) {
    static int x {0}; //在这里加了一个static说明符
    x++;
    cout << x << ' ';
} //将输出1 2 3 4 5

```

我们在这里用到了 **static** 说明符，它起到了两个作用：其一，把 **x** 变成了静态存储期对象；其二，规定了 **x** 是内部链接的——在每次循环中出现的 **x** 彼此是同一个事物！

那么既然每次循环时的 **x** 都是同一个事物，它就应该从 0 一路自增到 5，而不是在每轮循环中都从 0 自增到 1 然后就销毁了。

在循环体中是如此，在函数和类中也是如此。

```

void func() {
    static unsigned count {0}; //定义一个静态变量count，并初始化为0
    ++count; //count自增
    //...
}

```

在这个函数中，我们也把 **count** 定义成一个静态存储期对象。这种静态存储期对象的定义语句只会被执行一次；而在此之后，程序每次运行到这里时，都会直接跳过 **count** 的定义。

那么这个 **count** 可以表示什么呢？试想，每次这个函数调用时，程序都会执行 **++count** 这步（前面的定义被跳过），那么 **count** 的值当然就可以用来表示 **func** 函数的执行次数！

但是也请读者注意：用 **static** 改变 **x** 和 **count** 的存储期及链接，并不会改变它的作用域。我们不能在上文的 **for** 循环之外使用 **x** 这个名字，也不能在 **func** 函数之外使用 **count**。这方面的特例只有静态成员，如果我们定义一个公有静态成员，那么外界就可以获取它的值。对于类也是如此，我们可以在一个类中声明静态成员：

```

class A {
    public:
        static int count; //声明一个静态成员count
    private:
        static int num;
};

```

这里我们也使用 **count** 这个名字，但它与 **func** 属于两个不同的作用域，所以不会产生任何冲突。

静态成员，无论它的访问权限是什么，都只能在类中声明；静态成员的定义需要写在类定义的外部。

```

int A::count {0}; //必须在外部定义
int A::num {}; //即便是私有成员

```

在使用静态成员的时候，我们有两种写法。可以用类名搭配作用域解析运算符来写，即 **A::count**；或者用对象名搭配成员访问运算符来写，即 **a.count**。第一种写法是静态成员所独有的，这是因为，静态成员有单独的存储空间，对于所有对象来说都相同。想想，这不正是内部链接的特点吗——每个对象的 **count** 成员都是同一个事物！

内部链接和外部链接的区别是，内部链接仅仅说明“在本翻译单元内它们是同样的事物”，那么对于不同翻译单元来说呢？别忘了，我们在第一章中讲过跨文件编译，这里起码有两个源文件，也就是起码两个翻译单元<sup>29</sup>，它们之间是什么关系呢？

<sup>29</sup>头文件只是一种代码复用的形式，它并不会影响什么链接作用。我们完全可以不用头文件，直接用多个源文件来搞定那些功能——只是写起来要有很多重复代码，修改也比较麻烦。

我们在前文中讲过，我们可以把函数的声明写在某个源文件中，而把定义写在另一个源文件中——它们二者就是“同一个事物”。这正是一种外部链接，不同翻译单元中的同名（且同参数列表）函数对应的是同样的事物<sup>30</sup>。

在全局作用域中声明（定义）的对象是默认为外部链接的，我们也可以使用 `extern` 说明符。

```
//1.cpp
extern int a; //这里的extern不可省略
//2.cpp
extern int a {3}; //这里的extern可省略
```

标记为 `extern` 的对象都是静态存储期对象；但它与 `static` 规定的内部链接不同，`extern` 对象是外部链接的。

**单一定义规则**要求我们，任何一个类/函数/对象都只能有一个定义。既然这里的 `a` 是外部链接的，那么它们当然是同一个事物，也就只能有一个定义。因而，`2.cpp` 中的是定义，而其它源文件中不应该再有定义。但是，`extern` 也不会改变作用域，我们要想在 `1.cpp` 中使用 `a` 这个变量，就必须声明。

这里还有另一个问题——编译器不能确定 `int a` 究竟是声明还是定义<sup>31</sup>。为了明确表示它是一个声明，我们需要用 `extern` 说明符，且不能提供初始化。所以我们要写成 `extern int a`，这样才不会违反单一定义规则。

如果我们希望一个全局对象是内部链接的，那么我们可以在定义语句中使用 `static` 说明符。

```
//1.cpp
static int a {-1}; //内部链接
//2.cpp
static int a {1}; //这两个a是不同的事物
```

在这里，不同源文件中的 `a` 各自具有内部链接；但它们没有外部链接，也就意味着它们是两个不同的事物。（读者可以自己尝试设计一个方案，来分别输出这两个 `a` 的地址，作为挑战题）所以我们给这两个 `a` 分别进行定义，这是不违反单一定义规则的。

## 7.4 编码风格

国际上有一项有趣的比赛，名为国际 C 语言混乱代码大赛（International Obfuscated C Code Contest, IOCCC）。此项比赛的宗旨在于鼓励人们写出最有创意和最让人难以理解的 C 语言代码。图 7.3 就是一个这样的例子（代码部分），它的代码是一个飞行器的形状，而运行结果也是一个飞行模拟器。

这份代码很复杂，而且很难懂，非常合乎“有创意”和“难以理解”的宗旨。而这也正是我们在日常写代码时要规避的，否则无论是写代码时检查逻辑，还是写完代码复盘内容，抑或是交给别人学习，这样难懂的代码都会给人增加诸多不必要的麻烦。

它难以理解的方面主要有以下几点：

- 排版混乱。虽然这个排版很有创意，但这使得我们在阅读时难以理清代码之间的逻辑。
- 命名混乱。这段代码用了各种简短的名字，比如 `L`, `o` 甚至 `_` 这样的名字！命名不清很容易导致人在阅读代码时不知所云。

<sup>30</sup>在翻译单元中，定义于全局作用域的名字，在整个源代码文件中都是可用的，而在其它源代码文件中不可用。正因如此，我们需要在每个翻译单元中提供声明，这样才可以通过编译（编译步骤需在链接步骤之前完成）。

<sup>31</sup>因为定义变量时可以不初始化，直接写成 `int a` 这样。

```

#include <math.h>
#include <sys/time.h>
#include <X11/Xlib.h>
#include <X11/keysym.h>
double L,o,P
,_=dt,T,Z,d=1,d,
s[999],E,h=8,I,
J,K,w[999],M,m,O
,n[999],j=33e-3,i=
1E3,r,t,u,,W,S=
74.5,l=221,X=7.26,
a,B,A=32.2,c,F,H;
int N,q,C,y,p,U;
Window z; char f[52]
; GC k; main(){ Display*e=
XOpenDisplay( ); z=RootWindow(e,0); for (XSetForeground(e,k=XCreateGC (e,z,0,0),BlackPixel(e,0))
; scanf("%lf%lf%lf",y +n,W+y, y+s)+1; y ++); XSelectInput(e,z= XCreateSimpleWindow(e,z,0,400,400,
0,0,WhitePixel(e,0) ),KeyPressMask); for(XMapWindow(e,z); T=sin(O){ struct timeval G={ 0,dt*1e6}
; K= cos(j); N=1e4; M+= H"_"; Z=D*K; F+=_P; r=E*P; W=cos( O); m=K*W; H=K*T; O+=D_._F/ K+d/K*E"_"; B=
sin(j); a=B*T*D-E*W; XClearWindow(e,z); t=T+E+ D*B*W; j+=d_._F*E; P=W*E*B-T*D; for (o+=(I=D*W+E
*T*B,E*d/K *B+v+B/K*F*D)_; p<y; ){ T=p[s]+i; E=c-p[w]; D=n[p]-l; K=D*m-B*T-H*E; if(p [n]+w[ p]+[s
]== 0|K <fabs(W=T*r-I*E +D*P) |fabs(D=t *D+z *T-a *E)> K)N=1e4; else{ q=W/K *4E2+2e2; C= 2E2+4e2/ K
*D; N-1E4&& XDrawLine(e ,z,k,N ,U,q,C; N=q; U=C; ) ++p; } L+=_ (X*t +D*M+m*1); T-X*X+ 1*l*M *M;
XDrawString(e,z,k ,20,380,f,17); D=v/l*15; i+=(B *l-M*r -X*Z)_; for(; XPending(e); u *=CS!=N){
XEvent z; XNextEvent(e ,&z);
++*((N=XLookupKeysym
(&z.xkey,0))-IT?
N-LT? UP-N? & E:&
J:& u: &h); --*(

DN -N? N-D? ?N=-
RT?&u: & W:&h:&J
); } m=15*F/1;
C+=(I=M/ 1,l*H
+I*M+a*X)"_; H
=A*r+v*X-F*1+(

E=. 1+X*4.9/1,t
=T*m/32-I*T/24
)/S; K=F*M+(
h* 1e4/l-(T+
E*5*T*E)/3e2
)/S-X*d-B*A;
a=2.63 /1*d;
X+= d*1-T/S
+(. 19*E +a
+. 64+J/1e3
)-M* v +A*
Z)_.; l +=
K *_; W=d;
sprintf(f,
"%5d %3d"
"%7d",p =1
/1.7,(C=9E3+
0*57.3)%0550,(int)i); d+=T*(. 45-14/1*
X-a*130-J*. 14)_./125e2+F* _*v; P=(T*(47
*I-m* 52+E*94 *D-t*. 38+u*. 21*E) /1e2+W*
179*v)/2312; select(p=0,0,0,0,8G); v-=(
W*F-T*(. 63+m-E*19*D*25-. 11*u
)/107e2)_.; D=cos(o); E=sin(o); } }

```

图 7.3: 飞行模拟器 (1998 年 IOCCC 获奖作品) 的代码

资料来源: Wikipedia

- 语句臃肿。在很多地方，硬生生地把一个复杂功能用单一的语句表达出来，这样就会让代码变得十分难懂。
- 完全没有注释。很多人十分反感在代码中写注释，觉得很麻烦——我自己就是如此。但是有效的注释可以大大增加代码的可读性。很多时候，我们用一句话就可以表达清楚一个函数的作用，比把代码写得清楚还要有用。

那么我们就根据这个“反面教材”，来讲一下如何写一段令人赏心悦目的代码。

### 代码排版

在排版方面，适当的空格、缩进和换行能让代码更易读。空格的作用很明显，它可以将逻辑上不同的事物区分开来（很多时候，仅靠运算符来进行区分还不够清晰）。以下是一个对比：

```
// 无空格
for(i=0;i<10;++i)
    std::cout<<arr[i].member;

// 有空格
for (i = 0; i < 10; ++i)
    std::cout << arr[i].member;
```

注意这里的 `std::cout` 和 `arr[i].member` 之间没有空格，其实也可以有。我的理解是：这里不需要空格，因为它们是用来表示同一个事物的，所以不需要用空格来区分开。读者当然也可以有自己的理解，并逐渐形成自己的风格，只要它易读就好了。

对于缩进和换行来说，不同代码风格的习惯更是迥异——不过大体原则还是相似的，无非是在花括号是否换行、标签是否不缩进等细节上有出入。

斯特劳斯特鲁普在自己的文章<sup>32</sup>中简单介绍了一种 C++ 编码风格，它是对 K&R 风格<sup>33</sup>的变体。本书也推荐读者学习和使用这种“斯特劳斯特鲁普变体”<sup>34</sup>。接下来对它作简要介绍；如果读者希望了解更多，可以去阅读他的文章。

一般我们会对逻辑上有附属关系的语句加一层缩进。比如说 `if` 语句的作用域内，就要加一层缩进。

```
if (a > 0)
    std::cout << "Positive";
else if (a < 0)
    std::cout << "Negative";
else
    std::cout << "Zero";
```

其它诸如 `for`, `while`, `switch` 等结构，以及函数定义、类定义等也是如此。

再比如，在初始化数组的时候，假如初始化内容非常冗长，我们也可以换行加适当缩进。

```
struct Data {
    int num;
    Data *next;
};

Data heads[3] {
```

<sup>32</sup> [PPP Style Guide - Bjarne Stroustrup](#)

<sup>33</sup> K&R 风格（Kernighan & Ritchie Style, K&R style）是一种常见于 C/C++ 等花括号编程语言的风格，其核心特点在于：首尾花括号与引出花括号的语句保持相同缩进，而花括号内的语句增加一层缩进。

<sup>34</sup> 至于其它的常见编码风格，[Brace placement in compound statements - Wikipedia](#) 列表中有所介绍。读者也可以挑选一种合自己心意的来使用。

```

    {0, nullptr},
    {0, nullptr},
    {0, nullptr}
};

```

我们这样定义<sup>35</sup>，肯定比下面这种写法要美观吧——

```
Data heads[3]{{0, nullptr}, {0, nullptr}, {0, nullptr}};


```

这些都是常见的缩进原则<sup>36</sup>。至于换行，适当的换行配合缩进，也能让程序更加整洁。

```

// 写到同一行
if (ch > 'a' && ch < 'z' || ch > 'A' && ch < 'Z' || ch > '0' && ch < '9') {
    //...
}

// 适当换行+缩进
if (ch > 'a' && ch < 'z'
    || ch > 'A' && ch < 'Z'
    || ch > '0' && ch < '9')
{
    //...
}

```

孰优孰劣，就不言而喻了吧。

## 命名

在写一些小的函数时，我们常常会用一些简短的变量/函数或类名称，比如说

```

template <typename T>
T max(T a, T b) {
    return a > b ? a : b;
}

```

但这种习惯最好不要带到大规模函数中，尤其是那种需要数据处理或交互的场合，很容易让人搞不懂一个变量名意味着什么。

函数名和类名尤其应当如此。虽然名字是可以随便起的，但最好还是要做到让人顾名思义。标准库中的 `max` 和 `strlen` 函数，`string` 和 `list` 类等，都有这样顾名思义的优点。

至于具体的命名，可以用简称（比如有人会用 `siz` 代替 `size`，当然这样也可以避免与 `std::size` 撞名，所以也很受常用 `using namespace std` 人士的喜爱）或全名。至于我自己写代码时，一般都使用全名（反正我不用 `using namespace std`）。读者可以选择自己喜欢的命名风格。

## 语句的处理

有些程序员对于“压行”有一种执念，能写在一句（一行）代码中的内容，就尽量写在同一句，而不是把它拆成多句（多行）。为了压行，他们可以使用各种手段，比如把嵌套的选择结构用条件运算符拼凑起来，比如：

<sup>35</sup> 补充：全局变量定义在 Data 段或 BSS 段。如果我们没有提供初始化的话，它会存储于 BSS 段，同时获得默认值。这个默认值对于整型来说是 0，对于指针来说是 `nullptr`，因此我们可以只定义 `heads[3]` 但不提供初始化，效果相同。这里只为讲解而如此。

<sup>36</sup> C/C++ 都是自由格式语言（Free-form language），自由格式语言对空格的要求不太，对缩进和换行则是几乎没有要求；非自由格式语言，如 Python，写起来更简洁，但它对格式要求更严格，比如说缩进直接与逻辑挂钩，所以就不能像 IOCCC 示例那样乱写。

```
return exp1 ? exp2 ? opt1 : opt2 : exp2 ? opt3 : opt4; //甚至可以继续缝
```

这种习惯对于稍大规模的项目来说是不利的，因为稍大规模的项目不可避免地会有很多代码，这时最重要的并不是“行数少”，而是代码逻辑一目了然，容易让人（包括自己）看懂。否则一旦代码写出了问题，自己就要费很大事来从这一句冗长的代码当中抽丝剥茧，这是何苦呢？

所以，除非这段代码的逻辑简单到可以写成 `max` 那样，否则我们还是老老实把代码的逻辑写得清楚一点，这样能免去很多麻烦。

```
if (exp1) {
    if (exp2) {
        opt1;
    }
    else {
        opt2;
    }
}
else {
    if (exp2) {
        opt3;
    }
    else [
        opt4;
    ]
}
```

还有一些时候，我们需要表示一个很长的表达式——比如说一元二次方程的求根公式。

```
#include <utility>
#include <cmath>
using std::sqrt;
//把sqrt引入全局命名空间中，这样我们就不用再写std::sqrt了
using pair = std::pair<double, double>;
//pair是一个类模版，它的对象可以存储两个值，适合本例
//这里使用using之后，我们可以直接用pair来代替std::pair<double,double>
pair quadratic_solve(double a, double b, double c) {
    return pair(
        (-b - sqrt(b * b - 4 * a * c)) / 2 / a,
        (-b + sqrt(b * b - 4 * a * c)) / 2 / a
    ); //这里用到了std::pair的构造函数，我们下一章再讲。
}
```

这里我们就不管命名的问题了，因为我们在数学上就是用  $a, b, c$  的，而且这个函数的功能也没有复杂到需要用多少名字。但仅就表达式的形式来看，它也是非常冗长难看的。

对于这种，乃至更冗长的表达式，我们最好是用中间变量的方式来简化我们的表达。我们在数学中是用中间变量  $\Delta$  来表示根式内的部分，而在编程中我们也可以这样做。

```
pair quadratic_solve(double a, double b, double c) {
    const double Delta {b * b - 4 * a * c}; //常量中间变量Delta
    return pair(
        (-b - sqrt(Delta)) / 2 / a,
```

```
(-b + sqrt(Delta)) / 2 / a  
); // 这样写就显得简洁了一些  
}
```

## 注释

注释的好处，已经不再需要我多作说明了。本书示例代码中的注释详尽之致，想必读者都能感受得到。对于编译器来说，注释部分的内容会被完全忽略，所以注释主要是为了给人看的。

良好的注释对于我们理解代码来说很重要，尤其是过了几天到几个月之后，如果你回看源代码，会感谢你当时写过的注释的。

C++ 中允许我们使用行注释和块注释，我们在前文中的绝大部分注释都是行注释，它的作用范围就是从 `//` 开始到行尾的部分。如果我们要写多行注释的话，那就只能在每行都用一个 `//` 了<sup>37</sup>。

块注释以 `/*` 开始，以 `*/` 结尾，当中的所有部分，无论是否有换行，都会被当成注释内容来对待。所以它更适合我们写大段注释（毕竟，写注释也要适当换行嘛）。

```
// 这是行注释  
/*这  
是  
块  
注  
释*/ int main() { // 块注释结束后，后续内容就不是注释了  
    //...  
}
```

---

<sup>37</sup> 也可以在上一行的注释末尾加反斜线 \，不过我不是很推荐这样做。

# 第八章 类与函数进阶

到第七章结束，读者应当已经掌握了 C++ 语言的必要知识，并具备了独立搭建简单 C++ 工程的能力。因此从本章开始，我也会略微转换讲解风格，不再用长篇累牍的知识点，大水漫灌。我将带领读者，从一些简单的例子开始练习，进而完成一个类，一个类模版，或是一系列函数，从而实现一些功能。

我的目标是，在本章的结尾带领读者写出一个简化版的 `string` 类。C++ 的 `string` 库中已经有这个类，按理说不需要我们实现。但是，“会用”和“会写”还是不一样的！在用的时候，我们对很多陷阱一无所知，对很多问题浑然不觉——这恰恰是封装良好的优点，我们使用某个功能时无需在这些杂碎问题上浪费必要的时间——一旦自己从头开始搭建，这些问题就会纷纷暴露出来。

只有用过了知识，我们才能掌握；只有暴露了问题，我们才能进步。任何一门编程语言都是如此，C++ 也不例外。

## 8.1 运算符重载

我们在前文中已经用函数的方式来操作链表，进而实现一系列功能。在 6.3 节中，我们定义了这些函数：

- `insert_after(Data*, int)` 在某个位置之后插入单个数据。
- `delete_after(Data*)` 删除某个位置之后的单个数据。
- `delete_after(Data*, Data*)` 删除某两个位置之间的数据（左不包含右包含）。
- `transfer(Data*, Data*, Data*)` 进行片段转移，把一个片段之内的数据转移到某个位置之后。
- `clear_list(Data*)` 把某个位置之后的所有动态内存都回收（是一种特殊的删除）。
- 还有一个我们提过但没有写出来的片段插入函数，这个可以通过临时链表再片段转移来实现。

这些函数的名字都很清楚，让人顾名思义，这也是我们在上一章中推荐的写法。但是在用起来的时候呢，就稍微有点麻烦了。

类似的例子比比皆是，如果我们需要拼接两个 `std::string` 对象，那么我们就需要用到 `append` 成员函数。

```
std::string str {"Bjarne"};
str.append("uStroustrup"); // 把字符串 " Stroustrup" 拼接到 str 后面
```

或者是在访问 `std::vector` 对象时用 `at` 成员函数。

```
std::vector<int> arr {1,2,3,4}; // 初始化为长度为 4 的数组
std::cout << arr.at(2); // 输出 3，因为下标是从 0 开始的，所以 2 号是第 3 个数
```

这样并非不可以，而且也不是特别麻烦——总比不用函数要简便吧。但这样多少会给人一种“生分”的感觉——我们自定义的东西更低一等，它不配用数组那种`[]`运算符来访问数据，非得用一个函数`at`才行。

C++ 中允许我们进行运算符重载（Operator overloading），有了运算符重载之后，我们可以用运算符对自定义类型的对象进行某种操作，就像自定义函数一样。

```
str += "\uStroustrup"; // 等效于 str.append(" Stroustrup");
std::cout << arr[2]; // 等效于 std::cout<<arr.at(2);
```

重载运算符的好处不止于“方便”而已。它还为我们后续的泛型编程提供了可能。试想，无论是普通数组`T[N]`，还是`std::vector`，还是`std::array`，或者是`std::valarray`<sup>1</sup>，只要它是可以表示“数组”的类型，那么我们就可以用下标运算符`[]`来访问其中的单个数据。那么我们就可以不管它具体是什么类型，只要都用下标运算符来操作它就行了（这正是泛型的理念）。

```
template<typename T> // T 可以是任何一个数组
void arr_output( // 参数过多，所以换行写
    T arr, // 数组
    std::size_t size, // 数组长度，最好用 std::size_t 类型；用 int 或 unsigned 也可
    const char *delim = '\u' // 间隔符，默认用空格符
    std::ostream &out = std::cout // 默认用 std::cout 来输出
) {
    for (int i = 0; i < size; i++) // 无需担心，int 与 std::size_t 可以比较大小
        out << arr[i] << delim; // 无论 arr 是哪种数组，都可以用使用下标访问
}
```

因为`std::vector`,`std::array`,`std::valarray`都对下标运算符有重载，所以无论`arr`是普通数组，还是上述哪一种数组类型，我们都可以统一使用下标运算符来访问。

那么本节，我就边做边讲，带领读者学习一下运算符重载的基本知识。

## 实操：`std::vector` 语法糖

语法糖是指在编程中使用的某些简便语法，它能让我们写起代码来更方便。引用就是一种语法糖，它的本质是指针。我们实际写代码时就能感受到，使用引用来传参比使用指针要轻松多了。`std::istream`就重载`>>`运算符时，传入的参数就是引用。

```
basic_istream& basic_istream::operator>>(int &value);
// 这里它的若干重载之一，其右操作数为 int& 类型
```

正因如此我们才可以把输入语句直接写成`std::cin>>num`这种传引用的形式。如果要用指针传参的话，我们要写成`std::cin>>&num`传地址才对。

`std::vector`有许多丰富的功能，我们在前文中已经介绍过了。这里我们再为它添加一些语法糖，来简化一些操作，比如`push_back`等。

为了避免类模板的信息干扰读者的思路，我们还是用`using`语句来起一个别名，并且显式实例化一下针对`int`类型的版本。我们可以在全局作用域中这样写：

```
#include <vector> // 头文件
using vecint = std::vector<int>; // 类型别名
```

接下来我们把`vecint`当作一个数组类型就行了，它拥有`std::vector`的一般特征，其中存储的数据都是整数。

---

<sup>1</sup>关于`std::array`和`std::valarray`，我们会在后文中谈到它们。

## 移位运算符的重载

`vecint` 允许我们在数组尾部用 `push_back` 成员函数插入新数据，它的声明形式是

```
void push_back(const T &value);
void push_back(T &&value);
```

第二个重载涉及右值引用，我们暂不讨论，只看第一种即可。

我们可以重载左移位运算符 `<<` 来表示“添加到后面”<sup>2</sup>，这样写就好：

```
vecint& operator<<(vecint&, int); // 声明
vecint& operator<<(vecint &vec, int num) { // 定义，其中的 num 可以是 const
    vec.push_back(num); // 在内部仍然使用 push_back 成员函数来实现这个功能
    return vec; // 返回值是 vec 本身
}
```

接下来我们如果想要向一个数组 `arr` 中添加数据，就不需要写成 `arr.push_back(0)` 了，直接用 `arr<<0` 即可。

运算符重载的语法很像是函数定义<sup>3</sup>，只是名字不再是那种自定义标识符，而是 `operator` 加一个运算符（在这里是 `<<`）的形式。而在其它方面，诸如操作数，返回值，都与函数很相似。

但是，在运算符重载时，参数列表中不能接收任意多个数据。对于一元运算符<sup>4</sup>来说，它必须接收一个参数，不能多不能少；对于二元运算符来说，它必须接收两个参数，不能多不能少。在这里，`<<` 的左操作数是 `vecint&`，右操作数是 `int`。这里的左操作数要传引用，这是因为我们需要修改 `vec` 的内容；而右操作数不需要修改，所以传值就够了<sup>5</sup>。

至于返回值，这里也有那么一点讲究。其实我们完全可以像 `push_back` 本来的定义那样，把返回类型定成 `void`。而在这里，定义成 `vecint&` 类型也能提供一种方便，那就是连续使用：

```
arr << 0 << 1 << 2; // 连续添加三个数
// 等效于下面的写法
arr.push_back(0);
arr.push_back(1);
arr.push_back(2);
```

我们可以用运算符结合性的知识来分析一下。左移位运算符的结合性是从左向右，所以这段代码应该解析为 `((arr<<0)<<1)<<2`。其中 `arr<<0` 的返回值就是对 `arr` 的引用，它又可以作为新的操作数，与 1 作用，然后是与 2 作用。以此类推，还可以进一步连续使用。这个语法很像是连续输出。

## 自增/自减运算符的重载

`vecint` 允许我们使用 `pop_back` 函数来删除数组的最后一个元素。它的声明形式是

```
void pop_back();
```

我们可以重载后缀自减运算符 `--` 来表示“删除最后面的数”。不过这里涉及到一个问题——前缀自增/自减运算符和后缀自增/自减运算符都可以重载，那么怎么区分它们呢？

<sup>2</sup> 我们一般表示某种线性数据结构时，习惯上把首元素放在左端，尾元素放在右端，所以添加到后面就是添加到右侧，这是很形象的。

<sup>3</sup> 在 C++ 中，运算符和函数之间有着密不可分的联系。我们在第二章介绍运算符时就在讲操作数、返回值、副作用，而在函数中也在讲实参、返回值、副作用。它们虽然不完全相同，但确实很相像。后面我们还会遇到函数对象，它进一步打破了函数与运算符之间的壁垒。

<sup>4</sup> 一元运算符（Unary operator），指的是只能接收一个操作数的运算符；下文的二元运算符（Binary operator）指的是能且只能接收两个操作数的运算符。

<sup>5</sup> 一些人会片面地认为，传引用比传值节省内存占用，所以凡是传参必传引用，如果不需要修改就加之以 `const`，这是忽略了“传引用的本质是传指针”而形成的偏见。举个例子，在某些环境中，一个指针需要 8 字节，而一个 `int` 变量只需要 4 字节，这时传引用不就比传值更浪费内存了吗？

C++ 中规定，如果我们要重载前缀自增/自减运算符，那么按照一般形式来写就好。如果我们要重载后缀自增运算符，就需要在参数列表中添加一个不具名的 `int` 形参。这个形参不需要起到什么实质作用，它只是为了区分不同的定义方式。如果我们要重载后缀自减运算符的话，就应该这样写：

```
vecint operator--(vecint&, int); // 声明后缀自减
// 如果声明前缀自减，参数列表应为(vecint&)
vecint operator--(vecint &vec, int) { // 第二个参数没有作用，所以不需要名字
    vec.pop_back(); // 在内部仍然使用pop_back成员函数来实现这个功能
    return vec;
}
```

接下来我们就可以直接用后缀自减来替代 `pop_back` 成员函数了。

```
vecint arr {1,2,3,4};
arr--; // 删除末尾数据
std::cout << arr.back(); // back() 用于输出末尾元素，这里将输出3
```

这里的返回类型是 `vecint`，而不是 `vecint&`，所以我们不能用它来进行连续删除。读者可能会好奇：难道这里的返回值不能设为引用形式？

并非如此。对于重载运算符的返回值，C++ 没有任何限制；但是在习惯上，我不倾向于把后缀自增/自减运算符的返回值设为引用——这是因为，C++ 内置类型的后缀自增/自减本来就不可以连续使用。

```
int num;
++++num; // 允许
num++++; // 不允许
//error: lvalue required as increment operand
```

还记得吗？对于内置类型来说，前缀自增/自减返回的是处理后的结果，而后缀自增/自减返回的是处理前的结果——这个处理前的结果是一个临时变量，不能按引用返回，只能按值返回。虽然我们自己重载的运算符没有这个问题，但是鉴于人们通常不用连续后缀自增/自减，所以我也不这么重载。如果读者喜欢这么用，当然也可以这样重载：

```
vecint& operator--(vecint&, int); // 声明后缀自减，它返回引用
vecint& operator--(vecint &vec, int) {
    vec.pop_back();
    return vec;
}
```

读者可能还有另外一个困惑：既然不允许后缀自减连续使用，那么我们何不重载前缀自减来实现这个功能呢？

当然可以！不过这样就可能会产生理解上的障碍，让人以为——前缀自减运算符是在删除最前面的数据<sup>6</sup>。运算符不同于函数，函数可以用名字来表达完整、明确的信息，比如 `pop_back`。这样写固然比使用运算符要麻烦，但它不会带来理解错误啊！

运算符虽然用起来方便，但它能为程序员表达的信息实在太少，所以程序员对运算符的理解，很大程度上要依赖我们在这门语言中的“约定成俗”。因此呢，我们在重载运算符时，虽然可以想怎么写就怎么写，但还是要尽量让一个运算符的含义看上去容易理解，不易产生歧义才行。

<sup>6</sup>前面的注释中也提过，在线性数据结构中，我们一般把首位元素写在左边，所以前置自减运算符很容易让人想到是在操作最前面的数。

### 解引用运算符的重载

`vecint` 的 `size` 成员函数也很常用，我们也考虑重载一个运算符来表达它。

```
size_type size() const noexcept;
```

这里的 `size_type` 一般是指 `std::size_t` 类型，它是 `vecint` 的“成员别名”。它是最合乎 C++ 标准的“大小”类型，很多关于大小的数据，比如此处的 `vec.size()` 或者是我们熟知的 `sizeof`，其实都是 `std::size_t` 类型的。读者可以用 `std::is_same` 检验之。

```
using std::cout, std::endl;
cout << std::is_same<decltype(sizeof(0)), std::size_t>::value << endl;
cout << std::is_same<vecint::size_type, std::size_t>::value << endl;
```

言归正传。要选取哪个运算符呢？我们有以下考虑因素：

1. 它必须是一个单目运算符。毕竟 `size` 函数不需要接收什么参数，所以这个运算符不再需要别的操作数，只需要一个操作数，也就是“需要求哪个 `vecint` 对象的 `size`”。
2. 它尽量能表达出 `size` 这个意思来。但是很可惜，附录 A 表中没有哪个运算符非常中我们的意，所以我们需要略微调低一下这一条的优先级。
3. 它的本意不要和我们的预期相差太远。否则的话，被误解的几率会成倍增加。比如说，取反运算符 `~` 一般被当作取反或析构来用，很少被用在其它地方，所以它就不适合表示 `size`。

最后思索再三，我选择了取内容运算符 `*`。读者或许会困惑：这难道不是最容易被误会的吗？如果取内容运算符被用来表达 `size`，那么我们要怎么用指针方法来表达数组的每个单元<sup>7</sup>？

其实，指针形式的表达只对普通数组有效。普通数组的数组名本身就可以隐式类型转换为指针，所以我们可以用取内容运算符来访问它。而 `vecint` 的对象不可以转换为指针类型，所以我们自然也没有用 `*` 来访问其数据的道理。此外，STL 为了解决这方面的不统一，引入了“迭代器”<sup>8</sup>的概念，因此取内容都是对迭代器来操作的，而不是对容器。

总之，取内容运算符已经尽可能满足了我们的上述要求，所以我们就选择它好了。

```
std::size_t operator*(const vecint&); // 声明
std::size_t operator*(const vecint &vec) {
    return vec.size();
}
```

这里的 `std::size_t` 也可以换成 `unsigned` 或者 `unsigned long long`，反正它们都可以互相进行类型转换。

然后我们就可以直接使用它了。

```
vecint arr {1,2,3};
std::cout << *arr;
```

注意在定义 `operator*` 时我们用的就是 `const vecint&` 类型的参数了，这是为了避免拷贝内部数据造成时间/空间浪费，并且我们又有防止误修改实参的需要。关于拷贝的问题，我们之后再谈。

<sup>7</sup>对于数组来说，下标表达 `a[i]` 会被解释成 `*(a+i)`。

<sup>8</sup>我们会在第十一章和精讲篇中介绍迭代器的具体细节。

## 重载中的代码重用

C++ 已经为 `vecint` 重载了六种比较运算符，用于进行字典序<sup>9</sup>比较。如果对象 `arr1` 在字典序上应排于对象 `arr2` 之前，那么 `arr1 < arr2` 的返回值就是 `true`。

```
vecint arr1 {1,3,5}, arr2 {3,1,5};
std::cout << (arr1 < arr2); // 输出 1
```

鉴于 C++ 标准中已经实现了这个功能，我们没必要自己实现一遍。我懒得查标准库的代码，所以就让 ChatGPT 为我们提供了一种可能的定义方式：

```
// 此代码仅供观赏，不要把它添加到源代码中，否则就是重复定义了
template<typename T>
bool operator<(const std::vector<T>& a, const std::vector<T>& b) {
    return std::lexicographical_compare(a.begin(), a.end(), b.begin(), b.end());
}
```

其中的 `std::lexicographical_compare` 是一个 `algorithm` 库中的算法，可以用来对两段数据进行字典序比较。这就是一种代码重用的方式，我们只需要套现成的功能来实现我们的目的就足够了。

我们还可以进一步代码重用，并以此来重载其它五个比较运算符。试想，如何用小于号来表示 `a > b` 呢？其实就是 `b < a` 嘛。

```
// 同上，此代码仅供观赏。下同
template<typename T>
bool operator>(const std::vector<T>& a, const std::vector<T>& b) {
    return b < a; // a > b 就是 b < a，这俩一样
}
```

同样的道理，`a <= b` 其实就是 `!(b < a)`；`a >= b` 其实就是 `!(a < b)`；`a != b` 其实就是 `a < b || b < a`；`a == b` 其实就是 `!(a < b || b < a)`。所以剩下四个都可以仿照此理来定义出来，所有的比较都可以靠小于号来定义，不需要重复写 `std::lexicographical_compare` 的代码，很简洁。

```
template<typename T>
bool operator<=(const std::vector<T>& a, const std::vector<T>& b) {
    return !(b < a);
}

template<typename T>
bool operator>=(const std::vector<T>& a, const std::vector<T>& b) {
    return !(a < b);
}

template<typename T>
bool operator!=(const std::vector<T>& a, const std::vector<T>& b) {
    return a < b || b < a;
}

template<typename T>
bool operator==(const std::vector<T>& a, const std::vector<T>& b) {
    return !(a < b || b < a);
}
```

<sup>9</sup>字典序 (Lexicographic order)，是指按照单词首字母顺序在字典中进行排序的方法。在英文字典中，排列单词的顺序是先按照第一个字母以升序排列 (a, b, c, ..., z)；如果第一个字母一样，那么比较第二个、第三个乃至后面的字母。如果比到最后两个单词不一样长，那么把短者排在前。在编程意义上，我们可以人为给定一个字母表及顺序，这样就可以排成字典序了。比如说 (1, 3, 5) 就排在 (3, 1, 5) 之前，这是只比较两者的首“字母”就可以判断出来的。

## 运算符重载的一些规则

讲了一些比较基础的例子之后，我们最后来解读一下运算符重载的常见规则。这些内容摘录并整理自[运算符重载 - cppreference](#):

- 重载运算符接收的参数中，必须至少有一个是自定义类型<sup>10</sup>。如果重载运算符是成员函数，那么本条自动满足。  
——也就是说，我们不能对 `int`, `double` 这些类型再添加新的运算规则。它们是封闭的，同时也是安全的。
- 有些运算符的重载受到限制，它们只能作为成员函数来定义，不能作为非成员函数来定义，包括（复合）赋值运算符、函数调用运算符和下标运算符。  
——到此为止我们都在定义非成员函数，这主要是因为我们不能撬开 `vecint` 的定义来修改它（这也是封装的优点）。在后文中，我们会自己试着写一个简易的 `valarray` 类，这样就可以定义成员函数了。
- 有些运算符是不允许重载的：作用域解析运算符 `::`、成员访问运算符 `.`、成员指针访问运算符 `.*`、条件运算符 `? :`，以及 `sizeof` 等。附录 A 中都有所介绍。  
——毕竟，有些东西还是要保留其本意的，不然就会出现很大的麻烦。C++ 已经给了我们足够的余裕，剩下的运算符都可以无条件重载，或是在有限条件下重载。
- 不能创建新的运算符，也即附录 A 之外的运算符，比如 `$`, `**11`, `<>`。  
——这更容易理解，因为我们是在“重载”运算符，当然就是重新定义已经存在的运算符，而不是自行开发新的。
- 运算符的优先级、结合性和操作数的数量不会发生变化。  
——当然，不然编译器就不知道如何理解代码了。这些属于运算符本来的特征既不会变化，也不应该变化。
- 重载的运算符 `->` 必须要么返回裸指针，要么（按引用或值）返回同样重载了运算符 `->` 的对象。
- 重载的运算符 `&&` 与 `||` 在使用时不会再有短路求值特性。  
——这也好理解，毕竟重载之后它就不一定实现与原来相同的功能了，所以短路求值作为一个“针对原本目的而做的优化”就没有必要，更不应该保留了。

这些规则可能看上去比较乱，读者未必现在就能理解。没关系，等你经验丰富了之后就自然会掌握了。

至于本节中定义的那些语法糖，它们主要是作为教学目的出现的，而且也不是很规范，所以往后我们就不会再用了。如果读者还想试试这些语法糖的效果，可以看一下这段代码：

```
/* 重载加法运算符，从而返回两个数组的和
规则是，新数组的每个元素值都是两数组对应位置的值之和
如果两数组长度不等，则忽略较长的部分 */
vecint operator+(const vecint &a, const vecint &b) {
    vecint v; // 默认为空
    for (int i = 0; i < std::min(*a, *b); i++) {
        v << a[i] + b[i]; // 依次在后端插入 a[i]+b[i] 的值
```

<sup>10</sup>这里的“自定义”类型是指除了指针、数组和内置类型之外的类、枚举等类型，比如 `std::vector`。

<sup>11</sup>在有些情况下，我们可能连用某个一元运算符，比如对于二阶指针写成 `**pp`。但这不意味着我们是在“使用 `**` 运算符”；这其实应该叫作“两次使用 `*` 运算符”！

```

    }
    return v; //作为数组来返回，因为是临时变量，所以不应当按指针/引用返回
}

```

## 8.2 成员函数与友元函数

在上一节中，我们讲了一些非成员函数形式的运算符重载，并见识到了它的方便之处。不过使用非成员函数也存在一些缺点，比如说：

- 非成员函数不属于这个类，所以不能访问这个类的私有成员。如果要进行什么操作，我们只能走曲线，通过那个类的成员函数来访问它。这样的确很有安全性，但其麻烦也是显而易见的。
- 一些运算符只能定义为成员函数，不能作为非成员函数存在，比如下标运算符。

很容易想到的解决方法当然是——把我们想要重载的运算符定义为成员函数。

我们就以 `std::valarray` 为例吧。它是一个类模版，和 `std::vector` 一样近似于数组。它们的区别在于，`std::vector` 更关注数据的存储和动态操作，而对计算性质的支持较少；而 `std::valarray` 则在数据的批量计算方面支持了更丰富的功能，但在数据存储和动态操作方面几乎没有什么功能。下面的例子展示了 `std::valarray` 在批量计算方面的效果，这是原生 `std::vector` 无法做到的。

```

//需要包含 valarray 库
std::valarray<double> arr {-2.57,-0.987,0.57,2.14159};
std::valarray<double> arr1 {arr + 1}; //每个数都增加1
std::valarray<double> arr2 {std::sin(arr)}; //直接求出每个数的正弦值
for (int i = 0; i < arr1.size(); i++) {
    std::cout << arr1[i] << '\u2022'; //输出arr1的每个数，以供验证
}
std::cout << std::endl;
for (int i = 0; i < arr2.size(); i++) {
    std::cout << arr2[i] << '\u2022'; //输出arr2的每个数，以供验证
}
std::cout << std::endl << arr2.sum(); //sum() 成员函数返回arr2所有数的和

```

这段代码的运行结果如下：

---

```

-1.57 0.013 1.57 3.14159
-0.540972 -0.834376 0.539632 0.841472
0.0057561

```

---

看起来运行结果十分符合预期。这也就是 `valarray` 的强大之处。

`std::valarray` 看起来十分好用。我们也可以自行定义一个 `valarri` 类，并配备这些功能。

```

#include <initializer_list>
class valarri {
public:
    valarri(const std::initializer_list<int> &iplist) { //留到下一节再讲
        _size = 0;
        for (int x : iplist)
            _arr[_size++] = x;
}

```

```

    }
    //...待补充
private:
    static const std::size_t MAX_SIZE {100}; //最大容量
    std::size_t _size; //前的数据存储量
    int _arr[MAX_SIZE]; //一个数组用来存放数据
};

```

其中的 `valarri` 成员是一个构造函数，我们会在下一节中介绍构造函数相关的知识。这里读者不必纠结，直接把它抄进 `valarri` 类的定义中就可以了。

接下来我们就看一下，有哪些我们需要的成员函数。

## 成员函数的声明与定义

首先我们需要一个 `size` 成员函数，它是公有的，能让外界知道这个数组存了多少数据。

```

public:
    std::size_t size() { //返回值是数组当前的存储量
        return _size; //返回私有成员 _size
    }

```

原生的 `std::valarray` 不支持 `push_back`, `pop_back` 等操作，所以我们不管。

为了方便地返回这个数组中的最大值，最小值和总和，我们可以定义 `max`, `min` 和 `sum` 成员函数。这里的 `max` 和 `min` 不会与 `std::max` 和 `std::min` 冲突，不仅是因为它们属于不同的命名空间，还因为它们之间有着“成员”与“非成员”的区别。读者会在实际编程中慢慢领悟到这一点的。

```

public:
    int max() { //返回值是数组当前的最大值
        int maximum {_arr[0]}; //先假设arr[0]就是最大值
        for (int i = 1; i < _size; i++)
            if (maximum < _arr[i]) //一旦遇到了一个比maximum更大的
                maximum = _arr[i]; //就把maximum变成这个数
        return maximum;
    }

    int min() { //返回值是数组当前的最小值
        int minimum {_arr[0]}; //先假设arr[0]就是最小值
        for (int i = 1; i < _size; i++)
            if (minimum > _arr[i]) //一旦遇到了一个比minimum更小的
                minimum = _arr[i]; //就把minimum变成这个数
        return minimum;
    }

    int sum() { //返回值是当前数组中有效数据的总和
        int summation {0}; //先置为0
        for (int i = 0; i < _size; i++)
            summation += _arr[i]; //然后把每个数都加到summation上
        return summation;
    }

```

然后我们只需要把这些内容都整理到 `valarri` 的定义中就行了。

但是当我们真的把一大堆成员函数的定义都写在类定义中的时候，我们就会发现这些内容非常冗长，即便我们把排版做得再好，也还是会有重点不突出的感觉。回想一下我们在 7.1 节中讲过的例子，那时我们在一个头文件中声明函数，在一个源文件中定义函数，而又在另一个源文件的主函数中调用这些函数。这种结构多么清晰，多么一目了然啊。

在定义成员函数的时候，我们也可以把声明和定义分开写，成员声明写在类的定义内部，而成员定义写在类的定义外部。所以我们可以把这个类的定义写成这样：

```
// 头文件部分 Header.h
#pragma once
#include <initializer_list>
class valarri {
public:
    valarri(const std::initializer_list<int>&); // 声明构造函数
    std::size_t size() { return _size; } // 这个函数很短，一行就够，直接定义吧
    int max();
    int min();
    int sum(); // 声明 max, min 和 sum
    //... 待补充
private:
    static const std::size_t MAX_SIZE {100}; // 最大容量
    std::size_t _size; // 前的数据存储量
    int _arr[MAX_SIZE]; // 一个数组用来存放数据
}; // 这样就整洁多了吧
```

这个部分的内容可以放到头文件中。

接下来我们在类的外部进行定义，可以把它放到某个源文件中。不过我们不要忘了，我们是在类作用域的外部进行定义，如果我们直接写成

```
int max() { /*...*/ } // 这是在定义 ::max
```

这样编译器就会把它当作是一个非成员函数来看待，这不是我们想要的。为了区分，我们需要用作用域解析运算符来指明：我们是在定义 valarri 类的 max 成员函数。

```
// 源文件部分 Definition.cpp
#include "Header.h"

int valarri::max() { 我们定义的函数是 valarri::max，不是 ::max；而返回类型是 int
    int maximum {_arr[0]};
    for (int i = 1; i < _size; i++)
        if (maximum < _arr[i])
            maximum = _arr[i];
    return maximum;
}

int valarri::min() {
    int minimum {_arr[0]};
    for (int i = 1; i < _size; i++)
        if (minimum > _arr[i])
            minimum = _arr[i];
    return minimum;
}

int valarri::sum() {
```

```

int summation {0};
for (int i = 0; i < _size; i++)
    summation += _arr[i];
return summation;
}

valarri::valarri(const std::initializer_list<int> &ilist) {
    _size = 0;
    for (int x : ilist) {
        _arr[_size] = x;
        ++_size;
    }
} // 定义的顺序无所谓，不用非得按照声明的顺序来

```

这些就是声明的部分，我们可以把它放到一个源文件中，然后就可以在另一个源文件中调用它们了。

## 成员形式的运算符重载

非成员形式的运算符重载有比较多的限制，所以我们就来看看成员形式的运算符重载。

这两种运算符重载没有本质上的不同，它们接收参数的个数都是一样的，它们的优先级、结合性也是一样的。但在语法上，它们还是有区别的。

### 加法运算符的重载

举个例子，如果我们要重载加法运算符，以支持两个 valarri 类对象直接相加，我们应该这样声明和定义：

```

// 声明部分
public:
    valarri operator+(const valarri&); // 重载加法运算符，以进行两数组加法
    valarri operator+(int); // 重载加法运算符，以进行数组与数的加法
// 定义部分，记得#include<algorithm>，因为这里用到std::min了
valarri valarri::operator+(const valarri &a) { // 重载 valarri 成员函数 operator+
    valarri v {}; // 这里好像不能省略花括号？我们到了下一节再讲它
    v._size = std::min(_size, a._size); // 这里有三个_size 成员，需要区分开
    for (int i = 0; i < v._size; i++)
        v._arr[i] = _arr[i] + a._arr[i]; // 这里也有三个_arr 成员，需要区分开
    return v; // 返回值是 v
}
valarri valarri::operator+(int n) { // 重载 valarri 成员函数 operator+
    valarri v {}; // 同样不能省略花括号
    v._size = _size; // 这两个_size 成员不是同一个东西
    for (int i = 0; i < v._size; i++)
        v._arr[i] = _arr[i] + n; // 这两个_arr 成员不是同一个东西
    return v; // 返回值是 v
}

```

读者可能关心以下几个问题，我将逐一说明：

为什么重载的加法运算符只接收了一个参数？不是说，成员函数接收的操作数个数不变吗？

我们在下一节中会讲到，任何一个成员函数都带有一个隐藏参数，即“调用此函数的对象”。以 `size` 成员函数为例，当我们调用它的时候，比如写成 `_arr.size()`。这里看似不需要传递参数；但在汇编代码中我们就可以看出，实际上这个过程把 `_arr` 以指针（或引用）的形式传递过去了。

对于运算符来说也同理。这里的 `valarri+(const valarri &a)` 还有一个隐藏参数，就是调用它<sup>12</sup>的对象本身；另一个参数是这里的 `a`。

```
valarri arr1 {}, arr2 {};
arr1 = arr1 + arr2; //arr1调用了这个函数，arr2就是形参a对应的实参
```

为什么我们有的时候用 `_arr`，有的时候用 `v._arr` 和 `a._arr` 呢？它们分别代表什么？

我们在用某个对象来调用成员函数时，只能把这个对象以“匿名”的方式作为参数传递——比如说吧，`arr1` 调用加法运算符时虽然也作为参数传入，但是它对应的形参是什么？我们不知道它的名字。这就会造成一个问题：如果我们想要访问它的成员，那该怎么办？

常规的解决方法有两种。第一种最直观，那就是压根不需要用成员访问运算符，直接写成 `_arr` 和 `_size`；第二种我们稍后再讲。以这句代码为例：

```
v._size = std::min(_size, a._size);
```

它有三个 `_size` 成员出现。`v._size` 和 `a._size` 分别是 `v` 和 `a` 的成员，这个不会搞混吧。而 `_size` 正是“调用此运算符的对象”的成员。

如果放在 `arr1+arr2` 的情境下，那么 `_size`, `a._size` 和 `v._size` 分别对应 `arr1._size`<sup>13</sup>, `arr2._size` 和 `(arr1+arr2)._size`<sup>14</sup>。图 8.1 展示了它们之间的关系。



图 8.1: `arr1+arr2` 调用过程中的三个对象及其成员  
以红色、橙色、蓝色区分这三个对象的相关信息

其它的算术运算符，比如四则运算和位运算，也可以仿照此法进行重载，这里就不再赘述了。

### 下标运算符的重载

下标运算符的重载很简单，直接写就行了。

```
// 函数定义很短，所以直接定义也行
public:
    int& operator[](int i) { // 如果允许通过下标运算符改变数组的值，那就用int&
        return _arr[i];
    }
```

<sup>12</sup>成员运算符也可以用成员访问的形式来调用，比如 `arr1+arr2` 也可以写成 `arr1.operator+(arr2)`。但是这样调用就比较麻烦了，还不如自己写个函数。

<sup>13</sup>这只是象征性的写法，实际上 `arr1._size` 的写法相当于在 `valarri` 的类作用域外部访问该类的私有成员，这是禁止的。

<sup>14</sup>初学者可能对 `(arr1+arr2)._size` 这种写法感到困惑，但是不要忘记，`arr1+arr2` 的返回值也是一个 `valarri` 类型的对象，所以它当然也有自己的成员咯。

这里的返回类型是 `int&`，所以如果我们修改下标运算符的返回值，那么我们也就直接修改了数组当中的内容。

下标运算符的返回值并不局限于整型，或者可以隐式转换为整型的类型。任何类型都可以作为下标存在。举个例子来说，`std::map` 是定义在头文件 `map` 当中的类模版，它表示的是一种映射关系。对于 `std::map<Key, T>` 来说，我们可以用 `Key` 类的键（或者说，索引）数据来找到对应的 `T` 类数据。`std::map` 重载了下标运算符 `[]`：

```
T& operator[](const Key &key);
T& operator[](Key &&key);
```

这样一来，我们就可以通过下标运算符，方便简单地使用 `Key` 类的数据找到 `T` 类的值。

### this 指针与赋值运算符的重载

C++ 内置类型的变量都支持赋值<sup>15</sup>，这是改变一个数值的最简单、最直接、最常用的方式——难怪高德纳会把赋值语句和指针变量共同视作计算机科学中最有价值的宝藏。那么我们能不能重载赋值运算符，以此实现 `valarri` 类对象间的直接赋值呢？答案当然是肯定的。

赋值运算符的返回值是对左操作数的引用，所以在这里，我们也要返回一个对左操作数的引用。但是当我们真的写起这个函数时就会发现情况好像不太对：

```
// 声明部分
public:
    valarri& operator=(const valarri&);

// 定义部分
valarri& valarri::operator=(const valarri&){
    // 在这里实现赋值的主要部分
    return /*等下，这里要返回什么？*/;
}
```

在这里我们遇到了困难。左操作数是作为匿名形参传入的，我们要怎么表达“调用这个函数的对象”呢？如果表达不了的话，那我们岂不是就无法返回这个对象了吗？在这种情形下，我们就必须用 `this` 了。

`this` 是一个关键字，它代表的含义是一个指针。不过 `this` 不是传统意义上的指针，它只能在非静态成员函数等地方出现。它只能表示一个地址值，不能被修改（像是常量指针）；但我们可以通过对这个地址值取内容，进而修改它指向的对象——它指向的对象正是“调用此函数的对象”。

既然它指向的就是“调用此函数的对象”，那么我们以 `*this` 作为返回值不就好了？

另外，因为 `this` 指向的就是“调用此函数的对象”，那么我们用指针的成员访问运算符，岂不是也可以解决前文中提到的问题？

```
v._arr[i] = this->_arr[i] + a._arr[i];
```

当然了，这种写法还是比较麻烦。所以倘若没有消歧义的需要，我们就不必这样写，直接用 `_arr[i]` 就可以了。

回到赋值运算符的重载。对于赋值运算符来说，我们需要防范一种特殊情况，就是自我赋值。

```
arr1 = arr1; // 这是需要特殊考虑的！
```

如果进行了自我赋值，那就应该什么也不做直接返回 `*this`，否则就是在浪费时间。

```
valarri& valarri::operator=(const valarri &a) { // 其中的a应作为常量引用
    if (&a == this) // 检测a的地址是否与本对象一致
```

<sup>15</sup>常量和常量表达式不能赋值。所以这里所指的变量不包含常量。

```

        return *this; //如果一致，直接返回*this
    //...
}

```

排除了这个特殊情况之后，我们就可以处理一般情况了，也很简单。

```

valarri& valarri::operator=(const valarri &a) { //返回类型和参数均为引用
    if (&a == this) //检测a的地址是否与本对象一致
        return *this; //如果一致，直接返回*this
    _size = a._size; //记得更改_size
    for (int i = 0; i < _size; i++)
        _arr[i] = a._arr[i]; //有效数据范围内的每个数都要赋值，后面的可以不管
    return *this; //返回*this
}

```

我们还可以重载复合赋值运算符，就以加赋值运算符 `+=` 为例吧。在这里我们也进行了代码重用，从而让我们写起函数更简单。

```

//声明部分
public:
    valarri& operator+=(const valarri&); //重载加赋值运算符，可以加之以数组
    valarri& operator+=(int); //重载加赋值运算符，可以加之以数
//定义部分
valarri& valarri::operator+=(const valarri &a) {
    return *this = *this + a; //利用已经重载的赋值运算符和加法运算符
}
valarri& valarri::operator+=(int n) {
    return *this = *this + n; //利用已经重载的赋值运算符和加法运算符
}

```

## 友元函数

在重载加法运算符时，我们只有“数组 + 单个数字”的版本：

```
valarri valarri::operator+(int n);
```

不过，加法运算满足交换律，所以我们也应当有一个“单个数字 + 数组”的版本。但是当我们真的要实施起来的时候，就会发现困难所在：

我们不能把这个运算符重载为成员函数，因为成员函数的第一个参数必须是 `valarri` 类型，但我们期望它是 `int` 类型。

但是当我们试图重载一个非成员函数时，我们又会遇到新的麻烦：

```

valarri operator+(int n, const valarri &a) {
    valarri v {};
    v._size = a._size;
    //error: 'std::size_t valarri::_size' is private within this context
    for (int i = 0; i < v._size; i++)
        v._arr[i] = a._arr[i] + n;
    //error: 'int valarri::_arr [100]' is private within this context
    return v;
}

```

编译器的报错信息含义很明显，我们不能通过非成员函数访问私有成员 `_size` 和 `_arr`。

这里我们有很多解决方法，比如说直接在函数中返回 `a+n`。

```
valarri operator+(int n, const valarri &a) {
    return a+n; // 代码重用
}
```

但我们还是需要一种方法，让我们可以在特定条件下，从外界访问某个类的私有成员。我们的答案是友元<sup>16</sup>。

声明一个友元函数的方法很简单，就是在类当中，用 `friend` 关键字加这个函数，以此表示“该函数可以访问此类的私有/受保护对象”。

```
// 友元声明部分
class valarri {
    //...
    friend valarri operator+(int, const valarri&);
    // 声明此函数为 valarri 的友元函数
};

// 定义部分
valarri operator+(int n, const valarri &a) { // 定义 valarri 的友元函数 operator+
    valarri v {};
    v._size = a._size;
    for (int i = 0; i < v._size; i++)
        v._arr[i] = a._arr[i] + n;
    return v;
} // 一切顺利
```

我们还可以把友元声明和定义放到一起。

```
class valarri {
    //...
    friend valarri operator-(int n, const valarri &a) { // 定义减法友元
        valarri v {};
        v._size = a._size;
        for (int i = 0; i < v._size; i++)
            v._arr[i] = n - a._arr[i];
        return v;
    }
};
```

但是请读者注意，即便友元函数的定义写在 `class` 内部，它也是非成员函数（如果是成员函数的话那就没必要友元了嘛），因此参数列表必须写两个形参（而不是一个形参加一个不具名参数），而且函数体中不能使用 `this`。

### 友元是二者的关系，而非一者的性质

还有一个常见的例子是输入。我们可以用 `std::istream` 类的对象 `cin` 来为一个 `int` 型变量或者 `double` 型变量输入其值，但是我们不能直接输入一个 `valarri` 类的对象。有的时候我们会有这方面的需要，所以我们需要写一个运算符，能够直接实现 `std::cin>>a`。

<sup>16</sup>以笔者的观点看，友元（Friend）这个名字实在容易让人误会。友元是一种单向关系不是双向关系，但 friend 几乎都是表示双向关系的。对于友元函数来说单向双向无所谓，但是对于友元类来说是有所谓的。

让我们考虑一下——要实现这个目的的主要困难在于，其一，我们不能修改标准库代码<sup>17</sup>，所以我们肯定不能把这个功能写成 `std::istream` 的成员函数；其二，因为左操作数是 `std::cin`，不是 `valarri` 类，所以我们也不能把这个功能写成 `valarri` 的成员函数。

所以我们只剩下一条路，那就是非成员函数。这里我们必须要修改 `valarri` 的私有成员，所以我们需要把它定义成 `valari` 的友元函数。

```
// 友元声明部分
class valarri {
    // ...
    friend std::istream& operator>>(std::istream&, valarri&);
    // 需要包含 iostream 头文件
};

// 定义部分
std::istream& operator>>(std::istream& in, valarri& a) {
    a._size = 0; // 因为它是 valarri 的友元，所以可以访问 valarri 的私有成员
    while (in && a._size < a.MAX_SIZE) // 要保证 in 正常且 a._size 不达到最大容量
        in >> a._arr[a._size++]; // 注意 ++ 是后置的，不是前置
    if (!in) // 检测一下 in 的状态，保证其状态正常
        input_clear(in); // 这次就不是用默认参数，而是用 in 了
    return in; // 返回 in，以便连续输入
}
```

需要提醒读者，这个运算符只是 `valarri` 的友元，它不是 `std::istream` 的友元——我们并没有在 `std::istream` 的定义当中声明这个友元。在这里，我们也不需要它是 `std::istream` 的友元，因为我们并没有使用这个类的什么私有成员，我们只用它的公有成员来完成这些操作。

总之，友元并不是某个函数或某个类的性质，它是函数与类、类与类之间的关系。而且这种关系是单向的：A 是 B 的友元，并不意味着 B 是 A 的友元。

### 8.3 构造与析构

在很多时候，每当我们需要定义一个对象时，我们就会给它提供初始化。在 C++ 中，**构造函数（Constructor）**为我们提供了一种方便的初始化方法；而**析构函数（Destructor）**能够在一个对象的生存期结束前对其进行最后处理——尤其是动态内存回收。

STL 中的 `std::vector`, `std::list`, 以及非 STL 的 `std::string`<sup>18</sup>, `std::valarray` 等类模板的内部机制都是靠动态内存分配来实现的，所以我们才可以动态地改变它的大小。在这里，我们也不妨把刚才定义的 `valarri` 也改成用动态内存分配来实现。

#### 构造函数

构造函数是一种特殊的成员函数，可以为一个对象提供初始化信息。每当我们定义一个新的对象时，相应的构造函数就会被调用，从而为这个对象进行初始化。

在 C++ 中，如果我们不自定义一个构造函数的话，编译器会为这个类自动生成一个默认构造函数（Default constructor）。一个平凡的默认构造函数<sup>19</sup>会什么也不做；如果我们希望它做些什么，对成员进行何种初始化，那么我们就需要按照可能的需求，自行定义一些构造函数。

<sup>17</sup>任何时候都不要修改标准库中的代码。——笔者的忠告

<sup>18</sup>`std::string` 虽然有很多与 STL 容器相似的特性，但它不是 STL 的一部分。

<sup>19</sup>由编译器生成的默认构造函数，通常来说是平凡的；但是也有例外，不过我们现在还没碰到，读者亦不必深究。

就以 `valarray` 为例，现在我们需要让它用指针和动态内存分配的方式来管理数据，所以我们可以把上一节中的函数定义改成这样（主要修改的是私有成员部分）：

```
class valarray {
public:
    //... 构造函数和析构函数部分，待补充
    std::size_t size() { return _size; } // 返回当前的存储量大小
    int max();
    int min();
    int sum(); // 声明 max, min 和 sum
    valarray operator+(const valarray&); // 可以与数组相加
    valarray operator+(int); // 可以与数相加
    int& operator[](int i) { return _arr[i]; } // 可以下标形式访问
    valarray& operator=(const valarray&); // 可以赋值
    valarray& operator+=(const valarray&); // 可以加之以数组
    valarray& operator+=(int); // 可以加之以数
    friend valarray operator+(int, const valarray&); // 可以以数加之
    friend std::istream& operator>>(std::istream&, valarray&); // 需 iostream 库
    //... 待补充
private:
    static const std::size_t MAX_SIZE {100000}; // 可以允许的容量会很大
    std::size_t _cap; // 此数组的容量（可变）
    std::size_t _size; // 此数组的存储量
    int *_arr; // 这里改成指针，以便将来使用动态内存分配
}; // private 成员的顺序需要慎重考虑，详见下文
```

接下来我们向其中添加构造函数。

构造函数的名字要与类名保持一致，所以在这里我们要定义成 `valarray`。构造函数不能拥有返回值<sup>20</sup>，所以我们在定义时直接写它的名字，带上参数列表和函数体就行了。

```
// 声明
public:
    valarray(); // 默认构造函数
// 定义
valarray::valarray() {
    _arr = nullptr; // 指针置空是好习惯
    _cap = _size = 0; // 一开始的容量和存储量当然是 0
}
```

当我们定义一个对象，并且不提供初始化，或者是提供了空的初始化时<sup>21</sup>，就会调用默认构造函数。

```
valarray arr1; // 不提供初始化，将调用默认构造函数
valarray arr2 {}; // 提供空初始化，将调用默认构造函数
```

只有当我们没有定义任何构造函数时，编译器才会生成一个默认构造函数（为了不破坏“定义变量时必须调用构造函数”的规则）。只要我们定义了一个构造函数，编译器就不会为我们生成默认构

<sup>20</sup> 后文我们会讲到，构造函数的作用并不是“制造和返回一个新的对象”，这个对象实际上在构造函数调用之前就已经存在，并且作为隐藏参数传入构造函数中。

<sup>21</sup> 未必。如果该类有接收 `std::initializer_list` 的构造函数，那么提供空初始化时将会调用该构造函数，而非默认构造函数。我们会在下文中讲到相关内容。

造函数。这时，如果我们不提供默认构造函数的话，那么就会形成一个默认构造函数的空缺——也就意味着，这时我们必须要提供定义，不能再省略了。

如果读者还有印象的话，应该会记得，我们在上一节中每逢定义一个 `valarri` 对象时，即便不想提供任何初始化，也需要写成 `valarri v{}` 而不能写成 `valarri v`，原因就在于，我们已经定义了一个构造函数，但没有定义默认构造函数，所以我们不能不提供初始化。解决方法也很简单，像刚才一样提供一个默认构造函数就行了。

心细的读者可能会发现另一个问题——我们使用 `valarri v{}` 为什么就行了呢？难道提供空初始化和不提供初始化有什么区别吗？

其实，问题的关键在于，这里的 `valarri v{}` 并不是像 `int i{}` 的空初始化那样；这个初始化实际上传入了一个 `std::initializer_list<int>` 类型的参数！如果不信，你可以试试 `valarri v()` 这种写法，它是不能通过编译的。

这就不得不谈到 `std::initializer_list` 了。它是一个类模版，可以存储一个常量数组(`const T[N]`类型)。我们可以这样验证它的类型<sup>22</sup>：

```
// 包含头文件 initializer_list
auto ilist = {1,2,3}; //auto会自动识别初始化内容的类型，并作为ilist的类型
std::cout << std::is_same< //用std::is_same来判断类型
    decltype(ilist), //ilist的类型在此之前被自动识
    std::initializer_list<int>
> ::value;
```

这里的 `ilist` 是一个常量数组，它的值不能改变，只能被读取。

我们不能用下标运算符来直接读取 `initializer_list` 对象的值，必须用迭代器加循环的方式逐个访问：

```
auto ilist = { 1,2,3 };
for (auto it = std::begin(ilist); it != std::end(ilist); it++) //循环
    std::cout << *it << ' ';
```

这样就会输出

---

1 2 3

---

还有一种更简单的方法是用范围 `for` 循环，请读者参考下面 `valarri::valarri(const std::initializer_list<T> &ilist)` 的定义：

```
// 声明
public:
    valarri(const std::initializer_list<int>&); //接收一个初始化表作为参数
// 定义
valarri::valarri(const std::initializer_list<int> &ilist) {
    _cap = ilist.size(); // 初始化_cap为ilist的大小
    _arr = new int[_cap]; //分配_cap大小的空间，刚好装得下
    _size = 0; // _size从0开始
    for (int x : ilist) // 范围for循环，遍历ilist中的每个数
        _arr[_size++] = x; // 赋值，_size自增
    // 循环结束后_size会与_cap相等
```

<sup>22</sup>这里走了一点曲线，先用 `auto` 来创建一个对象，再去判断这个对象的类型。这是因为花括号嵌套内容在 C++ 中的解释有多种可能，单纯的 `{1,2,3}` 未必是个对象，所以直接用 `decltype({1,2,3})` 是行不通的。

```
}
```

有了构造函数之后，我们就可以这样定义 `valarri` 类的对象了：

```
valarri arr1 ({1,2,3}); // 直接初始化语法，用()内的数组{1,2,3}进行初始化
valarri arr2 {{1,2,3}}; // 统一初始化语法，用{}内的数组{1,2,3}进行初始化
valarri arr3 {1,2,3}; // 列表初始化语法，用数据1,2,3构成的数组进行初始化
valarri arr4 (); // 直接初始化语法，用()内的数组{}进行初始化，其中数组为空
valarri arr5 {{}}; // 统一初始化语法，用{}内的数组{}进行初始化，其中数组为空
valarri arr6 {}; // 列表初始化语法，用没有数据的空数组进行初始化
// 以上初始化全都调用 valarri::valarri(const std::initializer_list<int>&)
valarri arr7; // 没有提供任何初始化，将调用默认构造函数 valarri::valarri()
```

其中 `arr1`, `arr2`, `arr3` 的初始化方式有别，但效果相同；`arr4`, `arr5`, `arr6` 的初始化方式有别，但效果相同；至于 `arr7`，它的效果就完全不同了。接下来我来讲解一下这七个初始化的异同。

`arr1` 和 `arr4` 的初始化语法都使用了圆括号，它的另一种替代语法是用等号。但是读者不要误会，这里的等号和“赋值”没有任何关系，它的真实含义是构造函数。

```
valarri arr1 = {1,2,3}; // 直接初始化语法，用数组{1,2,3}进行初始化
valarri arr4 = {}; // 直接初始化语法，用数组{}进行初始化，其中数组为空
```

在直接初始化的情况下，圆括号内的部分，或者等号后的部分，会作为参数传递给相应的构造函数——在这里当然就是 `valarray::valarray(const std::initializer_list<int>&)`。

在 C++11 以后，我们一般的建议是使用统一初始化语法，也就是把圆括号换成花括号，写成 `arr2` 和 `arr5` 那样的形式。这两个花括号的含义是不一样的，外层的花括号表示这是一个统一初始化，内层的花括号表示这是一个数组，它的元素就是花括号内部的那些。

而 `arr3` 的定义语法叫作列表初始化——我们在定义原始数组的时候就经常这么干：

```
int array[5] {1,2,3,4,5}; // 这里用到的也是列表初始化
```

其中列表初始化中的所有数据都必须是同一类型的。

总之，`arr2` 与 `arr3` 的初始化方式不同，一个是统一初始化，一个是列表初始化，但是它们最后都是调用了同一个构造函数，并且实现效果相同。至于 `arr7`，我们根本就没有提供初始化（所以说初始化写成 `{}` 和完全不写是不同的！）

## “列表初始化”与“统一初始化”间的权衡

`std::string` 类（作为 `std::basic_string<char>` 的别名）有一个构造函数是这样的（其中的 `CharT` 是 `char`, `size_type` 是 `std::size_t`, 下同。至于 `alloc`, 我们不用管，反正都有默认值）：

```
basic_string(
    const CharT *s,
    size_type count,
    const Allocator &alloc = Allocator()
);
```

它的作用是：取 `s` 字符串的前 `count` 个字符，作为此 `std::string` 对象的内容。以下是一个测试例子：

```
std::string str1 {"cppHusky", 3};
std::cout << str1; // 将输出 cpp
```

读者会发现，我们在这里用的是花括号，它是一种统一初始化语法。

还有一个构造函数是这样的：

```
basic_string(
    std::initializer_list<CharT> ilist,
    const Allocator &alloc = Allocator()
);
```

它的作用是：接收一个 `char` 数组，进而把它们拼成本字符串。

```
std::string str2 {'c', 'p', 'p', 'H', 'u', 's', 'k', 'y'};
std::cout << str2; // 将输出 cppHusky
```

我们在这里用的也是花括号，它是一种列表初始化语法。

所以说，统一初始化语法和列表初始化语法都在使用“一个花括号套若干项（可以是零个，一个或多个）”的操作。因此万一它们之间存在冲突，会发生什么呢？这是我们不可回避的问题。

来看一个典型的歧义例子。`std::string` 类有一个构造函数是这样的：

```
basic_string(
    size_type count,
    CharT ch,
    const Allocator &alloc = Allocator()
);
```

它的作用是：以 `count` 个 `ch` 字符相拼接，从而形成本字符串。

```
std::string str3 {33, '='}; // 试图通过统一初始化调用 (size_type, CharT) 版本
std::cout << str3; // 但输出的内容却是 !=
```

我们发现，这次的输出内容并没有按我们预想的那样，输出 33 个 = 来。

原因就在于，编译器把这个 {33, '='} 解读成了列表初始化，于是 33 被隐式类型转换成了 !!!。为了解决这个问题，我们不得不用传统的直接初始化方式。

```
std::string str4 (33, '=');
std::cout << str4; // 将输出 ======
```

这样就合乎预期了。

那么我们在之前定义 `str1` 的时候为什么没有遇到过这个问题呢？这是因为，"cppHusky" 无法隐式类型转换成 `char` 型，所以编译器不会把这里的初始化语法视为列表初始化，而是统一初始化。

总结一下：编译器会优先把使用花括号的初始化方法当作列表初始化，并可以为此进行隐式类型转换；但是，如果不能通过隐式类型转换匹配到任何一个构造函数，那么编译器就会把这种初始化当作是统一初始化。

## 成员的初始化

构造函数的作用是为这个类的对象提供初始化。但是“对象有初始化”并不意味着“这个对象的成员有初始化”。

读者可能会问：我写构造函数的目的不就是为了给它的成员提供初值吗？如果不写构造函数，直接用默认构造函数的话，那么我定义的局部对象的成员就会得到不确定的初值；现在我写了构造函数，让它有了确定的初值，难道这还不叫“初始化”吗？

先别急，我们来回顾一下之前写过的构造函数，比如 `valarri::valarri(const std::initializer_list<int>` 然后我们就会发现，这里的每个成员获得“初值”的方式都是“赋值”，而不是真正意义上的“初始化”！

```

valarri::valarri(const std::initializer_list<int> &iplist) {
    _cap = iplist.size(); // _cap 通过赋值获得“初值”
    _arr = new int[_cap]; // _arr 通过赋值获得“初值”
    _size = 0; // _size 通过赋值获得“初值”
    for (int x : iplist)
        _arr[_size++] = x; // 赋值，但这里改变的不是哪个成员，而是成员指向的内容
    // 循环结束后 _size 变为与 _cap 相等的值，这又是一个“初值”
}

```

如果仔细思考，你就会发现，这些成员全都是在“赋值”，而不是“初始化”。什么叫初始化？初始化就是，当一个对象/成员生存期开始的时候，就已经有了一个我们给定的值。而对于 `_cap` 或 `_arr` 来说，它们生存期的开始要早于我们为它们赋值。那么它们的初值是多少呢？我们并未指定——所以我说，“对象有初始化”并不意味着“这个对象的成员有初始化”。

那么如何初始化它们呢？我们有两种方法：**成员初始化列表（Member initializer list，初值列）** 和 **默认成员初始值（Default member initializer）**。而在实操上，我们可以把默认成员初始值当作是“默认初值列”。接下来的讲解也会顺着这个思路走。

### 初值列

初值列应当写在构造函数的参数列表和函数体之间，以冒号开始，并以逗号隔开。初值可以用圆括号或者花括号套起来，但是不能用等号。

下面的例子是一个新的 `valarri` 构造函数：

```

// 声明
public:
    valarri(int, std::size_t); // 声明部分不要写初值列
// 定义
valarri::valarri(int n, std::size_t size) // 将n重复size次，作为新数组
    : _cap {size}, // 用一个冒号开始值列；每个成员的初始化用逗号隔开
    _arr {new int[_cap]}, // 用 new[] 来初始化
    _size {0}
{
    for (; _size < _cap; _size++) // 通过循环赋值
        _arr[_size] = n; // 赋值，每个元素都赋成n
    // 循环结束后 _size 会与 _cap 相等
}

```

需要提醒读者的是，初值列应该写在定义中，而不是写在声明中。<sup>23</sup>

成员初始化的顺序仅仅取决于成员声明的顺序，至于初值列的顺序，那就没有影响了。这可能会造成一些问题——比如说，`_cap` 与 `_arr` 的初始化之间就存在依赖关系，我们必须先初始化 `_cap` 然后才能正确地初始化 `_arr`。这就要求我们在声明成员变量时，就要把 `_cap` 放在 `_arr` 之前（所幸，我们一开始的声明顺序就是正确的，所以我们不会遇到这些问题）。

```

private:
    std::size_t _cap;
    std::size_t _size;
    int *_arr; // 它的初始化依赖于 _cap，所以必须置于 _cap 之后

```

同理，我们也可以把另外两个构造函数也用初值列语法改写一下：

<sup>23</sup> 作为对比，函数的默认参数最好写在声明中，而不是写在定义中。读者不要搞混了。

```

valarri::valarri() :_cap {0}, _arr{nullptr}, _size {0}
    {} // 函数体中不再需要做什么了，空着就行
valarri::valarri(const std::initializer_list<int> &ilist)
    :_cap {ilist.size()},
    _arr {new int[_cap]},
    _size {0}
{
    for (int x : ilist) // 范围 for 循环，遍历 ilist 中的每个数
        _arr[_size++] = x; // 赋值，_size 自增
    // 循环结束后 _size 会与 _cap 相等
}

```

### 默认成员初始值

在写初值列时，我们发现，除去一些特殊情况（比如默认构造函数）外，好像 `_arr` 的初始化值都是 `new int[_cap]`。为了简单一点，我们可以直接设置一个“默认初值”，这就是默认成员初始值。

默认成员初始值要在类的定义当中写。可以用等号、圆括号或者花括号。不过延续本书的一贯的习惯，我们还是选择使用花括号。

```

private:
    std::size_t _cap {0}; // _cap 的默认初值是 0
    std::size_t _size {0}; // _size 的默认初值是 0
    int *_arr {new int[_cap]}; // _arr 的默认初值为 new int[_cap]，它依赖于 _cap

```

接下来我们就可以在此默认值的基础上，简化前面写过的三个构造函数：

```

valarri::valarri(){} // 默认初值正好合适，就不需要写多余代码了
valarri::valarri(const std::initializer_list<int>& ilist)
    :_cap {ilist.size()} // _cap 的初值需要单独指定，其它成员的初值就不需要了
{
    for (int x : ilist) // 范围 for 循环，遍历 ilist 中的每个数
        _arr[_size++] = x; // 赋值，_size 自增
}
valarri::valarri(int n, std::size_t size) :_cap {size} {
    for (; _size < _cap; _size++) // 通过循环赋值
        _arr[_size] = n; // 赋值，每个元素都赋成 n
    // 循环结束后 _size 会与 _cap 相等
}

```

这样一来，写初始化也可以事半功倍了。

### 析构函数

一个对象一经定义，它就存在了；一经初值列，它的成员就有了初值；调用完构造函数，它的存在就有了意义<sup>24</sup>。但是，无论这个对象的存储期是自动的、静态的，还是动态的，它总有存储期结束的那一刻。在生存期结束之前，它会调用析构函数——这是它安排后事最后的机会。

在 C++ 中，如果我们没有自定义析构函数，那么编译器会创建一个“什么也不做”的析构函数。如果不需要什么特别安排的话，自带的析构函数足够用了，我们根本不需要自行定义。

<sup>24</sup>关于“对象的生存期”始自何时，不同人有不同见解。有些人认为，其生存期始自定义之时；也有人认为，其生存期始自构造函数调用完成之时。本书不打算在这个问题上多做文章。

然而在涉及动态内存分配的问题上，我们就有责任定义一个能够回收动态内存的析构函数，以防内存泄漏。动态内存既不是指针的一部分，也不是对象的一部分，不会自动回收，我们只能用 `delete[]` 来回收，而析构函数就是在这方面的最佳选择。

析构函数也是一种特殊的函数，它既不能有返回值，又不能接收任何参数，一般也不需要我们手动调用<sup>25</sup>——析构函数会在对象生存期结束之前调用，调用完毕后对象的生存期立刻结束。

析构函数的名字是由一个波浪号加类名的形式写成的。它不可以重载，所以每个类只能有一个析构函数。

```
// 声明
public:
    ~valarri(); // 析构函数
// 定义
valarri::~valarri() {
    delete[] _arr; // 回收 _arr 指向的动态内存空间，注意必须用 delete[] 搭配
} // 其它成员不用再改动，_arr 也没必要指向 nullptr，反正析构函数结束了它们都没了
```

析构函数为我们提供了极大的方便，我们不需要手动回收它申请的动态内存，也不用害怕忘记回收（任何一个非动态存储期对象，只要它的生存期结束，它就一定会先调用析构函数）。STL 等类模版的内部实现全都要靠动态内存分配来实现，但我们不需要因此而为那么专属于动态内存分配的细节而感到头疼，只需要把 `std::vector` 或 `std::string` 类的对象当成自动/静态存储期的对象来使用就足够了。这正是封装的优点！

## 拷贝构造函数

有些时候，我们需要用另一个对象来为某个对象初始化，也就是把另一个对象的值复制给新定义的对象。这种情况最常见于参数传递。当我们按值传递某个类的参数时，程序都会调用这个类的构造函数来为这个形参对象进行初始化——这种构造函数又叫拷贝构造函数（Copy constructor）。

```
void fun(valarri a) { // 在传递参数时会进行拷贝构造
    //...
}
```

如果我们没有自定义拷贝构造函数的话，编译器会为我们定义一个。如果我们只需要单纯地把一个对象的值复制一份给另一个对象，那么用自带的拷贝构造函数足够了。

有些时候我们不得不自定义拷贝构造函数。比如，在设计含有动态内存分配的类时，构造函数那种单纯的“值拷贝”有时并不是我们想要的。什么意思呢？

## 浅拷贝及其缺陷

还以 `valarri` 为例。

```
valarri a {1,2,3,4}; // 用列表初始化创建一个长度暂时为 4 的数组
valarri b {a}; // 用 a 来拷贝构造 b
```

我们分析一下这个过程中会发生什么。其实很简单，`b` 的 `_cap` 私有成员将获得与 `a` 的 `_cap` 成员相同的值——其实我们也希望这样。

同样的道理，`b._size` 也将获得与 `a._size` 相同的值。这也是我们所希望的。

同理，`b._arr` 也将获得与 `a._arr` 相同的值。但是这里就有点耐人寻味了——

<sup>25</sup>一个例外是，布置分配的动态对象需要显式调用析构函数，否则可能发生内存泄漏。我们到了精讲篇再谈。

`_arr` 成员可是指针啊。如果它们指向同一段内存空间，这会意味着什么呢？会意味着，当我修改 `a[0]` 的时候，`b[0]` 也会跟着同步修改；会意味着，当我要析构 `a` 但不想析构 `b` 的时候，`a._arr` 指向的内存空间被回收，而 `b` 指向的内存空间一样被回收了（因为它们指向的就是同一个内存空间）。

简单点说就是——你这算哪门子拷贝啊，你这压根就是没扯清关系的别名吧。

这种现象就是浅拷贝（Shallow copy），它的缺陷是：它只是单纯地进行“值”层面的复制；但是对于指针来说，我们需要复制的未必是指针的值（地址），而可能是指针指向内存空间中的内容。这时候自带的拷贝构造函数就无能为力了，因此我们需要自己写一个拷贝构造函数，以此解决这个问题。

### 深拷贝及其实现

深拷贝（Deep copy）的特点在于，它不是复制地址而是复制内容。图 8.2 展示了它们之间的区别。



图 8.2: 浅拷贝与深拷贝

要实现深拷贝其实很简单，就像一般的构造函数那样写就行了，只是参数类型变成了 `const valarri&` 而已，剩下的只是按我们的需要复制值或内容而已。

```

// 声明
public:
    valarri(const valarri&); // 拷贝构造函数
// 定义
valarri::valarri(const valarri &a)
    :_cap{a._cap},
     _size{a._size},
     _arr{new int[_cap]} // _arr 应指向自己独有的存储空间，而不是和 a._arr 一样
{ // (另：鉴于 _arr 的默认初值就是 new int[_cap]，所以这里可以省略这个初值)
    for (int i = 0; i < _size; i++)
        _arr[i] = a._arr[i]; // 拷贝每个值，而不是拷贝地址
}

```

不止是拷贝构造函数，赋值运算符也必须自定义，从而防范发生“复制了地址而不是内容”的情况。不过赋值运算符相比于拷贝构造函数，还需要考虑更多东西才行。

首先是防止自我赋值。这个好说，按照老办法，检测一下 `&a==this` 就行了。

还有，这个对象的 `_arr` 成员指向的内存空间可能已经存储了一些值。这些值当然可以不管，但 `_arr` 指向的内存空间不能不管。怎么处理呢？如果我们需要赋值的数据太多，现有的容量装不下，那

我们就得重新分配一个更大的内存空间了；如果需要赋值的数据不多，现有的容量足够装得下，那我们就直接装进去就好了。所以我们需要一个条件语句。

```
valarri& valarri::operator=(const valarri &a) { //重载赋值运算符，防止浅拷贝
    if (&a == this) //防止自我赋值
        return *this;
    if (_cap < a._size) { //如果现有容量装不下a中的所有数据，那就要扩容
        delete[] _arr; //别忘了先回收现在指向的内存空间！！
        _cap = a._size; //希望分配一个大小为a._size的内存空间
        _arr = new int[_cap]; //分配动态内存空间
    }
    for (_size = 0; _size < a._size; _size++)
        _arr[_size] = a._arr[_size]; //用循环语句复制内容
    return *this; //返回*this
}
```

我们在本节中修改了 `valarri` 的定义，包括但不限于添加了 `_cap` 成员，且把 `_arr` 成员由数组改成了指针。所以我们在第二节中写的很多函数也需要重新修改调整。本书没有余裕一一赘述，就请读者自行尝试吧。

## 8.4 成员的属性

前三节的内容比较硬，读者可能也看乏了，所以本节我就来放放水。

### 静态成员

一个类的非静态成员是无链接的。就以 `valarri` 为例，此对象的 `_cap` 成员和彼对象的 `_cap` 成员完全不是一个事物，它们在内存中有不同的地址——仅这一点足够说明问题了。

静态成员（Static member）则不然，它具有内部链接。换句话说，只要是这个类的静态成员，甭管是此对象的成员还是彼对象的成员，它们都是同一个事物，有完全相同的内存地址。

```
struct C { //这里用 struct 是因为 struct 的成员访问权限默认为 public，方便我们讲解
    int nonsta {}; //默认成员初始值
    static int sta;
};

int C::sta {};//统一初始化为空，将初始化为0
int main() {
    C a,b;
    std::cout << &a.nonsta << std::endl << &a.sta << std::endl
        << &b.nonsta << std::endl << &b.sta << std::endl;
}
```

这段代码的输出结果为

---

```
0x7ffd441a2908
0x600c54
0x7ffd441a290c
0x600c54
```

---

我们可以看到，这里 `a.nonsta` 和 `b.nonsta` 的地址值不同；而 `a.sta` 和 `b.sta` 的地址值相同，这是因为它们的存储位置不一样——非静态对象的非静态成员位于栈段，而静态成员位于数据段或 BSS 段。

(非常量的) 静态成员变量不能在类当中定义，只能声明；所以我们要把它放在类外定义，就像上面这段代码中写的那样。而在定义的时候呢，我们不能再写 `static` 了。

`C::sta` 定义在全局作用域，所以它不仅对于 `C` 类来说是静态成员，而且还具有外部链接——我们可以在一个源文件中为 `C::sta` 进行定义，而在另一源文件中不定义就可以使用它。

```
//Header.h
struct C {
    static int sta; // 静态成员变量
};

//Definition.cpp
#include "Header.h"
int C::sta {}; // 它具有外部链接

//main.cpp
#include "Header.h"
int main() {
    std::cout << C::sta; // 此C::sta与Definition.cpp中的C::sta一致
}
```

我们还可能会图方便，直接把 `C::sta` 定义在头文件中，但是我们会立刻发现一个问题：如果把 `C::sta` 定义在头文件中，那么两个源文件在包含此头文件后，就会在两个文件中分别产生两个定义，这就违反了单一定义规则 (ODR)，如图 8.3 所示。

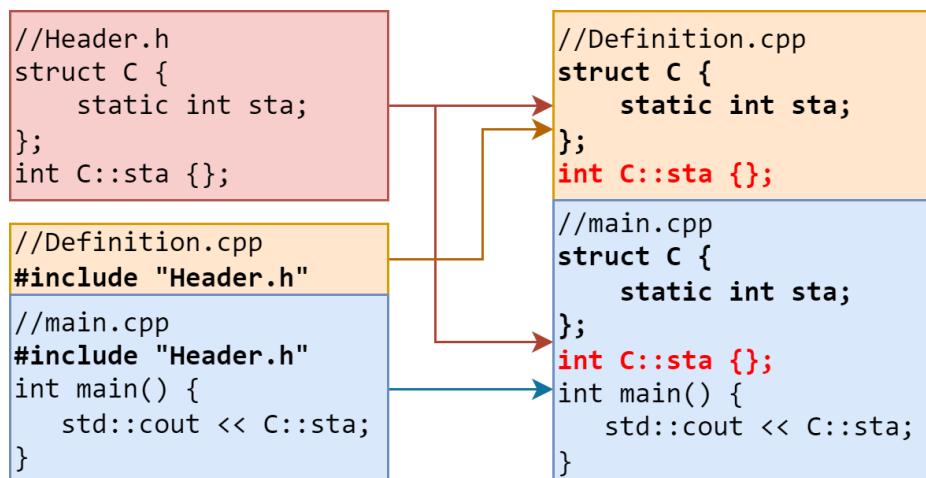


图 8.3: 把 `C::sta` 定义在头文件中将导致具有外部链接的变量被定义两次

所幸，从 C++17 起我们可以通过定义内联变量的方式来解决这个问题。

```
//Header.h
struct C {
    static int sta;
};

inline int C::sta {};
```

`inline`<sup>26</sup>关键字的作用在于，允许拥有外部链接的函数/变量在不同翻译单元中分别定义（但是，在每个翻译单元中仍然只能有单个定义）。这些定义应该要相同；如果不相同，编译器会自行选择其中一个作为它的定义（因为不能违反 ODR）——那就属于未定义行为（Undefined behavior）了，我们最好别这么干。

对于定义在头文件的函数/对象来说，内联是我们的最佳选择。

言归正传。静态对象有很多用处，比如说在 `valarray` 类中，我们把 `MAX_SIZE` 成员定义成了静态成员。这样的好处在于，其一，它对 `valarri` 类的所有对象都是统一的，我们不必担心某个对象的数据被误修改从而导致不统一；其二，静态成员的内存空间是单独的，而且对于所有对象来说只此一个，这也就意味着我们不需要为每个对象都定义一个专属的 `MAX_SIZE` 成员，从而浪费大把的内存空间。

还有一个常见的应用是对象个数计数器。我们可以定义一个 `_object_number` 静态私有成员，它的初始值为 0，可以在每个构造函数调用之时自增，而在每个析构函数调用之时自减。

```
// 声明
private:
    static std::size_t _object_number;
// 定义
inline std::size_t valarri::_object_number {0};
```

至于构造函数和析构函数中相关的修改，我就不在此一一写出了。

虽然 `_object_number` 是私有成员，但是它也要在类外定义（静态成员的定义是唯一一个可以在类外“访问”私有对象的机会）。

我们还可以定义静态成员函数。静态成员函数虽然是成员函数，但是它没有 `this` 指针，也不能是常成员函数……总之有不少限制。但是我们可以直接用类名的形式来访问它，不需要用任何对象。举个例子吧，我们可以定义静态成员函数 `object_number`，让它返回 `_object_number` 静态成员变量。这样我们既能读取这个私有成员的值，又能防止外界对它的篡改。

```
// 声明&定义
public:
    static std::size_t object_number() { return _object_number; }
    // 这个函数很短，直接定义就够了
```

这往后我们就可以用 `valarri::object_number()` 来返回当前有多少个 `valarri` 对象了。

```
std::cout << valarri::object_number();
```

## 常量成员

我们可以把一个成员变量定义成常量成员变量——这个名字有点怪，我们就干脆叫它作“成员常量”好了。

一个成员常量一经初始化就不能再改了——这个初始化是初值列或默认成员初始值那样的初始化，而不是在构造函数体内部的那种看似“初始化”实则“赋值”的操作。我们在 `valarri` 类中定义过一个成员常量 `MAX_SIZE`，实际上它还是一个静态成员。

```
private:
    static const std::size_t MAX_SIZE {1000000}; //MAX_SIZE是一个静态成员常量
```

<sup>26</sup>关于 `inline` 的其它作用，有些资料声称使用 `inline` 可以降低部分函数运行的时间；而有些资料则声称不能。笔者在这方面无暇深入研究，所以本着宁缺勿滥的原则，不作此方面的介绍，仅介绍笔者已知的 `inline` 关键字作用。

对于静态成员来说，如果它是 `const` 或者 `constexpr`，那么我们可以直接在类中给它定义，而不需要写在类外。

至于成员函数——我们可以定义常成员函数，但定义的方法是在参数列表和函数体之间的位置加上一个 `const`。就以 `valarri::size` 为例，我们完全可以把它定义成常成员函数，这样写就好：

```
// 声明&定义
public:
    std::size_t size()const { return _size; } // 返回当前的存储量大小
```

如果我们要把声明和定义分开写，那就需要在声明和定义处都写上这个 `const` 才行，一处也不能少。

```
// 声明
public:
    std::size_t size()const; // 声明处要有 const
// 定义
std::size_t valarri::size()const { // 定义处也要有 const
    return _size;
}
```

说了半天，常成员函数到底是做什么的呢？我们又为什么要定义它呢？

不知读者是否有试过这样一种情形：

```
void func(int &ref) {} // func 函数接收一个引用，但它什么也不做
int main() {
    const int a {}; // 把 a 定义成一个 int 型常量
    func(a); // 试图把 a 作为实参传入 func 中
// error: binding reference of type 'int&' to 'const int' discards qualifiers
    return 0;
}
```

这个报错信息的含义是，不能把 `int&` 类型的形参绑定到 `const int` 类型的实参。

其实这是 C++ 的一种防御机制，它不允许你把非常量的引用绑定到常量上；否则的话你就能够用这个引用来修改常量的值了，这成何体统啊？所以我们必须把 `func` 的参数设为 `const int &ref`。只有这样，我们才能从函数声明上保证不用 `ref` 来修改其值；也只有这样，编译器才会放心地让我们像 `func(a)` 这样调用函数。

成员函数也是同理。读者应该记得我们说过，非静态成员函数都有一个隐藏参数，那就是“调用它的函数”。这个参数就像刚才的 `func` 一样，它把“调用它的函数”以一种类似引用的形式传入参数，所以我们可以 `this` 指针来改变此对象的成员（或者直接改变，这未必不是一种语法糖）。

但是如果我们需要定义一个常量对象呢<sup>27</sup>，这时我们就会发现问题了——

```
const valarri arr{1,2,3,4}; // 定义一个常量对象
std::cout << arr[0]; // 试图访问 arr[0]，但我们没打算修改它的值
// error: passing 'const valarri' as 'this' argument discards qualifiers
```

这也同样是 C++ 的保护机制在作怪——它不允许你把 `arr` 作为一个参数传给 `valarri::operator[](int)`，因为这个函数并不保证传入的隐藏参数是常量引用。

这也是我们使用常成员函数的原因。当我们在参数列表之后加上一个 `const` 之后，我们就是在向编译器宣告：这个函数不会改变调用它的对象的值！

<sup>27</sup> 读者不要武断地认为，我们不需要用 `const` 定义常量对象，所以也不需要这些常成员函数之类花里胡哨的东西。不，我们需要！尤其是在引用参数传递时，只要你写成 `const valarri&`，这时这个参数就会被当作常量对象！

```
// 声明&定义
public:
    int& operator[](int i) { return _arr[i]; } // 非常成员函数
    int operator[](int i) const { return _arr[i]; } // 重载为常成员函数
```

这两个成员函数之间是重载关系。第一个函数接收一个隐藏的 `valarri&` 参数和一个 `int` 参数；第二个函数接收一个隐藏的 `const valarri&` 参数和一个 `int` 参数——所以它们是不一样的，编译器会有所选择<sup>28</sup>。

读者还需要注意，这里第二个函数的返回类型不是 `int&` 而是 `int`。这同样是防止修改常量的一种方式。只不过，如果我们不慎将它的返回类型真的定义成了引用，编译器也无法检查出来。如果在这种情况下“越狱”修改常量，那就会引发不可预知的后果，同样属于未定义行为。所以读者也勿必多留个心眼，不要在这里失误。

`valarri` 类中还有许多函数可以定为常成员函数，比如 `valarri::max`, `valarri::min`, `valarri::sum`, 还有 `valarri::operator+(const valarri&)` 之类的运算符，只要它们不需要修改 `*this` 本身，我们就可以把它们都定义成常成员函数——这样总是没有坏处的，毕竟可变对象可以调用常成员函数，但常量对象就真的不能调用非常成员函数（注意断句：非/常成员函数）了。

```
// 声明
public:
    int sum() const; // 总和
    valarri operator+(const valarri&) const; // 可以与数组相加
    // 其它几个省略

// 定义
int valarri::sum() const { // 返回值是当前数组中有效数据的总和
    int summation {0}; // 先置为0
    for (int i = 0; i < _size; i++)
        summation += _arr[i]; // 然后把每个数都加到 summation 上
    return summation;
}

valarri valarri::operator+(const valarri &a) const {
    valarri v; // 有了默认构造函数之后可以不写花括号了
    v._cap = std::min(_cap, a._cap); // 先确定v的容量
    v._arr = new int[v._cap]; // 分配动态内存。_arr原本指向空，不用回收什么
    for (v._size; v._size < v._cap; v._size++)
        v._arr[v._size] = _arr[v._size] + a._arr[v._size];
    return v; // 返回值是v
}
// 其它几个省略
```

## 插曲——常量对象的指针成员

一个类的对象可以用 `const` 限定符规定为常量对象。对于一个常量对象来说，它的基本数据类型成员变量——如 `int` 或 `double` 成员，都会变成 `const` 成员；它的数组会变成常量数组，我们不能用下标运算符改变数组的元素；它的指针会变成常量指针，即它的指向不能改变，但它的内容依旧可以改变。

<sup>28</sup>简单来说，编译器会对左值变量优先匹配 `T&` 参数，次优先匹配 `const T&` 参数；对左值常量优先匹配 `const T&` 参数，而不能匹配 `T&` 参数。我们在精讲篇中再谈。

常量指针是一个坑，我们需要在写代码时就加以防范，避免让这个类提供可以改变常量成员的途径，比如说在定义 `valarri::operator[](int) const` 时不小心把返回类型写成 `int` 这样。

一般来说，我们用这种方式来防范就已经足够。只要我们审慎地检查每一个点，就不会出现这种问题；但读者还是可能会想，我为什么不把它设为指向常量的指针呢？这样我们不是就可以防止修改内容呢？

但是请麻烦读者作一个整体性的思考：如果我们定义的是非 `const` 对象，那么我们应当允许这个指针没有限制地修改指向和内容；而如果我们定义的是 `const` 对象，那么我们应当禁止这个指针修改指向或内容。现在的困难是，如果我们定义了无任何限制的指针，那么常量对象的指针就能够修改内容；如果我们定义了指向常量的指针，那么非常量对象的指针也不能修改内容了！所以这种一刀切的方法是行不通的，无论怎样都会违背我们的初衷。

要正确地解决这个问题，我们的原始思路是——定义两个指针，一个是无限制指针 `_arr_m`；一个是指向常量的指针 `_arr_c`。

```
private:
    int *_arr_m; // 指向变量的指针
    const int *_arr_c; // 指向常量的指针
```

接下来，凡是在常成员函数中，我们就使用 `_arr_c`；凡是在非常成员函数中，我们就使用 `_arr_m`。另外需要注意，非常成员函数有可能会改变 `_arr_m` 的指向（比如说重新分配动态内存），所以我们每次改变 `_arr_m` 的地址值后都要把 `_arr_c` 和它的值对齐——也就是给 `_arr_c` 赋值为 `_arr_m` 的值<sup>29</sup>。

这种做法有两项缺陷：第一，我们需要用两个指针来操作，不仅麻烦，还增加了内存占用（原来我们只需要一个指针就够）；第二，我们总是要在改变 `_arr_m` 之后进行对齐，但这也是很容易被忘记的！其实它们的共同原因是，我们使用了两个指针；所以我们需要想到一种更有效，更方便，也更安全的方式。我的选择是：

```
private: // valarri 的 private 部分
class Arr{ // 内嵌的类，类名为 Arr
public:
    int*& p() {return _p;} // 返回类型是“对指针的引用”
    const int* p() const {return _p;} // 返回类型是“指向常量的指针”
private:
    int *_p;
} _arr; // 直接定义该类的 _arr 对象
```

这种方法的特点在于，我们只需要一个 `_arr._p` 指针就可以了，既能节省内存空间，又不需要考虑对齐问题。

如果一个 `valarri` 对象是常量，那么它的成员 `_arr` 也是常量，所以每次调用 `_arr.p()` 的时候调用的都是它的常成员函数，返回值是指向常量的指针。这个返回值不是对指针的引用，所以它不能修改 `_arr._p` 的指向；同时它也不能修改内容，因为返回类型就是指向常量的指针。

如果一个 `valarri` 对象不是常量呢？那么它的成员 `_arr` 也不是常量，所以每次调用 `_arr.p()` 的时候调用的都是它的非常成员函数，返回值是对指针的引用。这个返回值是对指针的引用，所以它能修改 `_arr._p` 的指向；同时它也能修改内容。

再补充一些内容——为了方便用初值列或默认成员初始值，我们需要为 `valarri::Arr` 类设计一个构造函数。所以我们可以把它改成这样：

```
class Arr{ // 内嵌的类，类名为 Arr
```

<sup>29</sup>这个对齐操作不是一个建议，而是必须的。因为一个变量可以隐式类型转换成常量，所以对于变量对象来说，`_arr_c` 可能在某些情况下用不到，但在彼时常量类型转换之后就可能用到了，不能弃之不顾。

```

public:
    Arr(int *ptr) : _p{ptr} {} //构造函数，接收一个指针初始化
    int*& p() { return _p; } //返回类型是“对指针的引用”
    const int* p()const { return _p; } //返回类型是“指向常量的指针”

private:
    int *_p;
} _arr {new int[_cap]};

//声明 _arr 时使用它的构造函数 Arr::Arr(int*), 这也是 _arr 的默认成员初始值

```

这个内嵌类对象的构造过程对于初学者来说，可能有点复杂，但是也不要太害怕，其中的道理还是我们前面讲的那些，只是在具体应用上，我们增加了一个嵌套，仅此而已。

接下来我们只需要在 `valarri` 所有成员函数中都使用 `_arr.p()` 就可以了，以下是一组例子：

```

public:
    int& operator[](int i) { return _arr.p()[i]; } //非常成员函数
    int operator[](int i)const { return _arr.p()[i]; } //重载为常成员函数

```

这里的 `_arr.p()[i]` 表示方式看上去有点怪，但是我一解释你就懂了：`_arr` 是什么？它是一个 `valarri::Arr` 类的对象嘛。`_arr.p()` 是什么？它是 `_arr` 调用的成员函数嘛。`_arr.p()` 的返回值是什么？`int*` 或者 `const int*`，总之是一个指针——实际上它会指向动态内存空间，也就可以作为动态数组使用。既然 `_arr.p()` 它是一个指针，那么我们用 `_arr.p()[i]` 来访问内容不就很合理吗？

## mutable 成员

对于任何一个常量对象来说，它的非静态成员都会成为常量。这也就意味着，一经初始化，这个对象的任何值都不能进行修改了。

不过，在有些情况下，出于记录、标记等需要，即便一个对象是常量，我们也希望修改它的个别成员。`mutable` 关键字的作用在于，它强制规定了无论这个对象是不是 `const` 常量，这个成员都是可变的。

先举个简单的例子：我们需要一个计数器，来记录每一个对象调用某成员函数的次数。常量成员当然可以调用常成员函数，所以我们的计数器也需要可以改动。这时候使用 `mutable` 就合情合理了。

```

class C {
public:
    void fun() { //非常成员对象版本
        ++count;
        //...
    }
    void fun()const { //常成员对象版本
        ++count; //计数器自增
        //...
    }
private:
    //... 其它非mutable 成员
    mutable std::size_t count; //计数器，即便对于常量对象来说也可变
};

```

还有一个更复杂的例子，就是作为延迟标记（Lazy tag）。

## 延迟标记

在实际编程的过程中我们可能会遇到这样的问题：

```
char str1[10000] /*太长，省略*/; str2[10000];
for (int i = 0; i < std::strlen(str1); i++)
    str2[i] = str1[i];
```

这段代码的运行效率很低，但我们不应该把它归咎于赋值语句；真正的问题出在 `std::strlen` 上。`std::strlen` 的算法原理是很原始的“逐个读取字符，直到遇到 '`\0`' 为止”。下面是它的一种可能的实现方式：

```
std::size_t strlen(const char *start) {
    const char *end = start;
    while (*end != '\0')
        ++end;
    return end - start;
}
```

换句话说，我们每次调用 `std::strlen(str1)` 时，都是在逐个读取 `str1` 中的数据，最后计算出值来。那么这样的调用进行了多少次呢？`std::strlen(str1)` 次！如果这个字符串长达 8000 字节，程序将进行  $8'000^2 = 64'000'000$  次读取——相比之下，它只不过进行了 8000 次 `str2[i]=str1[i]` 的赋值而已。

但是我们想一下就会发现，这 8000 次调用的结果值都是一样的，所以我们完全没必要让它一遍遍地重新计算，直接计算一次，然后把这个值记录下来就好了。

```
char str1[1000] /*太长，省略*/; str2[1000];
std::size_t strlen{ std::strlen(str1) }; //只调用 std::strlen 一次
for (int i = 0; i < strlen; i++)
    str2[i] = str1[i];
```

这就是一种原始的“耗费一些存储空间，从而换取更多的计算时间”（用空间换时间）思想，或曰记忆化（Memoization）。如果一个什么东西需要我们反复计算，但多次计算出来的值都相同，那我们就别浪费宝贵的时间了，直接找个变量把它存起来就好。

`std::string` 有一个 `length()` 成员函数，它的返回值就是一个预先计算好，并单独存起来的数<sup>30</sup>，所以它不需要进行多余计算，就能节省很多时间。

```
std::string str /*太长，省略*/;
for (int i = 0; i < str.length(); i++) //放心用，效率很高！
    //...
```

延迟标记在这个基础上更进一步。它连一开始的那次计算都不做了，只是打一个“搁置再议”的标签而已。什么时候开始计算呢？等我们需要这个值的时候再算。这样的好处是，如果在这个对象的整个生存期内都不需要用到这个值，那我们岂不是连第一次的计算都省了？（图 8.4）

对于常量对象来说，我们为它设计的延迟标记也需要有修改的机会。就不拿 `valarri` 来举例了，我们换个例子。

一个三角形的三条边长  $a, b, c$  确定以后，我们就可以根据海伦公式计算出它的面积  $A$ ：

$$s = \frac{1}{2}(a + b + c)$$

$$A = \sqrt{s(s - a)(s - b)(s - c)}$$

---

<sup>30</sup>我们在第六章第五节中有介绍过。

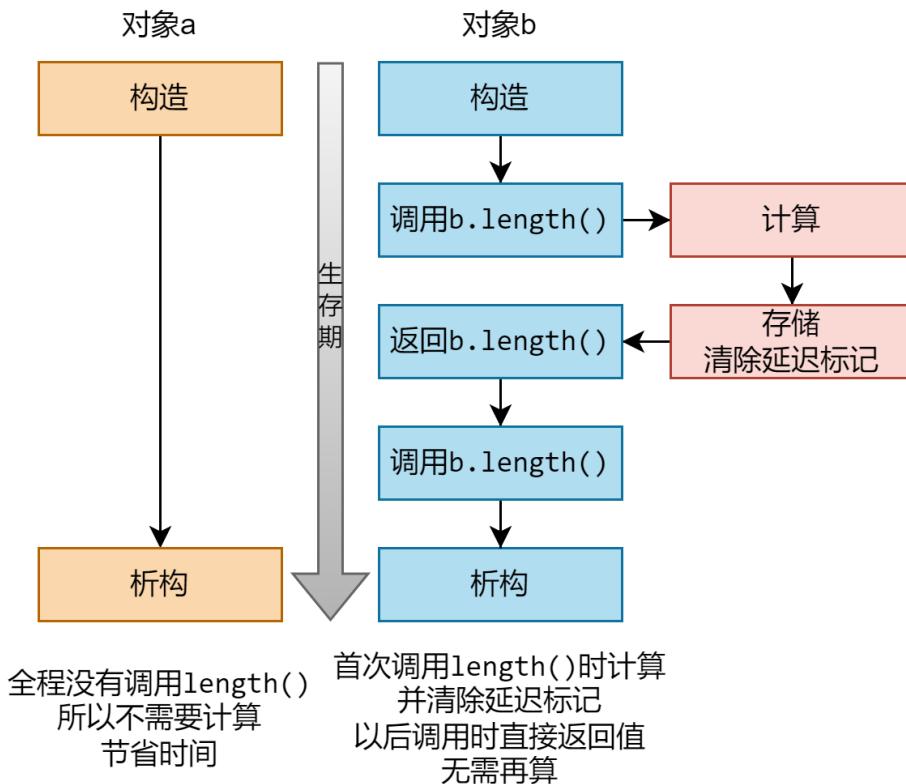


图 8.4: 延迟标记的作用

不过这个计算有点麻烦，我们可以提供一个延迟标记，等到需要这个值时再计算。

```
#include <cmath>
class Triangle {
public:
    Triangle(double a, double b, double c)
        : _a {a}, _b {b}, _c {c} {} // 函数体为空
    double area() const { // 它是常成员函数，保证_a, _b, _c 不会被修改
        if (!_area_cached) { // 如果还没有缓存_area 的值
            double s = (_a + _b + _c) / 2.; // 临时变量 s
            _area = std::sqrt(s * (s - _a) * (s - _b) * (s - _c));
            _area_cached = true; // 现在它缓存了，延迟标记变为 true
            // 这两个 mutable 变量是可以在常成员函数中修改的
        }
        return _area; // 返回已经缓存的面积值
    }
private:
    double _a;
    double _b;
    double _c;
    mutable bool _area_cached {false}; // 延迟标记， false 表示尚未缓存
    mutable double _area; // 缓存的面积值
};
```

于是，当我们定义一个对象时，无论是不是常量对象，它都有这样的特点：

- 如果一直不调用 `area()` 成员函数，这个值永远不会被计算，节省了计算时间；
- 第一次调用 `area()` 成员函数，这个值被计算出来并存储到 `_area` 中，同时延迟标记 `_area_cached` 消除；
- 以后再调用 `area()` 成员函数，直接返回 `_area` 就行了，无需再次计算。

很多人容易滥用 `mutable` 这个关键字，用来表达一些“可修改”的变量，然后把所有的对象都定义成 `const` 常量对象。这是非常糟糕的习惯！

```
class Individual { //不要这样做！
private:
    std::string name;
    unsigned id;
    enum Sex : bool {male, female};
    Sex sex;
    mutable age; //年龄是可变的
};

const Individual /*定义各种对象都在用const*/;
```

而合理的用法应当是：当一个对象在逻辑上是不可变的时候（比如说，这个 `Individual` 对象的任何特性都一经定义就不发生变化），我们才会把它定义成常量对象。但如果只是个别成员不可变（比如姓名、身份号码），那我们应该把这些成员定为常量，而不是用常量对象 + 其它成员设为 `mutable` 的方式来实现。

那么在什么时候我们才会用 `mutable` 呢？应当是类似于上面那两种情况，即：这个成员与对象本身的逻辑关联并不大，它充当一个单独作用的量；但无论这个对象是否是常量，该成员都必须可变。这种情况是比较少见的，所以 `mutable` 在实际编程中出现的频率很低。

总而言之，`mutable` 是仅当我们有特殊需求的时候才会使用的，因此不要像刚才的例子那样滥用。

## 8.5 类型转换函数

在介绍基本数据类型的时候，我们都提过它们之间的类型转换。比如说，一个 `int` 类型的数据可以转换成 `double` 型，反之亦然。我们可以用三种风格的语法来进行显式类型转换。就以 `int` 类型的 `i` 转换成 `double` 类型的 `d` 为例吧：

- C 风格类型转换：`d=(double)i`
- 函数风格类型转换：`d=double(i)`<sup>31</sup>
- 静态类型转换：`d=static_cast<double>(i)`

一般来说，在 C++ 中我们更推荐使用静态类型转换，因为这个方法最稳妥。C 风格方法在 C++ 中是 `const_cast`, `reinterpret_cast` 和 `static_cast` 的聚合。它在有些时候确实很方便，但是也能会因为太自由而带来我们不易发现的麻烦，所以我们需要加以限制。

至于函数风格的类型转换，这个更复杂些，这也是我们本节要讨论的重点。不过如果像我们刚才写的那样，一个圆括号里套单个数的话，它和 C 风格类型转换的效果是相同的。

<sup>31</sup>注意，有些类型，比如 `long long` 或 `int*`，如果想要通过函数风格进行类型转换，那么我们就要为这个类型套括号，写成 `(long long)(i)` 这样。

## 转换构造函数

我们可以用 `double` 类型来表示一个实数。虽然它的精度不够，不能表示真正意义上的无理数，但我们还是需要在精度上对它宽容一点的。

现在我不只想要表示实数，我还希望表示复数，那么我们就要写一个 `Complex` 类。它应当具备两个成员，一个表示实部，一个表示虚部，还支持一些简单的运算。

```
class Complex {
private:
    double _r {}; // 表示实部的值， 默认初值为 0
    double _i {}; // 表示虚部的值， 默认初值为 0
public:
    Complex(double r, double i) : _r {r}, _i {i} {} // 构造函数
    Complex(const Complex &c) : _r {c._r}, _i {c._i} {} // 拷贝构造函数
    Complex& operator=(const Complex &c) { // 赋值运算符
        _r = c._r;
        _i = c._i;
        return *this; // 其实这里也不需要防止自我赋值，反正没风险
    }
    //... 待补充
}; // 这个 Complex 类不需要手动写析构函数， 默认析构函数足够了
```

实数和复数之间有着密切的联系，我们能否让它们之间互相支持类型转换呢？我们的答案是“可以”！

在 C++ 中，我们可以通过构造函数来实现其它类型到此类型的类型转换。我们把这种函数叫作 **转换构造函数（Converting constructor）**。在转换构造函数当中最为特殊的，也就是传统意义上的转换构造函数，就是单个参数的转换构造函数<sup>32</sup>。

```
public:
    Complex(double r) : _r {r} {} // 单个参数的转换构造函数
```

有了这个构造函数之后，我们就可以用前述三种风格的类型转换来把 `double` 类对象转换为 `Complex` 类对象了。

```
void fun(Complex c) {} // 什么也不做，单纯传个参
int main() {
    fun(0.); // 把一个 double 对象传入 fun 函数？
}
```

事实证明这段代码是可以正常编译运行的——换句话说，`double` 类的 `0.` 被隐式类型转换成了 `Complex` 类的对象（按构造函数，这个对象的实部和虚部都应当是 `0.`），然后传入了 `fun(Complex)` 当中。

除了隐式类型转换，我们还可以写成如下显式的形式：

```
fun((Complex)0.); // C 风格类型转换
fun(Complex{0.}); // 函数风格类型转换，使用花括号比圆括号更好
fun(static_cast<Complex>(0.)); // 静态类型转换
fun(Complex{0.,0.}); // 它可以接收两个参数，相当于对列表进行了类型转换
```

我们刚才是定义了两个构造函数对吧，其中一个用来接收单参数，一个用来接收双参数；除此之外我们还应当定义一个不接收参数的默认构造函数。这么多，有点太麻烦了。

<sup>32</sup> 在 C++11 以前，只有单个参数的转换构造函数被认为是转换构造函数，其它的构造函数都不是。而在 C++11 以后，这个概念扩展到所有非 `explicit` 构造函数，但是单个参数的转换构造函数仍然是其中最常用的一种。本书也主要关注这一种。

为了简便起见，我们还可以把它改得更简洁一点，让一个构造函数代替这三个构造函数。读者可能猜到了，我们的方法就是设置默认参数：

```
class Complex {
private:
    double _r {}; // 表示实部的值， 默认初值为0
    double _i {}; // 表示虚部的值， 默认初值为0
public:
    Complex(double r = {0}, double i = {0}) : _r{r}, _i{i} {}
        // 带默认值的构造函数， 可以接收零到二个参数
    Complex(const Complex &c) : _r {c._r}, _i {c._i} {} // 拷贝构造函数
    Complex& operator=(const Complex &c) { // 赋值运算符
        _r = c._r;
        _i = c._i;
        return *this; // 其实这里也不需要防止自我赋值， 反正没风险
    }
    // ... 待补充
};
```

在这种情况下我们也可以把它当作转换构造函数来用，效果相同，就请读者自行尝试吧。

类型转换的实际规则比我们想象中复杂很多。举个例子来说，如果你这样写，那也是可以通过编译的：

```
fun(0); // 一个int型对象也能转换成Complex?
```

这里我们调用 `fun` 函数的时候传入的是 `0`，这可是 `int` 型的对象啊！但是我们的构造函数中写的可是 `double`，这怎么可能呢？

实际上，程序在运行时会先把 `int` 类型的 `0` 转换成 `double` 类型，然后再把 `double` 类型转换成 `Complex` 类型。所以一个看似云淡风轻的类型转换过程，其中会有多少复杂的细枝末节啊！

## 自定义转换函数

构造函数能够部分地解决我们的类型转换需求，但不是全部。试想，我们可以通过 `complex::Complex(double=0)` 构造函数来实现 `double` 到 `Complex` 的类型转换，但是我们实现不了 `Complex` 到 `double` 的类型转换啊！我们总不可能写一个 `double` 的构造函数吧。

自定义转换函数（User-defined conversion function）能解决这个问题。它和转换构造函数相辅相成，转换构造函数规定了“哪些类型能转换成此类型”，而自定义转换函数则规定了“此类型能转换成哪些类型”。自定义转换函数的定义语法和运算符重载有点相似，不过又不尽相同。

```
public:
    operator double() { // 不允许指定返回类型，也不允许使用参数
        return _r; // 这是类似截尾的处理方法，直接无视虚部，返回实部
    }
```

自定义转换函数不能指定返回类型，看上去和构造函数有点像，但它们根本不一样！构造函数是“没有返回值”，我们调用构造函数之前这个对象已经存在；而自定义转换函数其实有返回值，只不过它的返回类型是写在了函数名中，所以我们不用单拎出来再写一次。

自定义转换函数也不能指定参数，这是因为转换函数只能接收一个参数——没错，正是我们反复提及的那个“调用它的函数”。在实际的类型转换过程中，我们的确可以写成 `c.operator double()`，不过我们很少这个干，而是用那三种常见的类型转换语法。

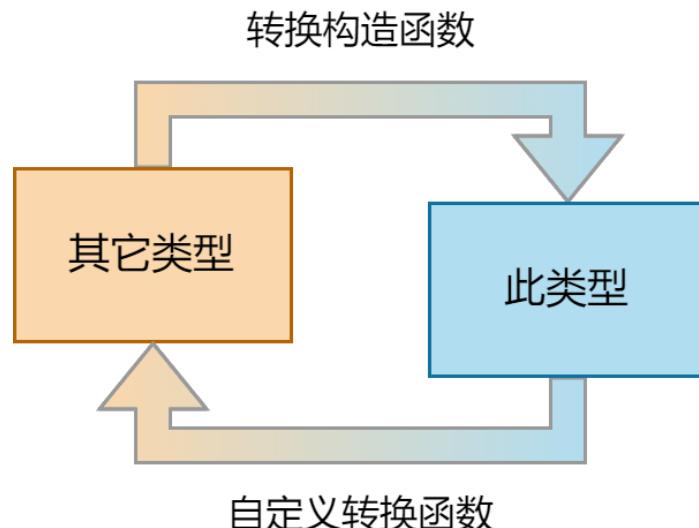


图 8.5: 转换构造函数与自定义转换函数的作用

```

Complex c{1,2};
std::cout << (double)c << std::endl; //C风格转换
std::cout << double{c} << std::endl; //函数风格类型转换，这里也用花括号
std::cout << static_cast<double>(c) << std::endl; //静态类型转换
std::cout << c.operator double(); //客串一下而已，完全不推荐使用

```

最后我们来尝试一个更有趣的例子：为 `valarri` 设计一个转换构造函数和一个自定义转换函数，允许我们把 `valarri` 对象转换成 `std::vector<int>` 对象，反之亦然。

为了方便起见，我们还是使用 `vecint` 作为 `std::vector<int>` 的别名，但其中道理是不变的。

```

#include <vector> //使用vector必备的库
using vecint = std::vector<int>; //别名声明

```

接下来我们先考虑它们的声明：

```

//声明
public:
    valarri(const vecint&); //vecint到valarri的转换构造函数
    operator vecint() const; //valarri到vecint的自定义转换函数

```

我们的转换构造函数应设为 `const vecint&`，这样的好处是它可以接收常量作为参数——常量对象当然也可以类型转换嘛。同样的道理，我们也应该把自定义转换函数设为常成员函数。

在定义其转换构造函数时，不要忘记定义构造函数的一般规则——初值列和默认成员初始值，代码重用等。如果你有计数器，也别忘了这个静态成员。

```

valarri::valarri(const vecint &vec) //vecint到valarri的转换构造函数
: _cap {vec.size()}, //valarri的成员对vec私有成员无访问权限，要用其公有成员
  _size {vec.size()}
{
    for (int i = 0; i < _size; i++)
        operator[](i) = vec[i]; //vecint也有重载下标运算符，且有常成员函数的重载
    _object_number++; //计数器自增
}

```

这里的 `operator[](i)` 用法有点特别，我来解释一下。

我们在前面已经定义了两个 `valarri` 的下标运算符重载，一个是常成员函数，另一个不是。

```
public:
    int& operator[](int i) { return _arr.p()[i]; } //下标，非常成员函数
    int operator[](int i) const { return _arr.p()[i]; } //下标，常成员函数
```

所以我们当然也可以调用这个成员函数了。这个成员函数的函数名是 `operator[]`，接收一个 `int` 型参数，所以我们调用时就要写成 `operator[](i)`，这就合理了吧。

如果你认为这种写法太过奇怪，不能接收，那么你也可以写成 `(*this)[i]`。`this` 是指向本对象的指针，那么 `(*this)` 就是本对象；而 `(*this)[i]` 就是本对象在调用下标运算符嘛，这就很好理解了。

至于自定义转换函数，它的道理是一样的，我就不加讲解直接写了。

```
valarri::operator vecint()const { //valarri到vecint的自定义转换函数
    vecint vec(_size); //调用了vecint的构造函数，直接指定vec的大小
    for (int i = 0; i < _size; i++)
        vec[i] = operator[](i); //逐个赋值，不要用push_back，反复重分配太费时
    return vec;
}
```

这里我们调用 `vecint` 的构造函数时依然要注意，要用圆括号而不是方括号。

有了这个类型转换函数之后，对于期待 `vecint` 的场合，我们也可以用 `valarri` 通过类型转换来传入参数；反之亦然。这样一来，我们就能可以在一定程度上打破这两个类型的隔绝状况。

## 隐式类型转换与函数重载

有的时候用隐式类型转换可以帮助我们节省很多运算符/函数重载方面的压力。就以复数类而言吧，如果我们要重载加法运算符，可能我们需要这些函数：

```
Complex Complex::operator+(const Complex &com)const; //成员函数
Complex Complex::operator+(double num)const; //成员函数
Complex operator+(double num, const Complex com); //非成员函数
```

这三种情况分别对应着复数 + 复数/复数 + 实数/实数 + 复数。看上去真是又麻烦，又杂乱——我们还要为减法/乘法/除法等，对每个运算符都要定义三个版本才行。

但是有了类型转换之后，我们只需要定义单个版本就够了：

```
Complex Complex::Complex(double r, double i); //转换构造函数，其中i可以有默认值
Complex operator+(const Complex &com1, const Complex &com2); //只此一个足够了
```

那么，在遇到复数加实数或者实数加复数的情况时会发生什么呢？答案是：传入的实数会隐式类型转换成复数。

```
Complex{1,2} + 1.5; //(1+2i)+1.5
```

在这里，`Complex{1,2}` 不是“构造函数的返回值”，因为构造函数是没有返回值的。这个语法的正确解读应为：`Complex(1,2)` 是构造函数创建的临时对象，这和 `int` 转换成 `double` 过程中创建了一个临时数据是同样的道理。

至于 `1.5`，它也会不动声色地隐式类型转换为 `Complex{1.5,0}`，于是我们不需要为了 `Complex+double` 专门写一个运算符重载，只需要让它自己隐式类型转换就够了。

## explicit

隐式类型转换虽然方便，但是未必总能合我们的意。有些时候，隐式类型转换不能满足我们的需要，所以我们需要显式类型转换。比如说，当我们试图用 `std::cout` 输出一个 `char*` 指针的值时，实际的输出是一个字符串。为了输出地址值，我们应当用显式类型转换来实现：

```
char *pch = nullptr;
std::cout << static_cast<void*>(pch); // 显式类型转换为 void*
```

还有些时候，一些我们本来不想进行，但却被编译器自动执行的隐式类型转换可能会引发出乎意料的错误。在不久前我们就看过了一个这样的例子：

```
std::string str3 {33, '='}; // 试图通过统一初始化调用 (size_type, CharT) 版本
std::cout << str3; // 但输出的内容却是 !=
```

这是因为，`int` 类型的 `33` 在这里被隐式类型转换成了 `char` 类型的 `'='`，然后与这里的 `'='` 一并构成 `std::initializer_list<char>`。于是实际调用的是这个构造函数<sup>33</sup>：

```
basic_string(
    std::initializer_list<CharT> ilist,
    const Allocator& alloc = Allocator()
); // 接收一个 std::initializer_list<CharT> 参数
```

然而我们的本意是调用这个构造函数：

```
basic_string(
    size_type count,
    CharT ch,
    const Allocator& alloc = Allocator()
);
```

回到这个问题本身，我们发现它的根源就在于：在编译的过程中，编译器选择了我们意料之外的**隐式类型转换方案**。这是隐式类型转换的一个致命缺点。

我们在写复数类的时候也可能遭遇类似的情境，我们不想进行某种类型转换，但编译器却自动进行了隐式类型转换。为了防止隐式类型转换的发生，同时又保留类型转换的功能，我们可以使用关键字 `explicit`。

`explicit` 说明符的作用在于，限制某种类型转换必须以显式类型转换(Explicit type conversion)的方式进行。如果我们想设计一个 `valarri` 向 `bool` 的转换函数，用 `true/false` 来表示这个 `valarri` 中是否有数据，那么我们可以这样写：

```
// 声明&定义
public:
    explicit operator bool() const { return _size; }
    // valarri 到 bool 的自定义转换函数，说明此数组中是否有数据，是常成员函数
```

读者需要搞清楚 `explicit` 的作用。它的作用不是“规定在这个函数中不能隐式类型转换”或者“这个类不能隐式类型转换”等等——这些都不对。它的作用是：规定了 `valarri` 到 `bool` 的类型转换过程必须显式。至于其它的任何类型转换过程，都不受此影响。

比如说吧，我们在函数体内就返回了一个 `std::size_t` 类型的 `_size`，但是期待的返回类型是 `bool`，这时会发生什么呢？`_size` 会被隐式类型转换为 `bool`<sup>34</sup>，然后作为返回值。所以说，声明为 `explicit` 的函数内部仍然可以进行其它类型之间的类型转换，这是不受 `explicit` 影响的。

<sup>33</sup> 此处的 `CharT` 正是 `char`, `size_type` 正是 `std::size_t`, 读者可以用 `std::is_same` 验证之。

<sup>34</sup> 在转换过程中，`std::size_t{0}` 会被转换为 `false`，其它值一律被转换为 `true`

如果我们要使用这个类型转换函数，就必须显式类型转换，或者用于条件、循环结构的判断句，或者用于逻辑表达式<sup>35</sup>。

```
void fun(bool) {}
int main() {
    valarri arr{};
    valarri a; // 将调用默认构造函数
    if (!a) // a没有重载!运算符，所以内置!运算符会期待a转换成bool类型
        std::cout << bool(a); // 可以进行显式类型转换
    if (a) // if期待a转换成bool类型，这里可以转换
        fun(a); // 试图在参数传递时使用隐式类型转换
    //error: cannot convert 'valarri' to 'bool'
}
```

## valarri 的初步实现代码

最后，我们可以把前五节已有的知识串联起来，完善一下我们的 valarri 代码。

下面是我修改完善后的完整代码，只有头文件和定义部分。读者可以自行写一个主函数并测试其主要功能。

相比于之前断断续续写的代码，这里的內容更详细，更具体，改动了部分代码（比如，在构造函数中添加 size 超过 MAX\_SIZE 的检测，把循环变量改成统一的 std::size\_t 类，还把依靠 \_arr.p()[] 成员实现的部分用 operator[]() 实现），新增了 resize 和 swap 成员函数，对注释部分也做了相当大的调整。

代码 8.1: Header.h

```
#pragma once
#include <initializer_list>
#include <iostream>
#include <vector> // 使用 vector 必备的库
using vecint = std::vector<int>; // 别名声明
class valarri; // valarri 类声明
void input_clear(std::istream& = {std::cin}); // input_clear 函数声明
class valarri { // valarri 类定义
private: // 私有成员部分，注意成员变量定义的顺序
    static std::size_t _object_number; // 已有 valarri 对象的数量
    static const std::size_t MAX_SIZE {100000}; // 可以允许的最大容量
    std::size_t _cap {0}; // 当前动态分配的容量， 默认初值为 0
    std::size_t _size {0}; // 当前实际存储的数据量， 默认初值为 0
    class Arr{ // 内嵌的 Arr 类
private:
    int *_p; // 私有指针成员 _p， 用来指向申请的动态内存空间
public:
    Arr(int *ptr) : _p {ptr} {} // 构造函数， 接收一个指针初始化
    int*& p() { return _p; } // 返回类型是“对指针的引用”， 其值为 _p
```

<sup>35</sup> 在条件和循环结构中，比如 `if(std::cin)`，虽然这里发生的也是隐式类型转换，但编译器是允许的，因为这里发生的隐式类型转换不会有任何二义性的可能。同理，逻辑运算符 `&&`, `||` 和 `!`（特指内置的运算符，而不是用户重载的）在接收操作数时也可以发生隐式类型转换，这种隐式类型转换是允许的。总之，`explicit` 的限制范围是个很复杂的问题。

```

    const int* p()const { return _p; } // 返回类型是“指向常量的指针”，其值为_p
    }_arr {new int[_cap]}; // 作为一个中介对象存在，是访问_p成员的窗口
public: // 公有成员部分
    valarri() { _object_number++; } // 默认构造函数，只进行计数器自增
    valarri(const std::initializer_list<int>&); // 接收一个初始化表作为参数
    valarri(int, std::size_t); // 将<参数1>重复<参数2>次
    valarri(const valarri&); // 拷贝构造函数
    ~valarri(); // 析构函数
    valarri(const vecint&); // vecint到valarri的转换构造函数
    operator vecint()const; // valarri到vecint的自定义转换函数
    explicit operator bool()const { return _size; }
    // valarri到bool的自定义转换函数，说明此数组中是否有数据，是常成员函数
    valarri& operator=(const valarri&); // 直接赋值运算符
    valarri operator+(const valarri&)const; // 可以与数组相加
    valarri operator+(int)const; // 可以与数相加
    friend valarri operator+(int, const valarri&); // 可以以数加之，友元
    valarri& operator+=(const valarri&); // 可以加之以数组
    valarri& operator+=(int); // 可以加之以数
    friend std::istream& operator>>(std::istream&, valarri&); // 需iostream库
    int& operator[](std::size_t i) { return _arr.p()[i]; } // 非常成员函数
    int operator[](std::size_t i)const { return _arr.p()[i]; } // 常成员函数
    void swap(valarri&); // 完全交换两个valarri对象的成员数据
    void resize(std::size_t, int = {0}); // 改变数据容量，如果扩大，将补<参数2>
    std::size_t size()const { return _size; } // 返回当前的存储量大小，常成员函数
    static std::size_t object_number() { return _object_number; }
    // 静态成员函数不能是常成员函数
    int max()const; // 最大值
    int min()const; // 最小值
    int sum()const; // 总和
};

inline std::size_t valarri::_object_number {0};
// 静态非常成员变量要在外部定义；如果定义在头文件中，需要有inline

```

代码 8.2: Definition.cpp

```

#include "Header.h"
#include <algorithm>
valarri::valarri(const std::initializer_list<int>& ilist)
    : _cap {std::min(MAX_SIZE, ilist.size())} // _cap 的初值要取二者最小值
{
    std::size_t counter {0}; // 长度计数器，用来防止超过MAX_SIZE
    for (int x : ilist) { // 范围for循环，遍历ilist中的每个数
        operator[](_size++) = x; // 注意自增是前缀还是后缀，请读者自行分析
        if (++counter >= MAX_SIZE) // 说明达到MAX_SIZE了
            break; // 直接退出for循环
    }
    _object_number++; // 对象计数器自增
}

```

```

valarri::valarri(int n, std::size_t size)
    : _cap {std::min(MAX_SIZE, size)} // _cap 的初值要取二者最小值
{
    for (_size < _cap; _size++) // 通过循环赋值，其中 _size 已经通过默认初始置 0
        operator[](_size) = n;
    _object_number++; // 计数器自增
}
valarri::valarri(const valarri &a) // 拷贝构造函数应当进行深拷贝，切记！
    : _cap {a._cap},
    _size {a._size}
{ // 这里不用防 _cap 超过 MAX_SIZE，因为其它部分共同保证了 a._cap 不会超过 MAX_SIZE
    for (std::size_t i = 0; i < _size; i++)
        operator[](i) = a[i]; // 拷贝每个值，而不是拷贝地址
    _object_number++; // 计数器自增
}
valarri::~valarri() {
    _object_number--; // 计数器自减
    delete[] _arr.p(); // 回收 _arr.p() 指向的动态内存空间，注意必须要用 delete[] 搭配
}
valarri::valarri(const vecint &vec) // vecint 到 valarri 的转换构造函数
    : _cap {vec.size()}, // valarri 的成员对 vec 私有成员无访问权限，所以要用其公有成员
    _size {vec.size()}
{
    for (std::size_t i = 0; i < _size; i++)
        operator[](i) = vec[i]; // vecint 也有重载下标运算符，并且有常成员函数的重载
    _object_number++; // 计数器自增
}
valarri::operator vecint() const { // valarri 到 vecint 的自定义转换函数
    vecint vec(_size); // 调用了 vecint 的构造函数，直接指定 vec 的大小
    for (std::size_t i = 0; i < _size; i++)
        vec[i] = operator[](i); // 逐个赋值，不要用 push_back 成员，反复重分配太浪费
    return vec;
}
valarri& valarri::operator=(const valarri &a) { // 直接赋值运算符
    if (&a == this) // 防范自我赋值
        return *this;
    if (_cap < a._size) { // 如果现有容量装不下 a 中的所有数据，那就要扩容
        delete[] _arr.p(); // 别忘了先回收现在指向的内存空间！！
        _cap = a._size; // 希望分配一个大小为 a._size 的内存空间
        _arr.p() = new int[_cap]; // 分配动态内存空间
    } // 这里也不用防范 _cap 超过 MAX_SIZE，原因同上
    for (_size = 0; _size < a._size; _size++)
        operator[](_size) = a[_size];
    return *this; // 返回 *this
}
valarri valarri::operator+(const valarri &a) const {

```

```
valarri v(0,std::min(_cap,a._cap)); //用构造函数为v初始化，代码重用
for (std::size_t i=0; i < v._size; i++)
    v[i] = operator[](i) + a[i];
return v;
}
valarri valarri::operator+(int n) const {
    valarri v {*this}; //用拷贝构造函数为v初始化，也是代码重用
    for (std::size_t i = 0; i < v._size; i++)
        v[i] = operator[](i) + n;
    return v;
}
valarri operator+(int n,const valarri &a) { //这是非成员函数，它是valarri的友元
    valarri v {a}; //同样是拷贝构造函数
    for (std::size_t i = 0; i < v._size; i++)
        v[i] = a[i] + n;
    return v;
}
valarri& valarri::operator+=(const valarri &a) { //加赋值运算符
    return *this = *this + a; //代码重用，加法与赋值连用
}
valarri& valarri::operator+=(int n) { //同上
    return *this = *this + n;
}
std::istream& operator>>(std::istream &in, valarri &a) {
    a._size = 0; //因为它是valarri的友元，所以可以访问valarri的私有成员
    while (in && a._size < a.MAX_SIZE) //要保证in正常且a._size没有达到最大容量
        in >> a[a._size++]; //注意++是后置的，不是前置
    if (!in) //检测一下in的状态，保证其状态正常
        input_clear(in); //这次就不是用默认参数，而是用in了
    return in; //返回in，以便连续输入
}
void valarri::swap(valarri &a) { //完全交换两个valarri对象的成员数据
    std::swap(_cap, a._cap); //交换_cap对象
    std::swap(_size, a._size); //交换_size对象
    std::swap(_arr.p(), a._arr.p()); //交换_arr._p对象，这是通过_arr.p()实现的
}
void valarri::resize(std::size_t size, int value) { //改变数据容量
    if (size > MAX_SIZE) //防止size超过MAX_SIZE
        size = MAX_SIZE;
    valarri v(value, size); //注意用圆括号
    for (std::size_t i = 0; i < std::min(_size, v._size); i++) //注意范围
        v[i] = operator[](i);
    swap(v); //将此对象与v交换，这样v会指向旧的内存空间并在析构时销毁
}
int valarri::max() const { //返回值是数组当前的最大值
    int maximum {operator[](0)};
}
```

```

for (std::size_t i = 1; i < _size; i++) // 注意循环的结束条件用 _size, 下同
    if (maximum < operator[](i))
        maximum = operator[](i);
return maximum;
}

int valarri::min() const { // 返回值是数组当前的最小值
    int minimum {operator[](0)};
    for (std::size_t i = 1; i < _size; i++)
        if (minimum > operator[](i))
            minimum = operator[](i);
    return minimum;
}

int valarri::sum() const { // 返回值是当前数组中有效数据的总和
    int summation {0}; // 先置为0
    for (std::size_t i = 0; i < _size; i++)
        summation += operator[](i); // 然后把每个数都加到 summation 上
    return summation;
}

void input_clear(std::istream &in) { // input_clear 函数定义
    in.clear();
    while (in.get() != '\n')
        continue;
}

```

## 8.6 复合类型与对象

前面五节，我们主要都在关注类与对象的内部特性。本节，我们转向它的外部特性来进行介绍。一个封装良好的类可以把内部特性全都隐藏起来，我们可以不管那些不为人知的细节，只要用它提供的公有成员来完成我们需要的工作就行了。

### 此类的对象，彼类的成员

如果读者有这个闲情雅致，可以再扫视一遍第六章和本章前五节的内容；如果没有时间，也请回看一遍目录。我们会发现，自定义类型没有那么神秘，它只不过是各种基本类型（Fundamental types）和复合类型（Compound types）组合而成的产物。

- 枚举，它是基于某种整型来实现的。我们可以人为改变它的枚举基。
- 结构体，它是一种无机的数据整合体<sup>36</sup>，把 **int**, **double** 等各种数据类型堆砌到同一个单元中。
- 联合体，它也是一种无机的数据整合体，可以用于特殊场合，以减少内存开支。
- 类。它把成员变量和其它细节封装起来，我们只需要使用它的公有成员，而不需要在乎内部构造，方便简单。但实际上，这个类的内部还是那些 **int**, **double**，还有数组、指针之类的数据，而这些数据的操作，正是我们在前五章中讲到的内容。

<sup>36</sup>**struct** 和 **class** 没有本质区别。严格说来，这里的“结构”指的是类似 C 语言中的结构体，它是没有成员函数，且所有成员变量均为公有成员的 **struct** 和 **class**。

我们还说过，在 C++ 中，类（Class）与类型（Type）常常是对等概念。我们可以说 `valarri` 是一个类型，也可以说 `int` 是一个类。C++ 为我们提供了很多功能，比如运算符重载，来让我们在一定程度上消除内置类型和自定义类型的区别。事实证明，C++ 在这方面的努力还是很成功的。

我们在类的定义中也可以把另一个类的对象，作为该类的成员——这就好像是把某个内置类型的变量作为该类的成员一样，是非常自然的事情。如果我们要定义一个类用来表示人的身份信息，那么让我们思考一下我们都可以用哪些成员：

- 名字。我们可以用字符串 `char[N]` 来表示。但是限于名字长短不一，如果 `N` 太大，那就会浪费内存空间；如果 `N` 太小，那就有不能正常完成之虞。更好的选择是用 `std::string`，它可以动态地管理内存，而且不需要我们去操心那些恼人的细节，诸如动态空间回收（因为已经写在 `std::string` 的析构函数中了）。
- 编号，比如身份证号。我们可以用整型来表示它们，但是请注意，如果编号过长，这个值可能无法直接用整型来表示，我们可以改换策略，用字符串类型来表示。如果编号的位数是固定的，那就更好，直接用 `char[N]` 就可以；如果编号的位数是不固定的，那么 `N` 就要取其最大可能值。
- 性别。我们直接以 `bool` 为枚举基，设置一个枚举类型就可以了。
- 身高、体重。这些可能是浮点数，我们用 `float` 和 `double` 就行。
- 年龄。它是非负的整数，所以可以用 `unsigned` 来表示<sup>37</sup>。
- .....

然后我们可以写这样一个类：

```
class PersonalInfo {
private:
    const std::string name; //name是std::string类的对象，PersonalInfo类的成员
    const enum : bool {male, female} sex; //这是比较省事的写法（代码多了显得乱）
    double height;
    double weight;
    unsigned age;
    //...更多
public:
    //...构造函数，以及其他功能的实现
};
```

我们可以看到，这里的 `name` 既是 `std::string` 类的对象，又是 `PersonalInfo` 的成员。

同样的道理，`sex` 既是一个（匿名的）枚举类的对象，又是 `PersonalInfo` 的成员。`height` 和 `weight` 是 `double` 类的对象；`age` 是 `unsigned` 类的对象。总而言之，我们看到，每个类的对象都可以与其它类的其它对象一起，共同构成新的类。所以此类的对象，也可以是彼类的成员。

这些类的对象之间还可以继续组合，比如说 `PersonalInfo` 对象可以作为 `Company` 的成员，它的现实意义就是一个公司的雇员体系；除了这个雇员体系外，公司可能还有资金系统，物流系统等等。就这样，从小单元到大整体，面向对象概念为我们描述这个精彩的世界提供了无穷的可能性。

<sup>37</sup>其实用 `unsigned char` 存储这个值，然后在需要的时候类型转换为整型，这才是最节省内存空间的方法；但是吧，我们没必要在节省内存空间这个问题上太强迫症了，毕竟现在已经不是那个内存空间寸土寸金的年代了。

## 示例：`std::string` 对象数组

我们可以定义基本数据类型的数组，还可以定义复合数据类型的数组（二维数组，指针数组等）。对于自定义类型来说也是如此。我们可以定义 `std::valarray` 对象的数组，或者是 `std::string` 类型的数组。

```
std::string str[5]{
    "Bjarne Stroustrup",
    "Donald Ervin Knuth",
    "Alexander Alexandrovich Stepanov",
    "Alan Mathison Turing",
    "Claude Elwood Shannon"
}; // 定义由5个std::string对象构成的数组str
```

如果要访问它的单个元素，我们需要怎么做呢？很简单，用下标运算符就行了。

```
std::cout << str[0] << std::endl; // 将输出 Bjarne Stroustrup
std::cout << *(str+4) << std::endl; // 将输出 Claude Elwood Shannon
```

请读者关注它们的类型。既然 `str` 是一个 `std::string[5]` 类型，那么这个数组就可以在涉及加减法的场合下将隐式类型转换为指针<sup>38</sup>。而对于指针来说，`str+4` 就意味着“第五个数组元素的地址”（别忘了，第一个元素是 `str+0`，所以推算下来 `str+4` 就是第五个元素）。再取内容，得到的就是 `std::string` 类型的结果了。于是重载了这个输出的 `std::cout` 自然可以输出对应的结果。

我们还可以输出单个字符，这是因为 `std::string` 类重载了下标运算符。

```
std::cout << str[1][0]; // 将输出 D
std::cout << (*str)[2]; // 将输出 a
```

我们还是来分析它的类型。`str` 的类型是 `std::string[5]` 自不必说，那么如同我们刚才分析的那样，无论 `str[1]` 还是 `(*str)` 都是 `std::string` 类型。而 `str[1][0]` 的后一个下标运算与前一个下标运算是有着根本不同的。后一个下标其实是成员函数 `std::string::operator[](std::size_t)`。

我们可以用 `typeid` 或 `std::is_same` 来判断它们的类型。（不过，GCC 编译器的 `typeid` 信息太难懂了，所以我们还是用 `std::is_same` 吧）

```
std::cout << std::is_same<
    decltype(str),
    std::string[5]
>::value << std::endl << std::is_same<
    decltype(str[0]),
    std::string& // 注意数组下标/取内容运算的返回值是引用
>::value << std::endl << std::is_same<
    decltype(str[0][0]),
    char& // 注意 std::string::operator[](std::size_t) 的返回值是引用
>::value;
```

输出结果都是 1，我就不再废话了。

## `valarray` 与动态内存分配

之前我们写的 `valarray` 仅供教学用途，我们可以通过自己写出一个简易 `valarray` 的方式，初步了解 `std::valarray` 的内部机制。但是在实际编程中我们还是尽量用 `std::valarray`，而不是我们

<sup>38</sup>顺便一说，虽然 `std::string` 类型是自定义类型，但是 `std::string` 有关的数组和指针类型都不是自定义类型。它们只是“复合类型”，属于数组或指针家族。

自己写的版本。原因也很简单，毕竟它支持的功能不仅更完善，而且更稳定。

在实际应用中，我们可能也会遇到需要动态内存分配的情况——不只是数组本身通过动态内存分配来实现可变大小，我们也可能需要“若干个这样的数组”——具体需要多少个，也是不确定的。

```
std::valarray<int> *p;
unsigned n;
std::cin >> n; // 输入个数，然后我们就创建n个std::valarray<int>对象
p = new std::valarray<int>;
//... 使用
delete p; // 别忘了哦
```

这个过程看似简单，但是内部是非常复杂的。在我们分配一段动态内存的时候，每个 `std::valarray<int>` 也都还有一个指针，等待分配动态内存，或者是已经分配到了动态内存；在我们改变数组内容的时候，就可能发生动态内存的重新分配。当我们释放动态内存的时候，这些对象的析构函数也会被调用，从而自动回收动态内存。这些内部细节全部对外不公开，这既能防止其内容被外界破坏，又能减少我们的工作量，这就是封装的优点。

我们发现，把动态内存管理的工作交给 `std::valarray`, `std::vector` 等类（类模版）来管理，显然要好过我们自己写 `new[]` 然后不小心忘了 `delete[]`。所以我们还可以更进一步，把“动态数组的动态数组”也交给这些类来管理。

```
std::valarray<std::valarray<int>> arr;
```

读者现在未必能够理解这段代码，等到我们讲到第十一章，读者就可以很容易地理解了。

## 智能指针简介

智能指针（Smart pointer）是为了解决我们“忘记回收动态内存”的问题而出现的。智能指针的一个对象就充当了一个指针。与普通指针最大的区别在于，在它的生存期结束时，智能指针将通过析构函数来回收动态内存空间——这样我们就不需要为了回收动态内存而操心。

我们会在第十一章讲解，并自己写一个简单的智能指针。请读者尽情期待吧！

## 8.7 实操：简单的 string 类

我们在本节的目的是：参照 `std::string` 类的功能，仿制一个 `string` 类。这个仿制版本不需要多么专业，也不需要多么复杂的功能。我们的目的在于学习与掌握，而不是拿出去实际使用。

我们先来看看 `std::string` 有哪些常用功能——我们可以在 [std::basic\\_string - cppreference.com](http://std::basic_string - cppreference.com) 中查到——知己知彼嘛。但是出于整理的目的，还是请允许我在书中花上一些篇幅来介绍。

## 功能简介

`std::string` 的全称是 `std::basic_string<char, std::char_traits<char>, std::allocator<char>>`。是不是很吓人？不过我们现在不用研究什么 `char_traits` 或者 `allocator`，只需要盯着我们的目的——仿制一个 `string` 类，就足够了。所以我们就用简单一点的方法，不用什么所谓动态内存分配器之类的东西，直接用 `new[]/delete[]`，诸如此类。

`std::string` 的构造函数非常多样：

- 默认构造函数将建立一个空字符串。
- 有的构造函数是建立 `count` 个 `ch` 字符构成的串。

- 有的构造函数是在另一个字符串对象 `other` 中截取 `pos` 开始的 `count` 个字符。
- 有的构造函数是以一个 `char*` 字符串的内容为内容建立起来的。
- 有的构造函数是用一个列表来初始化的，我们在前面讲 `valarri` 时也介绍过。
- 还有各种拷贝构造函数/移动构造函数<sup>39</sup>。
- .....

种种构造函数不一而足，我们当然也没有必要全部实现，拣一些常用的就行了。

`std::string` 的析构函数只有一个（况且反正析构函数不能重载），目的就是回收这个对象分配的动态内存。

`at` 和 `operator[]` 成员函数都可以用来访问内部数据。不同的在于，`at` 有更完善的异常检测<sup>40</sup>，对于下标越界行为有所防范，但 `operator[]` 出于方便使用的目的，就没有这方面的功能；但这也就导致它相对危险，不适合真的用在类的内部（我们此前写的 `valarri` 类就像玩似的）。

`assign` 和 `operator=` 成员函数都可以用来为一个 `std::string` 对象赋值，不过 `assign` 成员函数才是最好用的，尤其是在类的内部<sup>41</sup>。试想，赋值运算符只能接收一个参数，虽然它用起来很方便，但是局限性未免太大了点。而 `assign` 这种成员函数才真正适合用来实现内部功能。

`empty` 成员函数用来返回一个 `bool` 数据，表示这个对象是不是空字符串。我们在前面用自定义转换函数 `operator bool` 实现了这个功能，但它不是 STL 容器及 `std::string` 中的标准做法。

`size` 和 `length` 成员函数的作用完全等同，都是返回这个对象的字符串长度。<sup>42</sup>需要注意的是，我们应当有专门的成员变量来存储这个长度值，以此保证它的时间复杂度是常数级别<sup>43</sup>。至于延迟标记，我们就不再需要了，因为大多数成员函数都必须知道“长度”这个数据，所以我们几乎不会遇到“整个生存期内不需要计算长度”的情况。

`capacity` 成员函数返回这个对象的容量——其实也就是动态内存的长度。注意，“容量”与“字符串长度”是两个概念，不能划等号！

`reserve` 成员函数用于改变这个对象的容量，它接收一个参数。如果参数大于现有容量，那么就扩容；如果参数小于容量而大于等于字符串长度，那么就重新分配一个稍小的内存空间（相当于节省了部分不需要的内存空间）；如果参数小于字符串长度，那么就只把容量缩小到与 `size()` 相等，从而保证不会损失内容。

`resize` 函数的作用是，改变内容的长度，但不会影响容量。如果 `resize` 想要把内容缩短，那么内容将会截尾；如果它把内容加长，那么内容将会填充 '`\0`' 或者我们指定的字符 `ch`。

`append` 和 `+=` 成员函数都可以用来在此对象后面继续添加内容；但是如前所述，运算符版本的作用还是比较有限，所以我们在类内部还是用 `append` 吧。

`insert` 和 `erase` 成员函数有大量的重载，其中 `insert` 用于在指定位置插入字符或字符串，而 `erase` 函数相反，用于在特定范围内删除字符或字符串。其中部分重载会涉及到迭代器的相关知识，我们避开就好。另外，`erase` 函数的作用仅限于单纯的改变内容，不涉及动态内存的重新分配。

`clear` 成员函数用于清理字符串内容，相当于把它变成空字符串。但是请注意，这个函数和刚才的 `erase` 很像，它不会改变容量，只是单纯改变内容及长度。

`copy` 成员函数的作用是把本字符串的内容复制一部分到指定的 `char*` 字符串中，其作用有点像 `cstring` 库中的 `std::memcpy`。

<sup>39</sup>泛讲篇不讲移动构造，详见精讲篇。

<sup>40</sup>关于异常，我们将在第十二章和精讲篇中讲解。

<sup>41</sup>把麻烦留在定义的时候，把方便留在使用的时候，这也是面向对象编程的理念。

<sup>42</sup>其中的 `size` 能与 STL 成员函数相兼容，而 `length` 与字符串原本的“长度”概念近同。或许是因此才有两个名称不同但实际作用一致的成员函数吧。

<sup>43</sup>这里牵扯到时空复杂度的概念。简单来说，时空复杂度是衡量算法/数据结构性能的重要指标。常数复杂度意味着某个函数的运行时间与这个字符串的长度无关，一般来说这是最佳的时间复杂度。

`swap` 成员函数的作用是把此对象与另一个对象的内容完全交换，这算是我们的老朋友了。

`find` 成员函数是一个查找函数。丰富而强大的查找功能是 `std::string` 的一大特色。通过 `find` 函数，我们可以在一个对象中查找其它字符、字符串或对象（其实还是在字符串中查找特定字符串）。`std::string` 还提供了 `rfind` 成员函数，它就是从后向前找而已。至于 `find_first_of` 这些，各有各的作用，太多了，我就不一个个说了。

`compare` 成员函数和六个重载比较运算符（非成员函数）的作用是进行字典序比较。对于 `compare` 来说，如果此对象应排在彼对象之前，那么返回值为负数（具体多少，那就不一定了）；如果此对象应排在彼对象之后，那么返回值为正数；如果它们完全相同，那么返回值为 0。

还有一些其它类的成员函数与 `std::string` 类有着密不可分的关系。比如说 `std::istream` 类重载了 `>>` 运算符，可以接收外界到 `std::string` 的输入；`std::ostream` 类重载了 `<<` 运算符，可以直接输出 `std::string` 对象。

好了，关于 `std::string` 的功能，我们先介绍到这里。

实际上 `std::string` 的功能远比这些要多，但是我们没有必要做这么多（单纯只是刚才那一页，就已经是不小的工作量了），况且这里还有太多我们没学过的知识。

读者看完了上述内容之后可能已经感到头疼了。但是不必担心，我们不需要把它“背下来”，只需要在将来我们写的时候，不时回来翻阅，或者到 `cppreference` 中查阅一下，看看它的功能是什么，注意可能的问题，这就足够了。

## 规划

凡事预则立，不预则废。规划好这个类的定义部分，把需要的函数声明出来，然后再考虑具体的成员函数定义，这样比较好。

### 私有成员部分

我们可以借用之前定义 `valarri` 的思路，在私有成员部分分别定义两个 `std::size_t` 变量，名为 `_cap` 和 `_size`，记录本对象的字符串容量和长度。而 `MAX_SIZE`，其实不是必需的，这里就不用了。

至于指针，我们也可以沿用嵌套类 `Arr` 的思路来进行。另一种方法是私有继承——我们等到下一节再讲。

读者需要注意它们的声明顺序，因为在初值列中，初始化的顺序会取决于成员的声明顺序。所以对于有依赖关系的 `_arr` 和 `_cap`，要特别注意把 `_cap` 定义在前面。至于没有任何依赖关系的 `_size`，放在哪里都行；但我们还是倾向于把它们安排得更有条理一些，而不是乱放。以下是我本人的习惯：

```
class string { //class 的默认成员访问权限是 private，所以 private 可省略
    static constexpr std::size_t npos = -1; //串尾指示器
    std::size_t _cap {0}; //容量， 默认初值为 0
    std::size_t _size {0}; //长度， 默认初值为 0
    class Arr {
        char *_p;
    public:
        Arr(char* ptr = {nullptr}) : _p {ptr} {} //Arr 的构造函数
        char*& p() { return _p; } //当 arr 不是常量时调用
        const char* p() const { return _p; } //当 arr 是常量时调用
    } _arr {new char[_cap]}; //象征指针的成员， 默认用 new char[_cap] 初始化
    void realloc(std::size_t cap, bool copy = true) { //重分配内存函数
        char *tmp = _arr.p(); //临时指针， 用以暂存当前内存地址
    }
}
```

```

    _arr.p() = new char[cap]; // 分配新的内存地址
    if (copy) // 说明这个函数需要进行内容复制
        for (std::size_t i = 0; i < std::min(cap, _size); i++)
            _arr.p()[i] = tmp[i]; // 转移内容
    delete[] tmp; // 回收原来的内存地址
    _cap = cap; // 更新 _cap 的值
    if (_size > _cap)
        _size = _cap; // 保证 _size 不能大于 _cap
}
public:
    // ... 待补充
};

```

这里有一个特殊的静态常成员 `npos`, 它负责担任串尾指示器。举个实际例子你就懂了: 假如我们要用 `find` 成员函数查找某个字符串中的字符, 找到了就返回那个位置的下标<sup>44</sup>。问题来了——没找到的话返回什么? 返回 `0` 吗? 不行, 那就和“在下标 `0` 处找到”的意思冲突了。同理, 返回任何一个正整数都有可能都会造成冲突, 所以我们需要一个特殊值来返回。

这里还有一个私有成员函数 `realloc`, 它比较特别, 是辅助我们进行内存重分配的。简言之, 我们在定义其它成员函数时经常因为内存空间不够, 而需要进行内存重分配。`realloc` 可以帮助我们直接完成这项工作, 我们不再需要一遍一遍地写动态内存分配和复制数据的代码, 而是直接调用 `realloc` 函数, 省时省力, 经松解决! 此外, 因为 `realloc` 是私有成员函数, 不会被外界滥用, 这也保证了对象的安全。

`realloc` 函数有一个带默认值 `true` 的参数 `copy`, 它是一个标记, 表明我们在重分配内存时是否需要复制数据。什么意思呢? 我们有的时候调用 `realloc` 是为了在保留内容的前提下重新分配内存空间, 这时我们就需要在分配动态内存时把数据也一并复制到新的区域内; 但有的时候调用 `realloc` 只是为了获得一个更大的内存空间, 不需要在乎内容, 所以这种情况下复制数据就是浪费时间。因此我们需要在二者之间进行选择, 所以就需要一个 `bool` 型参数来帮助我们应对不同需求。

`npos` 就是这样的一个值, 这里的 `-1` 只是个象征性的值, 它可以用来表示两种含义: 一种是字符串末尾, 一种是下标错误。`find` 函数使用的正是第二种含义。

至于定义——为什么在这里初始化时不用花括号而是改用等号了呢? 这是因为 `std::size_t` 不能接收 `-1` 这个值。我们的实际目的是要通过有符号到无符号转换<sup>45</sup>, 把 `-1` 变成机器允许的最大无符号值——读者可能觉得有点头大了, 看不懂也不要紧啦, 直接抄代码就行。

下面我就在 `public` 部分中补充成员函数。

## 公有成员部分

先来看构造函数。我们不用把那十多个重载全都写出来; 挑几个, 意思一下就行了。虽然形参名是非必须的, 但是为了方便阅读和解释代码, 我们还是适当地把形参名写上比较好。

```

public:
    explicit string() {} // 默认构造函数, 定义成 explicit 防止被误用
    string(std::size_t count, char ch); // 重复 count 个 ch
    string(const string &other, std::size_t pos = {0},
           std::size_t count = {npos}); // 在 other 的 pos 位置起接收 count 个字符
    string(const char *s); // 用 s 字符串初始化

```

<sup>44</sup>我们可以理解成字符串的下标, 或者理解成该字符到首字符的距离, 都可以。下标值一般是 `std::size_t` 类型的, 表示非负整数。

<sup>45</sup>我们到精讲篇中再谈。

```
string(const char *s, std::size_t count); //用s字符串的前count个字符初始化
string(std::initializer_list<char>); //需要头文件initializer_list
```

其中第三个函数有默认值 `0` 和 `npos`, 如果都省略不写的话, 它就相当于一个拷贝构造函数。`npos` 在这里充当串尾指示器, 表明“`other` 有多长, 我就要多长”——总之, `npos` 不意味着一个“特定的长度”, 它有它的特殊意义。在实际操作的时候, 我们应当把“`pos` 情况”单独考虑。

至于析构函数, 就没什么多说的了。反正它既没有返回值, 又不能接收参数, 还不能重载, 所以只有这一种写法:

```
public:
~string(); //析构函数
```

接下来我们声明 `at` 和 `operator=`。因为我们尚未学习异常处理, 所以我们就不管 `at` 函数的那些细节。对于越界的下标, 统一把下标改为 `size()-1`; 而 `operator=` 不作越界检测。

其中的重载下标运算符非常简单, 直接定义在此处。

```
public:
char& at(std::size_t pos); //返回pos位置的字符
const char& at(std::size_t pos) const; //常成员函数版本
char& operator[](std::size_t pos) { return _arr.p()[pos]; } //下标运算符
const char& operator[](std::size_t pos) const { return _arr.p()[pos]; }
```

接下来我们声明 `assign` 和 `operator=`。`assign` 成员函数也有很多重载, 所幸它们和构造函数的要求很相像, 我们甚至可以考虑用构造函数加 `swap` 配合来节省代码工作量(我们曾在 `valarray::resize` 中用过这类技俩)。

```
public:
string& assign(std::size_t count, char ch); //赋值重复count个ch
string& assign(const string &str, std::size_t pos = {0},
               std::size_t count = {npos}); //在str的pos位置起接收count个字符
string& assign(const char *s, std::size_t count); //接收s的前count个字符
string& assign(std::initializer_list<char>); //用列表赋值
```

`operator=` 只需要在 `assign` 的基础上, 做一些方便使用的简化版本就可以了。

```
public:
string& operator=(const string &str) { return assign(str); }
string& operator=(const char *s) { return assign(s, std::strlen(s)); }
//其中std::strlen需要头文件cstring
string& operator=(char ch) { return assign(1, ch); }
string& operator=(std::initializer_list<char> ilist)
{ return assign(ilist); }
```

`empty`, `size` 和 `length` 成员函数的实现都很简单, 也不太需要讲解。我们可以直接在类内完成这些定义。

```
public:
bool empty() const { return _size; } //用到了隐式类型转换
std::size_t size() const { return _size; }
std::size_t length() const { return _size; }
```

注意它们都要是常成员函数, 这样无论对象本身是不是常量, 都可以调用这些成员函数。

`capacity` 也是同理, 只是它返回的是 `_cap` 成员变量。

```
std::size_t capacity() const { return _cap; }
```

接下来声明 `reserve` 和 `resize`。我们先不管它们的细节，只声明出来就行。

```
public:
    void reserve(std::size_t new_cap); // 改变容量到new_cap，除非影响内容
    void resize(std::size_t count, char ch = {'\0'}); // 改变内容长度
```

在 `std::string::resize` 的实际定义是把 `(std::size_t)` 和 `(std::size_t, char)` 作为重载出现的。在这里我们用默认值的方法，把它们认定成同一个函数，这样比较省事。

`append` 函数也有许多重载，我们挑选其中几个就行。

```
public:
    string& append(std::size_t count, char ch);
    string& append(string &str, std::size_t pos = {0},
                  std::size_t count = {npos});
    string& append(const char *s, std::size_t count);
    string& append(std::initializer_list<char>);
```

然后再 `append` 的基础上，做一些方便使用的简化版，这就是 `operator+=` 运算符了。

```
public:
    string& operator+=(char ch) { return append(1, ch); }
    string& operator+=(const string &str) { return append(str); }
    string& operator+=(const char *s) { return append(s, std::strlen(s)); }
    string& operator+=(std::initializer_list<char> ilist)
    { return append(ilist); }
```

`insert` 和 `erase` 也有很多重载，但是其中有大量需要用到迭代器，这些就省了；还有一些重载关系可以用默认值替代，我们就合并一下。所以最后我们只需要写四个 `insert` 和一个 `erase` 就可以了。

```
public:
    string& insert(std::size_t pos, std::size_t count, char ch);
    string& insert(std::size_t pos, const char *s, std::size_t count);
    string& insert(std::size_t pos, const string &str,
                  std::size_t s_pos = {0}, std::size_t count = {npos});
    // 在pos位置起，从str的s_pos位置起选取count个字符插入本字符串中
    string& insert(std::size_t, std::initializer_list<char>);
    string& erase(std::size_t pos = {0}, std::size_t count = {npos});
```

其中第三个的插入方式比较复杂，注释中已经加以说明。至于 `clear`，只需要用 `erase(0, npos)`——含义就是，从第 0 字符到末尾的字符全删除，那么自然就可以起到 `clear` 的预期效果了。

```
public:
    void clear() { erase(); } // 传入erase函数的默认值，然后自然能起到clear作用
```

至于 `copy` 函数，它的作用是把 `pos` 开始的 `count` 个字符移入 `dest` 字符串内。换句话说，是在向外界字符串输出<sup>46</sup> 内容。至于返回值，就是实际复制进字符串中的字符个数。

```
public:
    std::size_t copy(char *dest, std::size_t count,
                    std::size_t pos = {0}) const; // 把pos起的count个字符移入dest
```

<sup>46</sup> 对象之间也有“输入”“输出”概念。对于一个对象来说，从外界（其它对象等）接收信息是输入，而向外界（其它对象等）提供信息就是输出。

swap 还有 swap 函数，无需多言。

```
public:
    void swap(string&); // 交换此对象与彼对象
```

至于 find 函数，它的重载也挺多的，我们挑三个就行，对别对应着根据 string 对象/char[N] 字符串/字符来查找。多余的功能支持就不做了。至于返回值，当然应当返回“位置”这个信息了，这样才能说明是在哪里找到的。如果找不到的话就要返回 npos，这也是我们此前讲过的。

```
public:
    std::size_t find(const string &str, std::size_t pos = {0})const;
    // 从 pos 位置起找 str
    std::size_t find(const char *s, std::size_t pos,
                    std::size_t count)const; // 以 s 的前 count 个字符为索引，从 pos 位置起找
    std::size_t find(char ch, std::size_t pos = {0})const;
    // 这三个都是常成员函数，因为不需要改变对象的内容
```

compare 函数以及六个比较运算符都可以用来与另一对象进行字典序比较。读者可能会觉得这个工作有点太难了，但是不要担心，我们可以像之前那样，用 C++ 算法库中的 std::lexicographical\_compare 函数模版来解决它。只要实现了小于号，compare 和其它五个运算符就都可以实现了。

```
public:
    int compare(const string&)const;
    bool operator<(const string&)const;
    bool operator>(const string&)const;
    bool operator<=(const string&)const;
    bool operator>=(const string&)const;
    bool operator==(const string&)const;
    bool operator!=(const string&)const;
```

最后我们重载一下输入输出。因为第一个参数不是 string 类型，所以我们就定义为非成员函数，使其作为 string 的友元存在就好。

```
public:
    friend std::ostream& operator<<(std::ostream&, const string&); // 输出
    friend std::istream& operator>>(std::istream&, string&); // 输入
```

以上就是我们的 string 类的全部内容。接下来就让我们逐个击破，把它们都实现出来吧。

## 实现

我们首先把构造函数写出来。在其它很多函数的实现过程中，我们都必须要定义一些临时变量。所以完善的构造函数是不可或缺的。

```
string::string(std::size_t count, char ch)
    : _cap{count} // _arr 就用默认初值 new char[_cap] 来初始化足矣
{
    for (; _size < count; _size++) // _size 已经通过默认初值定为 0，无需再赋值
        at(_size) = ch; // 这里用到了 at 成员函数，稍后我们会看到其定义
}
```

读者可能发现，在 string 当中我们没有使用 **char**[]/**char**\* 字符串那样在字符串之后加入一个 '\0' 作为结束符。这是因为我们已经有了一个专门负责记录字符串长度的成员变量 **\_size** 了，所以自然不需要使用结束符这种字符来标记字符串的结尾了。

```

string::string(const string &other, std::size_t pos, std::size_t count)
    : _cap {other._cap} //当pos和count都取默认值时它充当拷贝构造函数，故_cap要一致
{
    if (pos >= other._size) //为了防范pos越界的情况
        at(0) = '\0'; //直接把第一个字符变成空字符，什么也不做
    else {
        if (count == npos //npos的意思是直接截到字符串末尾
            || pos + count > other._size
        ) //对于pos+count越界或者count==npos的情况
            count = other._size - pos; //接下来count的值就合适了
        for (; _size < count; _size++)
            at(_size) = other.at(_size);
    }
}

```

这个版本的构造函数是从另一 `string` 对象拷贝内容，它也可以作为构贝构造函数存在。

读者需要注意可能存在的访问越界情况，比如说起始位置 `pos` 或者是终止点 `pos+count` 已经超过了 `other._size`，我们要采取相应的对策。还有一种可能就是 `count==npos`，这意味着用户想要使用整个 `string` 的全部内容。对于这种情况，我们也可以人为地把 `count` 改成 `other._size-pos`，从而满足 `pos+count==other._size`——这样我们就指定了 `pos` 起 `_size` 止<sup>47</sup>的内容。

```

string::string(const char *s) : _cap {std::strlen(s)} {
    for (; _size < _cap; _size++)
        at(_size) = s[_size];
}

string::string(const char* s, std::size_t count)
    : _cap {std::min(std::strlen(s), count)} //防止count大于strlen(s)
{
    for (; _size < _cap; _size++)
        at(_size) = s[_size];
}

```

这个版本的构造函数是从一个 `const char*` 字符串拷贝内容。在这里我们不能通过默认参数的方法来把二者合并为同一个函数，因为默认参数的确定是编译时行为，而编译器是无法预测 `std::strlen(s)` 将会是多少的。

```

string::string(std::initializer_list<char> ilist)
    : _cap {ilist.size()}
{
    for (char ch : ilist) //范围for循环
        at(_size++) = ch; //注意自增符号要用后缀
}

```

这个版本的构造函数支持列表初始化，其中细节都是老生常谈，不再赘述。最后写一个很简单的析构函数，这方面的工作就宣告完成了！

```

string::~string() {
    delete[] _arr.p(); //别忘了用delete[]而不是delete
}

```

<sup>47</sup>在 `std::string` 及 STL 中，所有“a 起 b 止”的说法一般都是指“包含 a 而不包含 b 的中间区段”。下面不再重复。

接下来顺着我们定义的顺序完成其它成员函数即可。首先是 `at`，有常成员函数和非常成员函数版本。注意，`at` 是有边界检测的。但我们还没有学异常处理，所以干脆把异常下标都变成 `_size-1`。这个方法并不万全，万一 `_size` 为 0 那也不行；但是这是现阶段我们唯一能做出的努力了。

```
char& string::at(std::size_t pos) {
    if (pos >= _size) //防越界，但是对于pos==0的情况无能为力
        pos = _size - 1;
    return _arr.p()[_size];
}

const char& string::at(std::size_t pos) const { //注意返回类型是const char&
    if (pos >= _size)
        pos = _size - 1;
    return _arr.p()[_size];
}
```

接下来是赋值函数 `assign`。

```
string& string::assign(std::size_t count, char ch) {
    if (count > _cap) //说明动态内存空间不足
        realloc(count, false); //重新分配，这时不需要进行内容复制，故传入false
    for (_size = 0; _size < count; _size++)
        at(_size) = ch;
    return *this;
}
```

在这里 `realloc` 成员函数的第二个参数就发挥作用了。想想，在这个例子中，我们的目的只是重新分配一个动态内存；至于原来的内容，我们根本不需要保留——反正下一步就是用 `char` 来填充。

```
string& string::assign(
    const string &str,
    std::size_t pos,
    std::size_t count
){ //这是一种效率比较低的做法，但是操作非常简单
    string tmp {str, pos, count}; //用一个tmp暂存需要的内容
    swap(tmp); //将此对象与tmp交换
    return *this;
}
```

对于这种参数中含有 `string` 对象的赋值函数来说，一定要警惕“自我赋值”及类似现象。不过，在本例中我们选择了一种代码实现上更简单的做法，就是先利用构造函数产生一个符合条件的临时对象，然后再用 `swap` 成员函数交换这两个对象的成员。<sup>48</sup>

```
string& string::assign(const char *s, std::size_t count) {
    std::size_t len {std::strlen(s)}; //防止多次调用std::strlen
    if (count > len) //如果count大于len，那就越界了
        count = len; //修正count的值
    if (count > _cap) //说明动态内存空间不足
        realloc(count, false); //这里也不需要保留原内容
    for (_size = 0; _size < count; _size++)
```

<sup>48</sup>虽然这么写很简单，但它只是个权宜之计。C++ 是一种很重视执行效率的语言，而我们的这个方法是很低效的（比如，有时会导致无谓的内存重分配），比 C++ 自带的 `std::string` 就要更逊一筹了。

```

    at(_size) = s[_size];
    return *this;
}

```

这里我们定义 `len` 的目的是防止多次调用 `std::strlen()` 函数，避免浪费时间。

```

string& string::assign(std::initializer_list<char> ilist) {
    string tmp{ilist};
    swap(tmp); // 故技重施
    return *this;
}

```

这里我们故技重施，读者会发现这么写确实很方便。本例仅用于教学，如果读者想要自己练习，也可以用其它写法实现这个函数。

赋值运算符及 `empty`, `size`, `length` 和 `capacity` 已经在类内定义了，这里就可以跳过。读者可以回看前面的代码来温习之。有些细节可能会疏于讲解，但以读者现有的知识，想必不难看懂。

接下来是 `reserve` 和 `resize` 函数，二者一个改变容量，一个改变长度。其中的 `reserve` 和 `realloc` 非常相似，只不过前者不能在改变 `_cap` 时导致内容受损，所以要保证新的内存大小能容纳现有内容；而后者不在乎这些，允许内容量的损失。

```

void string::reserve(std::size_t new_cap) {
    if (new_cap < _size) // 防止损失有效内容
        new_cap = _size;
    realloc(new_cap); // 直接使用已有的 realloc 函数来实现
}

```

而 `resize` 在改变长度时，要考虑内存空间不足的情况——如果需要的长度比真实容量还大，那么我们还需要扩容。所以我们可以把这步单独拎出来处理，先保证容量充足；然后再改变长度。这也是一种“分而治之”的思想。

```

void string::resize(std::size_t count, char ch) {
    if (_cap < count) // 先保证动态内存空间充足
        realloc(count); // 这里传入第二个默认参数 true，说明原内容需要保留
    if (count > _size)
        for (; _size < count; _size++)
            at(_size) = ch; // 从 _size 位置到 count-1 全部需要变成 ch，注意范围
    else
        _size = count; // 直接截尾即可，所以把 _size 变成 count
}

```

读者需要注意，在这里我们是需要

下面是一系列 `append` 成员函数。

```

string& string::append(std::size_t count, char ch) {
    if (_size + count > _cap)
        realloc(_size + count); // 重分配
    for (; _size < count; _size++)
        at(_size) = ch;
    return *this;
}

```

在写 `append` 成员函数时要注意动态内存空间是否足够。如果不夠，应当重新分配。其它的细节都不难，读者看了这么多，应该已经比较熟悉了。

```

string& string::append(
    const string &str,
    std::size_t pos,
    std::size_t count
) {
    if (pos >= str._size) // 如果起点都没有意义，那就不用拼接了
        return *this; // 直接返回
    if (pos + count > str._size || count == npos)
        count = str._size - pos;
    if (_size + count > _cap) // 如果内存不够
        realloc(_size+count); // 那就重新分配内存，保留内容
    for (std::size_t i = 0; i < count; i++)
        at(_size+i) = str.at(pos+i);
    return *this;
}

```

在这个函数中，我们需要防范两种情况：一是 `pos` 或 `count` 的值不合理，二是 `count==npos` 需要特殊处理。其实还有第三个可能的问题：自我赋值。但是对于本例来说，即便 `this==&str` 也没有什么风险。读者可以自行分析之。

```

string& string::append(const char *s, std::size_t count) {
    return append(s, 0, count);
    // 先用构造函数把s转换成string对象，然后调用另一个append重载
}

string& string::append(std::initializer_list<char> ilist) {
    return append(string(ilist));
    // 先用构造函数把s转换成string对象，然后调用另一个append重载，默认参数0, npos
}

```

这里我们又进行了代码重用，通过另外两个成员函数，直接实现本函数的功能。在调用 `append(const string&, std::size_t)` 的过程中，函数期待传入 `string` 对象但我们提供了 `const char*` 或 `std::initializer_list<char>` 对象，这时就会发生隐式类型转换。

有了 `append` 函数之后，`operator+=` 只是在此基础上做略微改动而已。我们已经在定义中完成它们，所以不用再多说了。

下面是一系列 `insert` 成员函数。`string` 的 `insert` 与链表不同，我们能做的只有在插入字符之前把其余字符逐个后移，从而为新内容腾出空间。

```

string& string::insert(std::size_t pos, std::size_t count, char ch) {
    if (_size + count > _cap) // 如果容量不够
        realloc(_size+count); // 扩容
    for (std::size_t i = _size - 1; i >= pos; i--)
        at(i+count) = at(i); // 从后向前，依次把字符后移count位，以留出空间
    for (std::size_t i = pos; i < pos+count; i++)
        at(i) = ch; // 把pos起的count个字符变为ch
    _size += count; // 最后别忘了更新_size的值！
    return *this;
}

```

这个插入的操作方法对于初学者来说可能有点难。首先，我们需要把待插入片段的字符逐个后移 `count` 位。这个移动必须从最后一个字符开始，否则就会出错。接下来我们再把 `pos` 起的 `count`

个字符变为 ch，这样就能起到插入数据的效果了。

接下来我们完成一下 (`std::size_t, const string&, std::size_t, std::size_t`) 版本的成员函数。在此之后，我们就可以用有关构造函数配合这个成员函数，降低代码工作量。

```
string& string::insert(
    std::size_t pos,
    const string &str,
    std::size_t s_pos,
    std::size_t count
) {
    if (s_pos >= str._size) // 如果起始点已经越界
        return *this; // 那就直接返回
    if (s_pos + count > str._size || count == npos)
        count = str._size - s_pos; // 调整count的值
    if (_size + count > _cap)
        realloc(_size+count); // 必要的扩容
    for (std::size_t i = _size - 1; i >= pos; i--)
        at(i+count) = at(i); // 数据后移
    for (std::size_t i = 0; i < count; i++)
        at(pos+i) = str.at(s_pos+i);
    // 从本对象的pos位置起，把str对象的s_pos位置起count个数据依次复制到此处
    _size += count; // 别忘了更新_size的值
    return *this;
}
```

这个函数的细节略显复杂，但是读者只要把握住每个参数的含义，并配合我提供的注释、插图，想必还是可以看懂的。看懂了这个之后，我们就没有必要再在剩的两个函数上继续磨蹭，直接重用之前的代码就好。

```
string& string::insert(std::size_t pos, const char *s, std::size_t count) {
    return insert(pos, s, 0, count);
}
string& string::insert(std::size_t pos, std::initializer_list<char> ilist) {
    return insert(pos, ilist, 0, ilist.size());
}
```

而 `erase` 函数的作用正相反。在用它移除数据时，我们只需要关注内容本身和 `_size` 成员的更新就足够了，至于 `_cap`，那不是它应当关心的东西。所以这个函数就很简单了。

```
string& string::erase(std::size_t pos, std::size_t count) {
    if (pos >= _size)
        return *this;
    if (pos + count > _size || count == npos)
        count = _size - pos;
    for (std::size_t i = pos; i + count < _size; i++) // 注意范围
        at(i) = at(i+count);
    _size -= count; // 记得更新_size
    return *this;
}
```

`copy` 的功能也不复杂，很轻松就写出来了。

```
std::size_t string::copy(
    char *dest,
    std::size_t count,
    std::size_t pos
) const {
    if (pos >= _size)
        return 0;
    if (pos + count > _size || count == npos)
        count = _size - pos;
    for (std::size_t i = 0; i < count; i++) //把pos起的count个数据复制到字符串中
        dest[i] = at(pos+i);
    return count;
}
```

接下来是 swap 成员函数。在交换二者的 \_arr 成员时，我们直接用浅拷贝就行——也就是，把指针的指向交换，而不用麻烦我们再复制一遍内容。

```
void string::swap(string &str) {
    std::swap(_cap, str._cap);
    std::swap(_size, str._size);
    std::swap(_arr.p(), str._arr.p());
}
```

接下来是一系列 find 函数。我们只写两个版本，一个是单字符查找，另一个是 string 对象查找。至于 `char*` 的查找，我们只需要套一层隐式类型转换的外壳就行了。

```
std::size_t string::find(char ch, std::size_t pos) const {
    for (std::size_t i = pos; i < _size; i++) //从pos位置起开始寻找
        if(at(i) == ch) //找到了
            return i; //把i作为返回值
    //一直到_size处都没找到
    return npos; //npos可以表示没找到
}
```

其中的 `npos` 可以表示未找到，那么当我们实际使用这个函数的时候，只需检测返回值是否为 `npos` 就可以了。

```
std::size_t string::find(string &str, std::size_t pos) const {
    for (std::size_t i = pos; i + str._size < _size; i++){
        bool equal {true}; //先假设片段 [i,i+str._size) 与片段 str 相等
        for (std::size_t j = 0; j < _size; j++)
            if (at(i+j) != str.at(j)) { //一旦发现一个不相等的字符
                equal = false; //标记为不相等
                break; //退出循环
            } //可以想见，如果这个循环下来没有任何字符不相等，那么它就是我们要找的
        if(equal) //equal 为真
            return i;
    }
    //一直到找了一大圈都无果而终
    return npos; //没找到
}
```

这个函数涉及到的查找过程较为复杂<sup>49</sup>，如果读者尚不能理解，那也无妨，我们可以先把代码抄过来跑了再说。

```
std::size_t string::find(
    const char *s,
    std::size_t pos,
    std::size_t count
) const {
    return find(string(s, count), pos);
    // 这里要调用的是 string(const char*, std::size_t)，不是 string(const char*)
}
```

接下来是供以字典序比较的 `compare` 函数和比较运算符。我们只需要实现其中的小于号，剩下的就都不难了。

```
bool string::operator<(const string &str) const {
    return std::lexicographical_compare(
        &at(0), &at(_size), &str.at(0), &str.at(str._size)
    ); // 利用 std::lexicographical_compare 算法，具体的细节就不讲了
}

bool string::operator>(const string &str) const {
    return str < *this; // 利用已经重载好的小于号
}

bool string::operator<=(const string &str) const {
    return !(str < *this); // 注意运算符的优先级
}

bool string::operator>=(const string &str) const {
    return !(*this < str);
}

bool string::operator!=(const string &str) const {
    return (*this < str) || (str < *this);
}

bool string::operator==(const string &str) const {
    return !operator!=(str); // 进一步利用已经重载好的 operator!=
}

int string::compare(const string &str) const {
    return -int(*this < str) + int(*this > str); // bool 到 int 类型转换后再计算
}
```

最后是两个重载运算符，供以输入和输出。输出比较简单，从 0 到 `_size` 逐个字符输出就好了。

```
std::ostream& operator<<(std::ostream &out, const string &str) { // string 的友元
    for (std::size_t i = 0; i < str._size; i++)
        out << str.at(i);
    return out; // 返回 out 以便连续输出
}
```

输入就比较棘手了，要考虑的细节太多，我们还是放到第十三章再讲吧。读者可以先借鉴一下这段代码：

---

<sup>49</sup>顺便一说，这是一种比较低效率的算法；但是更高效的 KMP 算法（Knuth–Morris–Pratt algorithm）比这个还要复杂一些。

```

std::istream& operator>>(std::istream &in, string &str) { //string的友元
    char ch;
    str.clear(); //先清理str中的已有内容
    while (in.get(ch) && std::isspace(ch)) {
        //空，什么也不做
    } //这个循环的目的在于清理空白字符
    if (in)
        do { //注意，do-while结构会至少执行一次
            str.append(1, ch); //把输入的单个字符加到str的末尾
        } while (in.get(ch) && !std::isspace(ch));
        //在while中读取下一个字符，并判断它是不是空白字符
    if (in && !std::isspace(ch)) //如果这个字符不是空白字符，且属于下一个输入
        in.unget(); //把这个字符还回去
    return in; //返回in以便连续输入
}

```

## 完整代码

代码 8.3: Header.h

```

#pragma once
#include <iostream>
#include <initializer_list>
#include <cstring>
class string { //class的默认成员访问权限是private，所以private可省略
    static constexpr std::size_t npos = -1; //串尾指示器
    std::size_t _cap {0}; //容量，默认初值为0
    std::size_t _size {0}; //长度，默认初值为0
    class Arr {
        char *_p;
    public:
        Arr(char *ptr = {nullptr}) : _p {ptr} {} //Arr的构造函数
        char*& p() { return _p; } //当arr不是常量时调用
        const char* p() const { return _p; } //当arr是常量时调用
    } _arr {new char[_cap]}; //象征指针的成员，默认用new char[_cap]初始化
    void realloc(std::size_t cap, bool copy = true) { //重分配内存函数
        char *tmp = _arr.p(); //临时指针，用以暂存当前内存地址
        _arr.p() = new char[cap]; //分配新的内存地址
        if (copy) //说明这个函数需要进行内容复制
            for (std::size_t i = 0; i < std::min(cap, _size); i++)
                _arr.p()[i] = tmp[i]; //转移内容
        delete[] tmp; //回收原来的内存地址
        _cap = cap; //更新_cap的值
        if (_size > _cap)
            _size = _cap; //保证_size不能大于_cap
    }
public:

```

```

explicit string() {} //默认构造函数， 定义成explicit防止被误用
string(std::size_t count, char ch); //重复count个ch
string(const string &other, std::size_t pos = {0},
       std::size_t count = {npos}); //在other的pos位置起接收count个字符
string(const char *s); //用s字符串初始化
string(const char *s, std::size_t count); //用s字符串的前count个字符初始化
string(std::initializer_list<char>); //需要头文件initializer_list
~string(); //析构函数
char& at(std::size_t pos); //返回pos位置的字符
const char& at(std::size_t pos)const; //常成员函数版本
char& operator[](std::size_t pos) { return _arr.p()[pos]; } //下标运算符
const char& operator[](std::size_t pos)const { return _arr.p()[pos]; }
string& assign(std::size_t count, char ch); //赋值重复count个ch
string& assign(const string &str, std::size_t pos = {0},
               std::size_t count = {npos}); //在str的pos位置起接收count个字符
string& assign(const char *s, std::size_t count); //接收s的前count个字符
string& assign(std::initializer_list<char>); //用列表赋值
string& operator=(const string &str) { return assign(str); }
string& operator=(const char *s) { return assign(s, std::strlen(s)); }
//其中std::strlen需要头文件cstring
string& operator=(char ch) { return assign(1, ch); }
string& operator=(std::initializer_list<char> ilist)
{ return assign(ilist); }
bool empty()const { return _size; } //用到了隐式类型转换
std::size_t size()const { return _size; }
std::size_t length()const { return _size; }
std::size_t capacity()const { return _cap; }
void reserve(std::size_t new_cap); //改变容量到new_cap， 除非影响内容
void resize(std::size_t count, char ch = {'\0'}); //改变内容长度
string& append(std::size_t count, char ch);
string& append(const string &str, std::size_t pos = {0},
               std::size_t count = {npos});
string& append(const char *s, std::size_t count);
string& append(std::initializer_list<char>);
string& operator+=(char ch) { return append(1, ch); }
string& operator+=(const string &str) { return append(str); }
string& operator+=(const char *s) { return append(s, std::strlen(s)); }
string& operator+=(std::initializer_list<char> ilist)
{ return append(ilist); }
string& insert(std::size_t pos, std::size_t count, char ch);
string& insert(std::size_t pos, const string &str,
               std::size_t s_pos = {0}, std::size_t count = {npos});
//在pos位置起， 从str的s_pos位置起选取count个字符插入本字符串中
string& insert(std::size_t pos, const char *s, std::size_t count);
string& insert(std::size_t, std::initializer_list<char>);
string& erase(std::size_t pos = {0}, std::size_t count = {npos});

```

```

void clear() { erase(); } // 传入erase函数的默认值，然后自然能起到clear作用
std::size_t copy(char *dest, std::size_t count,
    std::size_t pos = {0})const; // 把pos起的count个字符移入dest，返回长度
void swap(string&); // 交换此对象与彼对象
std::size_t find(char ch, std::size_t pos = {0})const;
std::size_t find(const string &str, std::size_t pos = {0})const;
// 从pos位置起找str
std::size_t find(const char *s, std::size_t pos,
    std::size_t count)const; // 以s的前count个字符为索引，从pos位置起找
// 这三个都是常成员函数，因为不需要改变对象的内容
int compare(const string&)const;
bool operator<(const string&)const;
bool operator>(const string&)const;
bool operator<=(const string&)const;
bool operator>=(const string&)const;
bool operator==(const string&)const;
bool operator!=(const string&)const;
friend std::ostream& operator<<(std::ostream&, const string&); // 输出
friend std::istream& operator>>(std::istream&, string&); // 输入
};

```

代码 8.4: Definition.cpp

```

#include "Header.h"
#include <algorithm>
string::string(std::size_t count, char ch)
    : _cap{count} // _arr 就用默认初值 new char[_cap] 来初始化足矣
{
    for (; _size < count; _size++) // _size 已经通过默认初值定为 0，无需再赋值
        at(_size) = ch; // 这里用到了 at 成员函数，稍后我们会看到其定义
}
string::string(const string &other, std::size_t pos, std::size_t count)
    : _cap{other._cap} // 当 pos 和 count 都取默认值时它充当拷贝构造函数，故 _cap 要一致
{
    if (pos >= other._size) // 为了防范 pos 越界的情况
        at(0) = '\0'; // 直接把第一个字符变成空字符，什么也不做
    else {
        if (count == npos // npos 的意思是直接截到字符串末尾
            || pos + count > other._size
        ) // 对于 pos+count 越界或者 count==npos 的情况
            count = other._size - pos; // 接下来 count 的值就合适了
        for (; _size < count; _size++)
            at(_size) = other.at(_size);
    }
}
string::string(const char *s) : _cap{std::strlen(s)} {
    for (; _size < _cap; _size++)
        at(_size) = s[_size];
}

```

```
}

string::string(const char* s, std::size_t count)
    : _cap {std::min(std::strlen(s),count)} //防止count大于strlen(s)
{
    for (; _size < _cap; _size++)
        at(_size) = s[_size];
}

string::string(std::initializer_list<char> ilist)
    : _cap {ilist.size()}

{
    for (char ch : ilist) //范围for循环
        at(_size++) = ch; //注意自增符号要用后缀
}

string::~string() {
    delete[] _arr.p(); //别忘了用delete[]而不是delete
}

char& string::at(std::size_t pos) {
    if (pos >= _size)
        pos = _size - 1;
    return _arr.p()[_size];
}

const char& string::at(std::size_t pos)const {
    if (pos >= _size)
        pos = _size - 1;
    return _arr.p()[_size];
}

string& string::assign(std::size_t count, char ch) {
    if (count > _cap) //说明动态内存空间不足
        realloc(count, false); //重新分配，这时不需要进行内容复制，故传入false
    for (_size = 0; _size < count; _size++)
        at(_size) = ch;
    return *this;
}

string& string::assign(
    const string &str,
    std::size_t pos,
    std::size_t count
){ //这是一种效率比较低的做法，但是操作非常简单
    string tmp {str, pos, count}; //用一个tmp暂存需要的内容
    swap(tmp); //将此对象与tmp交换
    return *this;
}

string& string::assign(const char *s, std::size_t count) {
    std::size_t len {std::strlen(s)}; //防止多次调用std::strlen
    if (count > len) //如果count大于len，那就越界了
        count = len; //修正count的值
```

```
if (count > _cap) //说明动态内存空间不足
    realloc(count, false); //这里也不需要保留原内容
for (_size = 0; _size < count; _size++)
    at(_size) = s[_size];
return *this;
}
string& string::assign(std::initializer_list<char> ilist) {
    string tmp{ilist};
    swap(tmp); //故技重施
    return *this;
}
void string::reserve(std::size_t new_cap) {
    if (new_cap < _size) //防止损失有效内容
        new_cap = _size;
    realloc(new_cap); //直接使用已有的realloc函数来实现
}
void string::resize(std::size_t count, char ch) {
    if (_cap < count) //先保证动态内存空间充足
        realloc(count); //这里传入第二个默认参数true，说明原内容需要保留
    if (count > _size)
        for (; _size < count; _size++)
            at(_size) = ch; //从_size位置到count-1全部需要变成ch，注意范围
    else
        _size = count; //直接截尾即可，所以把_size变成count
}
string& string::append(std::size_t count, char ch) {
    if (_size + count > _cap)
        realloc(_size+count);
    for (; _size < count; _size++)
        at(_size) = ch;
    return *this;
}
string& string::append(
    const string &str,
    std::size_t pos,
    std::size_t count
) {
    if (pos >= str._size) //如果起点都没有意义，那就不用拼接了
        return *this; //直接返回
    if (pos + count > str._size || count == npos)
        count = str._size - pos;
    if (_size + count > _cap) //如果内存不够
        realloc(_size+count); //那就重新分配内存，保留内容
    for (std::size_t i = 0; i < count; i++)
        at(_size+i) = str.at(pos+i);
    return *this;
}
```

```

}

string& string::append(const char *s, std::size_t count) {
    return append(s, 0, count);
    //先用构造函数把s转换成string对象，然后调用另一个append重载
}

string& string::append(std::initializer_list<char> ilist) {
    return append(string{ilist});
    //先用构造函数把s转换成string对象，然后调用另一个append重载，默认参数0, npos
}

string& string::insert(std::size_t pos, std::size_t count, char ch) {
    if (_size + count > _cap) //如果容量不够
        realloc(_size+count); //扩容
    for (std::size_t i = _size - 1; i >= pos; i--)
        at(i+count) = at(i); //从后向前，依次把字符后移count位，以留出空间
    for (std::size_t i = pos; i < pos+count; i++)
        at(i) = ch; //把pos起的count个字符变为ch
    _size += count; //最后别忘了更新_size的值！
    return *this;
}

string& string::insert(
    std::size_t pos,
    const string &str,
    std::size_t s_pos,
    std::size_t count
) {
    if (s_pos >= str._size) //如果起始点已经越界
        return *this; //那就直接返回
    if (s_pos + count > str._size || count == npos)
        count = str._size - s_pos; //调整count的值
    if (_size + count > _cap)
        realloc(_size+count); //必要的扩容
    for (std::size_t i = _size - 1; i >= pos; i--)
        at(i+count) = at(i); //数据后移
    for (std::size_t i = 0; i < count; i++)
        at(pos+i) = str.at(s_pos+i);
    //从本对象的pos位置起，把str对象的s_pos位置起count个数据依次复制到此处
    _size += count; //别忘了更新_size的值
    return *this;
}

string& string::insert(std::size_t pos, const char *s, std::size_t count) {
    return insert(pos, s, 0, count);
}

string& string::insert(std::size_t pos, std::initializer_list<char> ilist) {
    return insert(pos, ilist, 0, ilist.size());
}

string& string::erase(std::size_t pos, std::size_t count) {
}

```

```
if (pos >= _size)
    return *this;
if (pos + count > _size || count == npos)
    count = _size - pos;
for (std::size_t i = pos; i + count < _size; i++) //注意范围
    at(i) = at(i+count);
_size -= count; //记得更新_size
return *this;
}

std::size_t string::copy(
    char *dest,
    std::size_t count,
    std::size_t pos
) const {
    if (pos >= _size)
        return 0;
    if (pos + count > _size || count == npos)
        count = _size - pos;
    for (std::size_t i = 0; i < count; i++) //把pos起的count个数据复制到字符串中
        dest[i] = at(pos+i);
    return count;
}

void string::swap(string &str) {
    std::swap(_cap, str._cap);
    std::swap(_size, str._size);
    std::swap(_arr.p(), str._arr.p());
}

std::size_t string::find(char ch, std::size_t pos) const {
    for (std::size_t i = pos; i < _size; i++) //从pos位置起开始寻找
        if (at(i) == ch) //找到了
            return i; //把i作为返回值
    //一直到_size处都没找到
    return npos; //npos可以表示没找到
}

std::size_t string::find(const string &str, std::size_t pos) const {
    for (std::size_t i = pos; i + str._size < _size; i++){
        bool equal {true}; //先假设片段[i,i+str._size)与片段str相等
        for (std::size_t j = 0; j < _size; j++)
            if (at(i+j) != str.at(j)) { //一旦发现一个不相等的字符
                equal = false; //标记为不相等
                break; //退出循环
            } //可以想见，如果这个循环下来没有任何字符不相等，那么它就是我们要找的
        if(equal) //equal为真
            return i;
    }
    //一直到找了一大圈都无果而终
}
```

```

    return npos; //没找到
}

std::size_t string::find(
    const char *s,
    std::size_t pos,
    std::size_t count
) const {
    return find(string(s, count), pos);
    //这里要调用的是string(const char*,std::size_t)，不是string(const char*)
}

bool string::operator<(const string &str) const {
    return std::lexicographical_compare(
        &at(0), &at(_size), &str.at(0), &str.at(str._size)
    ); //利用std::lexicographical_compare算法，具体的细节就不讲了
}

bool string::operator>(const string &str) const {
    return str < *this; //利用已经重载好的小于号
}

bool string::operator<=(const string &str) const {
    return !(str < *this); //注意运算符的优先级
}

bool string::operator>=(const string &str) const {
    return !(*this < str);
}

bool string::operator!=(const string &str) const {
    return (*this < str) || (str < *this);
}

bool string::operator==(const string &str) const {
    return !operator!=(str); //进一步利用已经重载好的operator!=
}

int string::compare(const string &str) const {
    return -int(*this<str) + int(*this>str); //bool到int类型转换后再计算
}

std::ostream& operator<<(std::ostream &out, const string &str) { //string的友元
    for (std::size_t i = 0; i < str._size; i++)
        out << str.at(i);
    return out; //返回out以便连续输出
}

std::istream& operator>>(std::istream &in, string &str) { //string的友元
    char ch;
    str.clear(); //先清理str中的已有内容
    while (in.get(ch) && std::isspace(ch)) {
        //空，什么也不做
    } //这个循环的目的在于清理空白字符
    if (in)
        do { //注意，do-while结构会至少执行一次
}

```

```
    str.append(1, ch); // 把输入的单个字符加到str的末尾
} while (in.get(ch) && !std::isspace(ch));
// 在while中读取下一个字符，并判断它是不是空白字符
if (in && !std::isspace(ch)) // 如果这个字符不是空白字符，且属于下一个输入
    in.unget(); // 把这个字符还回去
return in; // 返回in以便连续输入
}
```



# 第九章 类的继承

在上一章中，我们讲过了类与函数的进阶知识，并完成了一个简单的 `valarri` 和一个稍复杂的 `string` 类。而在本章中，读者将看到有关类的另一个重要知识：继承。

此前讲到的那些类，它们之间没有什么关联。`std::vector` 是 `std::vector`, `std::valarray` 是 `std::valarray`, 是两个不同的事物。它们之间唯一的共同点可能就在于“它们表示的都是数组”，以及有一些名字比较相似的成员函数，仅此而已。

而在本章，我们将见识到继承操作下，类的两种关系：一种是公有继承代表的“种与属”关系，另一种是私有继承代表的“整体与部分”关系。本章也是后续介绍多态性概念的基础，有着承上启下的作用。

## 9.1 基本概念

以下三种关系是 C++ 中常见的，读者务必搞清楚它们之间的区别，切勿混淆。

### 整体与部分（Has-a relationship）

还记得我们在第六章讲述结构体时如何说吗？

```
int cuboid[3]; // 定义一个3长度数组，用来表示一个长方体
```

通过这种方式，我们定义了一个数组用以表示长方体，但它终究是三个孤立的值（只是存放在一起方便访问），而不是一个“整体”——比如说，函数的返回值只能是单个数据，而不能是数组。这就好比，我们已经有了发动机、轮胎、蓄电池、保险杠等等一应俱全的零件，但就是没有一辆完整的汽车。

所以我们应该把这样一盘散沙的零件封装成一个完整的对象，这样也就有了“整体与部分”的概念。

一辆汽车，它的发动机就是它的一部分；一个人，他的大脑和心脏就是他的一部分；一个动物园，它的各个展区——狮子区、老虎区等，都是它的一部分；一个国家，它的各个城市，都是它的一部分。

以上所述，都是“整体与部分”的关系，用英文可以阐述为“**Has-a relationship**”，在 C++ 中可以诠释为“对象与成员”。如果我们定义一个 `valarri` 类的 `a`，那么 `a` 就是对象，`a._size`, `a._cap` 和 `a._arr` 都是成员。这些成员如果单独存在，它表达的含义就不明显了；但是倘若它们存放在一起，那么我们就可以根据它们各自的值，以及它们之间的相互关系，从而表示更丰富、更明确的含义。

我们在第八章曾讲过，一个类的成员同时也是另一个类的对象——这说明整体与部分的关系是有传递性的。作为汽车一部分的发动机，它也是由气缸、活塞、燃油泵等部分构成的。就这样，我们可以把一个对象拆解成若干成员，又把这些作为成员的对象拆解成更小单位的成员，一直拆解到 `int`, `double` 等基本类型及它们的指针、数组类型。

## 模版与实例

我们曾在第四章简单地讲解过泛型编程的概念。我们说，泛型编程并不直接定义具体的函数，而是定义了一个模版（Template）。

```
template<typename T>
T user::max(const T& lhs, const T& rhs) {
    return lhs > rhs ? lhs : rhs;
}
```

如果我们在代码中需要调用它来进行两个 `double` 数据求最大值，那么编译器就会照此模版生成一个 `user::max(const double&, const double)` 函数，我们把它称为这个模版的实例（Instance）。一个模版可能有千奇百怪的实例，用以处理各种不同的数据，但它们总有很多的相似之处。

现在我们把“模版与实例”这个概念扩展到更广义的层面上去。模版就是一个抽象的概念，它不对应着什么实体；而实例正是在这个模版概念之下产生的实体。“人类”就是这样的一个模版，它只是一个抽象概念，不对应任何实体；而“张三”“李四”这样的实体就是这个模版的实例，他们可能有不同的属性，比如身高、体重，但总有一些独特的、能与其它物种分开的属性，比如“读写能力”之类的。

这种“模版与实例”的关系，用英文可以阐述为“**Instance-of relationship**”，在 C++ 中可以诠释为“类与对象”。对于我们定义的 `valarri a` 来说，`valarri` 就是类，而 `a` 就是对象。

初学者很容易搞混“**Instance-of relationship**”与“**Has-a relationship**”，但只要记住一点：“类”是一个抽象的概念，不对应实体；而“对象”和“成员”都是具体的事物，它们对应着实体<sup>1</sup>。也正因为“类”与“对象”不是同一层次的概念，所以“模版与实例”的关系是没有传递性的。

## 属与种

在讲述这个关系前我想请读者思考一个问题：我家的哈士奇和李四家的萨摩耶是同类吗？

是吗？好像不是。不是吗？好像也是。

之所以存在这样的困惑，是因为我们没说清“同类”是什么概念。对于我家的哈士奇来说，它属于“哈士奇类（Husky）”；而对于李四家的萨摩耶来说，它属于“萨摩耶类（Samoyed）”——那么它们当然不是同类了。

但是我们还有另一种观察角度：无论我家的哈士奇还是李四家的萨摩耶，它们都属于“家犬亚种（Canis lupus familiaris）”——那么它们当然是同类了。

这时你就会发现一个问题：好像我家的哈士奇同时属于“家犬”和“哈士奇”哎。其实不只如此，它还属于“狼种（Canis lupus）”“犬属（Canis）”“犬科（Canidae）”“食肉目（Carnivora）”“哺乳纲（Mammalia）”“脊索动物门（Chordata）”“动物界（Animalia）”。看上去是不是很复杂？很吓人？

问题来了。在我们既有的认知中，一个对象只能是单个类的对象，那么现在它同时是这么多类的对象，这岂不是乱套了？其实并不会，因为这些类之间存在着一种“属与种”的关系。用英文可以阐述为“**Is-a relationship**”，在 C++ 中可以诠释为“基类与派生类”。<sup>2</sup>

“属与种”的关系描述的是两个类的关系，它们有点像数学上的集合间包含关系——如果一个对象是类 A 的对象，那么它一定也是类 B 的对象。在这个关系当中，A 就是基类，而 B 是派生类。对于派生类的对象来说，它既有基类的共性——比如我家哈士奇能“汪汪”叫，王五家的泰迪也能，又有派生类的独特性——比如我家哈士奇是拆家小能手，但王五家的泰迪就不能胜任了。

<sup>1</sup>这不是金科玉律！但是对于日常生活中九成以上的情况来说都足够了。

<sup>2</sup>值得一提的是，一般我们只用公开继承/受保护继承来表示“**Is-a relationship**”；至于私有继承，它更适合用来表示“**Has-a relationship**”。

## 何为继承？

继承（Inheritance）是一种代码重用的好方法。如果我们已经写了一个类 Dog，而我们想要再写一个 Husky 类，那么我们没必要把 Dog 已有的那部分功能再写一遍，直接让 Husky 类继承 Dog 类就足够了——然后我们只需要集中精力去写诸如“拆家”之类的独特功能即可。

```
struct Dog { // 狗类
    unsigned age; // 所有的狗都有年龄属性
    double weight; // 所有的狗都有体重属性
    //...
};

struct Husky : Dog { // 哈士奇类，公开继承自狗类
    void destroy() { /*...*/ } // 哈士奇独特的拆家本领
};

int main() {
    Husky mine {5, 27}; // 哈士奇类的对象拥有狗类的成员
    std::cout << mine.age << 'u'; // 输出mine的年龄
    std::cout << mine.weight; // 输出mine的体重
    mine.destroy(); // 拆家，开始执行
}
```

在这个继承操作当中，Husky 类继承自 Dog 类。其中的 Husky 称为派生类（Derived class），而 Dog 称为基类（Base class）。派生类的对象拥有基类的成员，所以 mine 可以使用 age, weight 等基类成员，就好像它们定义在了 Husky 类当中一样，如图所示。

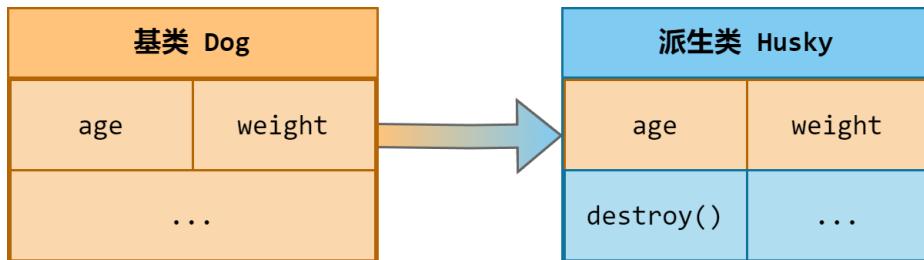


图 9.1: Dog 类的成员与 Husky 类的成员

## 9.2 公开继承与受保护成员

### 基本语法

继承的基本语法就是在定义<sup>3</sup>一个类时，用冒号加“继承方式”和“基类名”的形式来表明它按何种方式继承自哪个类。

```
class Base {
    //...
};

//Base类定义已毕

class Derived : public Base { //按public方式继承自Base类
    //...
};

//Derived公开继承自Base
```

<sup>3</sup>虽然在声明类时偶尔也可以这样做，但是我不推荐如此，因为在进行继承时编译器必须已经知道对应基类的定义。

其中的继承方式分为 **public**, **protected** 和 **private** 三种，在本节中我们先讲 **public**，即公有继承。

## 继承方式与访问权限

我们已经很熟悉“访问权限”这个概念了。访问权限关乎一个成员是否对外界可见：

- **public** 成员对外完全可见。
- **private** 成员对外完全不可见。
- **protected** 成员比较特殊，我们稍后讲到。这些成员对外界不可见，但对该类的派生类可见。

我们说，一个派生类对象具有基类的成员，比如 Husky 就有 age 和 weight。那么在派生类中，这些基类的成员又是何种访问权限呢？这是由“基类成员访问权限”和“继承方式”二者共同决定的，见表 9.1。

继承方式	基类成员权限	<b>public</b>	<b>protected</b>	<b>private</b>
	派生类成员权限			
<b>public</b>		<b>public</b>	<b>protected</b>	不可见
<b>protected</b>		<b>protected</b>	<b>protected</b>	不可见
<b>private</b>		<b>private</b>	<b>private</b>	不可见

表 9.1：派生类的成员访问权限取决于基类成员访问权限和继承方式

其中的 **protected** 成员（受保护成员）访问权限对于读者来说是个全新的概念，下面我就来解释一下它的作用和优缺点。

## **protected** 成员

我们谈过，**private** 成员的优点在于它是封闭的，只对类内可见。但对于继承来说，它也可能是个麻烦。

举个例子，Husky 类中的 destroy 成员函数需要用到 weight 这个成员变量。如果我们像上一节中那样在 Dog 类中把 weight 定义为 **public**<sup>4</sup>，那么无论 Husky 类还是其它无关的外界类/函数都有了修改 weight 的权限，这是很危险的；然而，如果我们把 weight 定义成 **private**，那么无论 Husky 还是其它外界类/函数都不能访问 weight<sup>5</sup>。

而 **protected** 则能很好地解决这个问题。基类的 **protected** 成员对于外界来说是不可见的，但它对于这个类的派生类来说则是可见的，如图 9.2 所示。

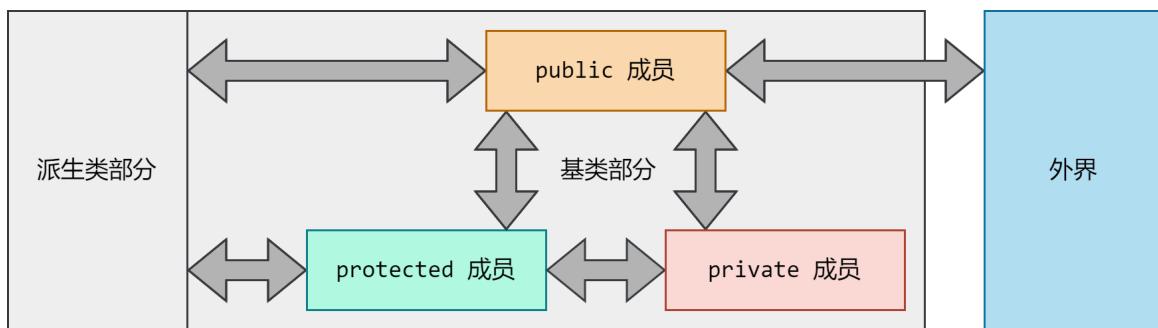
有了 **protected** 之后，我们可以这样写：

```
class Dog {
protected: // 受保护成员，对外不可见，但对Dog的派生类可见
    unsigned _age; // 所有狗都有年龄属性
    double _weight; // 所有狗都有体重属性
};

class Husky : public Dog { // 公有方式继承自Dog
```

<sup>4</sup> 我们此前提过，**struct** 成员的默认访问权限均为 **public**；**struct** 类的默认继承方式也是 **public**。所以 Dog 类中的成员都是公有成员，而 Husky 类以公开方式继承 Dog 类。

<sup>5</sup> 一种观点认为，**private** 成员是“不可继承”的，换句话说，基类的私有成员并不是派生类的成员。这种看法是合理的，因为一个类的成员总该对这个类可见；如果对这个类都不可见，那么它也算不上是这个类的成员。不过笔者在这里顾及理解上方便，还是选择这样讲。

图 9.2: **protected** 成员对外的可见性

```
// 这个类拥有成员 age 和 weight，成员权限为 protected
public: // 公有成员，对外界可见
    void destroy() { /*...*/ }; // 哈士奇独特的拆家本领
};
```

这样一来，`_age` 和 `_weight` 就对 Husky 类可见，使 Husky 类的函数可以方便地访问这些成员；同时它们又对无关的其它类不可见，保证了成员不受篡改的安全性。

## 构造与初始化

派生类拥有它基类的成员，这个关系很像是，一个派生类对象当中内嵌了一个基类对象。所以当

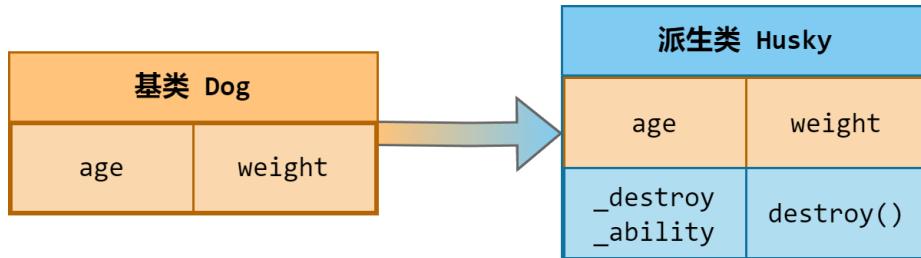


图 9.3: 一个派生类对象当中内嵌了一个基类对象

我们需要对基类对象进行初始化时，也不是按照“成员”进行初始化，而是把基类的对象当作一个整体，调用基类的构造函数进行初始化。

```
class Dog {
protected: // 受保护成员，对外不可见，但对 Dog 的派生类可见
    Dog(unsigned age, double weight) : _age{age}, _weight{weight} {}
    // 如果把 Dog 的构造函数定义成 protected 权限，那么只有其派生类才能正常调用它
    unsigned _age; // 所有狗都有年龄属性
    double _weight; // 所有狗都有体重属性
};

class Husky : public Dog { // 哈士奇类，公有方式继承自 Dog
    // 这个类拥有成员 age 和 weight，成员权限为 protected
private:
    int _destroy_ability; // 拆家能力
public:
    Husky(unsigned age, double weight, int ability)
```

```

        : Dog {age, weight}, _destroy_ability {ability}
    {} // 初始化内嵌 Dog 对象的成员，以及 _destroy_ability 成员
    void destroy() { /*...*/ }; // 哈士奇独特的拆家本领
};

class Retriever : public Dog { // 金毛寻回犬类，公有方式继承自 Dog
    // 这个类拥有成员 age 和 weight，成员权限为 protected
private:
    int _guide_ability; // 导盲能力
public:
    Retriever(unsigned age, double weight, int ability)
        : Dog {age, weight}, _guide_ability {ability}
    {} // 初始化内嵌 Dog 对象的成员，以及 _guide_ability 成员
    void guide() { /*...*/ } // 金毛独特的导盲本领
};

```

在这里，我们把 Dog 类的构造函数定义成受保护成员。按前文所述，这些成员对派生类以外的外界都是不可见的，所以我们不能在外部使用这个函数来进行构造<sup>6</sup>。换言之，我们这样就保证了：在外界中不能直接定义 Dog 类对象，只能定义它的派生类的对象<sup>7</sup>。

两个派生类 Husky 和 Retriever 对象的构造函数都是公有的，我们就可以在类外调用了。它们都接收三个参数，分别是 age, weight, ability。所以我们可以这样定义对象：

```

int main() {
    Husky mine {5, 27, 10000000};
    Retriever zhang3 {4, 25.67, 99};
}

```

在创建 mine 对象时，程序将调用 Husky::Husky(**unsigned, double, int**) 函数，那么这个函数做了什么呢？

```

public:
    Husky(unsigned age, double weight, int ability)
        : Dog {age, weight}, _destroy_ability {ability}
    {}

```

我们看到，这个函数在初始化时，其 Dog 成员部分是通过调用 Dog 类的构造函数来集中解决的；而独属于 Husky 的那部分，才是单独处理的。这一点对三种继承方式来说都是通用的——来自基类的成员，需要通过基类的构造函数来进行初始化。即便我们在代码中没有这么写，基类的构造函数也是会被调用的。我们可以用这段代码来验证之：

```

struct Base { // struct 成员默认为 public 成员，这里图方便就用 struct 了
    Base() { // Base 的默认构造函数
        std::cout << "Base::Base() is called." << std::endl;
    }
};

struct Derived : Base { // struct 类的默认继承方式也是公开继承
    Derived() { // 在 Derived 构造函数当中并没有写明要调用 Base 的构造函数
        std::cout << "Derived::Derived() is called." << std::endl;
    }
}

```

<sup>6</sup>其实这个类还有一个作为公有成员的默认拷贝构造函数和默认移动构造函数，所以确实还存在这样的方法使我们能够在外部定义对象。

<sup>7</sup>不过，从另一个意义上讲，派生类的对象也是基类的对象。

```

};

int main() {
    Derived de; // 定义Derived类的对象，预期会调用Derived类的构造函数
}

```

这段代码的运行结果如下：

---

```

Base::Base() is called.
Derived::Derived() is called.

```

---

这段代码能体现出两条信息：其一，在调用派生类的构造函数时，基类的构造函数也会被自动调用；其二，基类构造函数体的执行要早于派生类的构造函数体<sup>8</sup>。

## 析构

不只是构造函数，派生类的析构函数在调用时，也会调用基类的析构函数。我们同样根据一个例子来看它的效果。

```

struct Base {
    ~Base() { //Base的析构函数
        std::cout << "Base::~Base() is called." << std::endl;
    }
};

struct Derived : Base {
    ~Derived() { //Derived的析构函数
        std::cout << "Derived::~Derived() is called." << std::endl;
    }
};

int main() {
    Derived de;
}

```

这段代码的运行结果如下：

---

```

Derived:: Derived() is called.
Base:: Base() is called.

```

---

我们发现，对象销毁时析构函数的调用顺序，与对象创建时构造函数的调用顺序，刚好是相反的。在析构的时候，派生类的析构函数先调用，然后才是基类的析构函数。

也正因为，派生类的析构函数总是要调用基类的析构函数，所以我们根本不需要在写派生类时还要为了“基类成员的内存泄漏”而提心吊胆（只要你写的基类没有这个问题）。

```

class stack : std::vector<int> { //class的默认继承方式为private
public:
    //...
    ~stack() {} //无需为std::vector<int>的动态内存而担忧！自有它的析构函数来回收
};

```

---

<sup>8</sup>注意，这是函数体的执行顺序！这个实验并没有验证初值列的顺序。

### 9.3 私有继承

我们先来看看私有继承的相关知识。至于公开继承，我们的下一章几乎都是在借助它来进行讲解，所以现在就先不谈了。

与公开继承不同，私有继承一般不用来表示“属与种”的关系（Is-a relationship）。这是因为，在私有继承的条件下，基类的公有成员也只能作为派生类的私有成员存在，对外完全不可见。举个例子吧，所有的狗都会吠，所以我们应该把它作为狗的共性，置于 Dog 类下。

```
class Dog {
    //...
public:
    void bark() {} //吠叫本领
};

class Husky : public Dog { /*...*/ };
class Retriever : public Dog { /*...*/ };
```

如果是公开继承的话，Husky 和 Dog 类的对象可以在外部调用基类的 bark 成员函数。这样体现的正是“派生类对象也是基类对象”的效果。

但假如是私有继承呢？比如说这么写吧：

```
class Husky : Dog { /*...*/ }; //默认按私有方式继承
```

这就会出现一个问题：Dog::bark() 成为 Husky 的私有成员，它只对内可见，对外不可见。所以对外来说，相当于 Husky 类的对象没有吠叫本领，这显然是不合理的！那么私有继承适合表示什么关系呢？我们来看一眼这个例子：

```
class string { //class的默认成员访问权限是private，所以private可省略
class Arr {
    char *_p;
public:
    Arr(char* ptr = {nullptr}) : _p {ptr} {} //Arr的构造函数
    char*& p() { return _p; } //当arr不是常量时调用
    const char* p() const { return _p; } //当arr是常量时调用
} _arr {new char[_cap]}; //象征指针的成员，默认用new char[_cap]初始化
//...
```

读者应该对此比较熟悉，这是我们在第八章中讲讲过的 string 类，这里是一个代码片段。请你想一想，这里的 Arr 类对象 \_arr 是否满足以下特点：

- \_arr 的 public 成员对 string 来说好像是私有的<sup>9</sup>，它们对 string 类来说可见，但对 string 以外的部分来说完全不可见。
- \_arr 的 private 成员对 string 来说是不可见的。

而这个特点和私有继承是很相似的：

- 基类的 public 成员对派生类来说是私有的。它们对派生类来说可见，但对外完全不可见。
- 基类的 private 成员对派生类来说是不可见的。<sup>10</sup>

<sup>9</sup>虽然这里声称“\_arr 的成员也是 string 类的成员”显得不太合理，但我们讨论的只是可见性。

<sup>10</sup>如果你认为基类的私有成员对派生类来说也属于成员，那么这些成员是“不可见成员”；如果你认为基类的私有成员对派生类来说不属于成员，那么一切顺理成章。

私有继承表示的正是这种关系：一个对象是另一对象的一部分，`_arr` 是 `string` 对象的一部分。所以它表示的是“整体与部分”的关系（Has-a relationship）。

下面我就以 `Arr` 和 `string` 为例，改写上述代码：

```
class Arr {
    char *_p;
protected:
    Arr(char *ptr = {nullptr}) : _p {ptr} {} //Arr的构造函数
    //单独的Arr对象可能没有意义，所以把构造函数设为protected，使其仅对派生类可见
public:
    char*& p() { return _p; } //当Arr的对象不是常量时调用
    const char* p()const { return _p; } //当Arr的对象是常量时调用
};

class string : private Arr { //private是默认的，可不写
    static constexpr std::size_t npos = -1;
    std::size_t _cap {0};
    std::size_t _size {0};
};
```

读者可以看到，在这种情况下，`Arr` 类的对象是内嵌在 `string` 对象中的。`Arr` 中公有和受保护成员的部分，包括构造函数和重载的 `p` 函数，在 `string` 类中都是私有的。至于 `_p`，它在 `string` 类中完全不可见，只能通过 `p()` 来访问。

与原来那种定义方式（我们可以把它称为“组合方式”）不同，在私有继承方式下，这个内嵌的 `Arr` 对象没有名字；不过我们也不需要名字，直接把它当成自己的成员使用就行了。以下是对 `string::swap` 的改写，读者可以从中窥见它们在用法上的区别。

```
//组合方式
void string::swap(string &str) {
    std::swap(_cap, str._cap);
    std::swap(_size, str._size);
    std::swap(_arr.p(), str._arr.p()); //p()不是自己的成员，需要借助_arr成员调用
}

//私有继承方式
void string::swap(string &str) {
    std::swap(_cap, str._cap);
    std::swap(_size, str._size);
    std::swap(p(), str.p()); //别见外，把p()当成自己的成员函数来调用
}
```

## 构造与初始化

不过，因为这个内嵌的 `Arr` 对象没有名字，也没有显式的定义语句，所以我们不能像定义 `_arr` 时那样，为它提供一个默认成员初始值——不过，我们依然可以用初值列的方式来为它进行初始化：

```
string::string(std::size_t count, char ch)
    : Arr {new char[count]}, _cap {count} //注意new char[count]
{
    for (; _size < count; _size++)
        at(_size) = ch;
```

```
}
```

这里我们初始化 Arr 对象的方式也是调用基类的构造函数，即 `Arr{new char[count]}` 这样。读者需要注意，在这里我们传入的参数不是 `new char[_cap]` 了，这是因为初值列的初始化顺序是先基类成员，再派生类成员。简单点说就是，基类成员 `_p` 会在派生类成员 `_cap` 之前初始化，所以这时就不能依赖 `_cap` 来为 `_p` 初始化，只能另谋出路。

如果我们依然想要用 `_cap` 来为 `_p` 初始化，那么我们不妨把它也改为基类的受保护成员。

```
class Arr {
protected:
    std::size_t _cap;
    Arr(std::size_t cap, char *ptr = {nullptr})
        : _cap {cap}, _p {ptr} {} //Arr的构造函数
private:
    char* _p;
public:
    char*& p() { return _p; } //当Arr的对象不是常量时调用
    const char* p()const { return _p; } //当Arr的对象是常量时调用
};
```

但是这里存在另一个问题：C++17 标准并没有保证不同访问权限的成员间的初始化顺序<sup>11</sup>——换句话说，这样会把初始化变成一个未定义行为。为了预防这种情况的发生，我们还是要把 `_arr` 和 `_p` 都放在同一访问权限之下。

然而，如果把它们都放在 `private` 访问权限下，那么 `_cap` 就不能被派生类使用，这不符合我们的初衷；如果把它们都放在 `protected` 访问权限下，那么 `_p` 就可以被 `string` 类直接接触，起不到隔离的作用。为此，我们还需要更加取巧的方法才行，那就是——引用。

```
class Arr {
    std::size_t ori_cap; //通过构造函数的初值列初始化
    char* _p = new char[ori_cap]; //默认成员初始值，依赖于ori_cap
protected:
    std::size_t &_cap = {ori_cap}; //_cap是对ori_cap的引用，置于protected下
    Arr(std::size_t cap = {0})
        : ori_cap {cap} {} //Arr的构造函数
public:
    char*& p() { return _p; } //当Arr的对象不是常量时调用
    const char* p()const { return _p; } //当Arr的对象是常量时调用
};
```

像这样，我们实际上初始化的是 `ori_cap`，但是可以通过一个受保护的引用 `_cap` 来访问和修改这个值。这样，我们就在保证了正确初始化顺序的前提下，同时让 `_p` 和 `_cap` 都有了适当的访问权限。

接下来我们应该删去 `string` 类中的 `_cap` 定义，然后修改构造函数，把有关 `_cap` 的信息递交到 `Arr` 的构造函数来处理。

```
string::string(std::size_t count, char ch) : Arr {count} {
    for (; _size < count; _size++)
        at(_size) = ch;
}
```

<sup>11</sup>这个问题貌似在 C++23 标准中得到了解决。

这里我们只需要显式调用 `Arr` 的构造函数就行了。至于 `_size` 的初始化，自有默认成员初始值来招待。

### 实操：适配 `std::vector<int>` 的 `stack` 类

`std::stack` 是一种容器适配器<sup>12</sup>，用来表示“栈”这个数据结构。

栈（Stack）是一种数据结构，它只支持两种基本的修改操作：从末端堆入数据（Push），或者从末端移出数据（Pop），如图 9.4 所示。C++ 在 `stack` 库中定义了 `std::stack` 类模版，读者可以在 [cppreference.com](http://cppreference.com) 中查到它支持的功能等有关信息。关于 `stack` 类的具体实现，



图 9.4: 栈的数据操作方式：从末端堆入、从末端移出

图片来源：Wikipedia

我们当然可以从一个裸的数组开始写起，但是这样太麻烦了。为此，我们可以让 `stack` 类内嵌一个 `std::vector` 对象，从而通过调用 `std::vector` 现成的成员函数来实现我们想要的基本功能。

注意，这些 `std::vector` 的功能仅限 `stack` 类内部使用；而对于外部来说，不是所有的 `std::vector` 成员函数都能调用。因此我们可以用私有继承的方式来实现，这样就保证了 `std::vector` 的公有成员都是 `stack` 的私有成员，从而起到屏障效果。

```
class stack : private vector<int> { //private是默认的，可省略
    //...待补充
};
```

接下来我们就看一看，`std::stack` 有哪些功能，作为我们自己写 `stack` 类的参考。

### 功能简介

`std::stack` 的内部功能是基于 `std::deque`（双端队列）来实现的，但是我们还不熟悉这个容器，所以用 `std::vector` 来实现也未为不可。

`std::stack` 类的构造函数有很多，不过大多数都要用到我们还没学过的内容，所以一番挑拣下来，其实也就寥寥几个：

<sup>12</sup>容器适配器（Container Adapter），是一种在特定容器类型上提供特定接口的数据结构，它们通过包装底层容器（如数组或链表）来提供特定的功能。

- 默认构造函数将建立一个空的栈。
- 拷贝构造函数接收另一个同类<sup>13</sup>对象。
- 有的构造函数接收一个 Container 类对象——对于我们来说，这个 Container 指的就是 std::vector<int>。
- .....

std::stack 的析构函数也很简单。如果 stack 类没有什么动态内存操作的后顾之忧，那么我们不需要自行写一个析构函数，用默认的析构函数足够了。至于 std::vector<int> 中的动态内存空间，那应该是 std::vector<int> 的析构函数负责处理的。

赋值运算符很简单，只有从同类对象复制内容过来的版本。

top 成员函数可以返回当前栈的末尾变量的引用。而常成员函数的版本返回的是常量引用。

empty 返回一个 bool 变量，用来表示当前栈是否为空。

size 返回一个 std::size\_t 变量，用来表示当前栈的数据量。

push 用来在栈的末端堆入数据；而 pop 正相反，用来在栈的末端移出数据。

swap 函数是我们的老朋友了。

比较运算符也还是老样子，做字典序比较就行了。

读者能感觉到这个项目比前面的 string 要简单多了，那么事不宜迟，我们马上来规划一下要怎么做吧。

## 规划

我们还是按照先声明后定义的顺序来构建这段代码。除非定义部分很简单，否则我们都把它写在类外。

对于私有成员部分，我们有一个私有继承的 std::vector<int> 类内嵌对象，已经足够。

而在公有成员部分，我们首先需要定义构造函数：

```
public:
    explicit stack() {} // 什么也不做
    stack(const stack&); // 拷贝构造函数
    explicit stack(const std::vector<int>&); // 接收一个std::vector<int>对象
```

其中的两个函数被冠以 explicit，这就限制了它们的隐式类型转换操作。其中的默认构造函数即便什么也不做，也会调用基类 std::vector<int> 的默认构造函数。

至于析构函数，我们没有必要自定义，直接用默认版本就足够了。

接下来是赋值运算符。

```
public:
    stack& operator=(const stack&); // 赋值运算符
```

然后是 top 函数。注意，它分为常成员函数版本和非常成员函数版本，其返回类型是不同的。

```
int& top() { return back(); } // 调用std::vector<int>的back函数
const int& top() const { return back(); } // 常成员函数版本
```

在这个函数的定义中我们看到，我们是直接调用 back 函数来修改基类成员的。stack 函数内部并未定义 back 函数，所以根据名称查找规则<sup>14</sup>，它会找到 std::vector<int> 的 back 成员函数。

如果你想要避免的歧义，也可以通过作用域解析操作符来显式地指定我们要调用哪个成员函数：

<sup>13</sup>需要提醒，std::stack 是一个类模版而不是类。std::stack<int> 和 std::stack<double> 同属一个类模版，但却是两个不同的类。

<sup>14</sup>读者可以回忆一下我们在第七章第三节中讲过的关于“名称查找”的内容。对于继承关系来说，也有名称查找规则：如果在这个类中找不到这个名称，那么就去它的基类中查找。

```

int& top() {return std::vector<int>::back(); }
//显式指定调用 std::vector<int> 的 back 成员函数

empty 成员函数和 size 成员函数也不难。唯独需要注意一点——

bool empty() const { return size(); } //调用谁的 size 函数？
std::size_t size() const { return std::vector<int>::size(); }
//调用 std::vector<int> 的 size() 函数

```

先来看看这个 size 成员函数。我们发现，这个函数的名字与 `std::vector<int>::size()` 相互冲突。如果我们直接写 `size()` 的话，编译器会根据名称查找规则，找到 `stack::size()`，所以在这里就会发生 `stack::size()` 的无穷递归调用，进而发生错误。

为了解决这个问题，我们必须显式地让编译器知道我们真正想调用的是哪个函数，所以我们要写成 `std::vector<int>::size()`。至于 `empty` 函数，我们可以有很多种实现方法。除了上文中调用 `stack::size()` 加隐式类型转换的方法以外，还有这些选择：

```

bool empty() const { return std::vector<int>::size(); }
//调用 std::vector<int> 的成员函数，再隐式类型转换

bool empty() const { return std::vector<int>::empty(); }
//std::vector<int> 也有 empty() 成员函数，可以直接用它

```

读者选其中一种来实现就可以了。

`push` 接收一个 `const int&` 参数，而 `pop` 不接收参数，所以这样写就行了。

```

void push(const int&); //堆入数据
void pop(); //移出数据

```

`swap` 成员函数亦如此。

```
void swap(stack&); //交换内容
```

比较运算符还是一如既往地利用代码重用来实现。这里除了 `std::lexicographical_compare` 以外，我们还多了别的选择，我们稍后再讲。

```

bool operator<(const stack&) const;
bool operator>(const stack&) const;
bool operator<=(const stack&) const;
bool operator>=(const stack&) const;
bool operator==(const stack&) const;
bool operator!=(const stack&) const; //六个比较运算符

```

## 实现

接下来把没定义的成员函数定义一下。首先是构造函数和拷贝构造函数。

```

stack::stack(const std::vector<int> &v) : std::vector<int>{v} {}

//构造函数

stack::stack(const stack &s) : std::vector<int>{s} {}

//拷贝构造函数

```

这里的构造函数调用了 `std::vector<int>` 的拷贝构造函数，用以为 `stack` 类内嵌的 `std::vector<int>` 对象初始化。而下面的拷贝构造函数就显得有些反常，它传入的参数是 `stack` 类的？

其实这是因为，基类与派生类之间可以进行隐式类型转换。我们会在下一章中讲解这方面的详细知识；现阶段读者只需要明白，派生类的对象可以无风险地隐式类型转换成基类对象——得到的正是

那个内嵌于派生类中的基类对象<sup>15</sup>。所以这里传入的 s 会先隐式类型转换成 `std::vector<int>` 类型，然后用以构造。

```
stack& stack::operator=(const stack &s) {
    if (this == &s) // 防止自我赋值
        return *this;
    std::vector<int>::operator=(s);
    // 调用 std::vector<int> 的赋值运算符，并借助隐式类型转换参数传递
    return *this;
}
```

在赋值运算符中我们也使用了相同的技巧。我们先是通过作用域解析运算符指定了要调用谁的赋值运算符，然后又通过 `stack` 到 `std::vector<int>` 的隐式类型转换成功地把参数传递过去了。

接下来提供 `push` 和 `pop` 函数的定义。`std::vector` 中并没有 `push` 和 `pop` 成员，所以我们定义函数时不需要担心出现重名问题，可以不必再使用作用域解析运算符。

```
void stack::push(const int &val) {
    push_back(val); // 将调用 std::vector<int>::push_back
}
void stack::pop() {
    pop_back(); // 将调用 std::vector<int>::pop_back
}
```

当然，如果你愿意，也可以写成 `std::vector<int>::push_back` 这样。

```
void stack::swap(stack &s) {
    std::vector<int>::swap(s);
}
```

`swap` 成员函数的定义也是这样进行的，这里就不再赘述。

```
bool stack::operator<(const stack &s) const {
    return *this < static_cast<std::vector<int>>(s);
}
```

小于号这样定义就可以。需要注意，对于 `std::vector` 来说，小于号并不是成员函数，而是非成员函数，因此我们不能用成员函数的调用方法。

另外读者可能会对 `*this<static_cast<std::vector<int>>(s)`<sup>16</sup> 这样的写法感到好奇——难道这样不会像我们在第二章第四节末尾所讲那样<sup>17</sup>，`s` 被转换成 `std::vector<int>` 类型后又被转换成 `stack` 类型吗？其实不会，这是因为继承关系所带来的隐式类型转换是不平等的。派生类的对象可以无风险转换为基类对象<sup>18</sup>，但基类对象就不能隐式转换为派生类对象。所以对于 `*this<static_cast<std::vector<int>>(s)` 来说，只有一种可能的转换方式，就是二者都转换为 `std::vector<int>` 对象。

接下来的几个比较运算符就是小儿科了。

```
bool stack::operator<(const stack &s) const {
    return *this < static_cast<std::vector<int>>(s);
}
```

<sup>15</sup> 在类的内部，这个关系是成立的；对于公有继承来说，这个关系也成立。但是对于私有继承的派生类来说，在类外不允许这种隐式类型转换。

<sup>16</sup> 请读者留意，一些上古编译器会把两个半尖括号 `>>` 识别成右移位运算符。如果发生了这样的事情，读者只需在两个半尖括号之间加一空格，改写成 `> >` 就可以了。相似的情形还有两个半方括号 `[[` 这样的写法，可能会被编译器当作属性说明符，这时我们只需要加一空格就可以了。

<sup>17</sup> 我们曾就 `static_cast<int>(a)*b` 这个语法做过讨论，发现在这个类型转换的过程中，`a` 被转换为 `int` 型之后马上又被转换为 `double`，结果并没有实现我们的目的。

<sup>18</sup> 再次提醒，对于类内部来说才有这样的隐式类型转换关系；对于类外部来说就未必了。

```

}

bool stack::operator> (const stack &s) const {
    return s < *this;
}

bool stack::operator<=(const stack &s) const {
    return !(s < *this);
}

bool stack::operator>=(const stack& s) const {
    return !(*this < s);
}

bool stack::operator==(const stack &s) const {
    return !operator!=(s); // 利用已经声明的stack::operator!=
}

bool stack::operator!=(const stack &s) const {
    return *this < s || s < *this; // 比较运算符的优先级高于逻辑或运算符
}

```

好了，这样我们就完成了一个基本的 `stack` 类。

## 组合方式，还是继承方式？

如果我们要表示一个类的私有对象，那么有两种方式可供我们进行选择：一种是在我们之前用的组合方式（Containment），一种是我们刚才看到的私有继承方式（Private inheritance）。

绝大多数程序员都会毫不犹豫地选择组合方式，因为：

- 组合方式下，每个成员对象都有名字。我们可以很方便地知道哪个对象是什么、意味着什么，以及是哪个对象在调用成员函数——试想，我们刚才调用 `size()` 的时候很容易陷入到无穷递归的陷阱中；如果每个成员对象有各自的名字，那么我们一眼就可以看出我们正在调用的函数是什么，属于谁。
- 组合方式下，我们可以定义多个成员对象。虽然有些时候我们只需一个对象就够了——`stack` 正是这样的情形；但是如果我们需要内嵌多个对象，那么私有继承方式就不管用了，我们必须组合。

无论怎么说，这些优点都太显著，所以我们常常会忽略私有继承这种方式。

但是，私有继承方式也有它的些许优点：

- 如果我们希望像 `Arr` 那样只把 `p()` 当作一个随机而变的成员对象来使用的话，定义一个 `_arr` 作为中介还是麻烦了点。不如通过继承方式，让 `p()` 成为真正意义上的成员。
- 在继承方式下，基类的受保护成员对派生类也可见，这就为我们在某些特定条件下写代码提供了便利。
- 继承方式还为多态提供了可能，我们会在下一节中讲到。

如果要我描述的话，我认为私有继承相较于组合方式来说，有点像 `mutable` 成员相较于普通成员——如果没有什么这方面的专门需求，就不要用它。虽说如此，但是私有继承作为一种行之有效的表示“整体与部分”关系的方式，仍是读者应知应会的内容。

## stack 类的初步实现代码

代码 9.1: Header.h

```
#pragma once
#include <vector>
class stack : std::vector<int> {
public:
    explicit stack() {} //什么也不做
    explicit stack(const std::vector<int>&); //接收一个std::vector<int>对象
    stack(const stack&); //拷贝构造函数
    //用默认析构函数足矣，无需自行定义
    stack& operator=(const stack&); //赋值运算符
    int& top() { return back(); } //调用std::vector<int>的back函数
    const int& top() const { return back(); } //常成员函数版本
    bool empty() const { return size() == 0; } //调用谁的size函数？
    std::size_t size() const { return std::vector<int>::size(); }
    //调用std::vector<int>的size()函数
    void push(const int&); //堆入数据
    void pop(); //移出数据
    void swap(stack&); //交换内容
    bool operator<(const stack&) const;
    bool operator>(const stack&) const;
    bool operator<=(const stack&) const;
    bool operator>=(const stack&) const;
    bool operator==(const stack&) const;
    bool operator!=(const stack&) const; //六个比较运算符
};

```

代码 9.2: Definition.h

```
#include "Header.h"
stack::stack(const std::vector<int> &v) : std::vector<int>{v} {}
//构造函数
stack::stack(const stack &s) : std::vector<int>{s} {}
//拷贝构造函数
stack& stack::operator=(const stack &s) {
    if (this == &s) //防止自我赋值
        return *this;
    std::vector<int>::operator=(s);
    //调用std::vector<int>的赋值运算符，并借助隐式类型转换参数传递
    return *this;
}
void stack::push(const int &val) {
    push_back(val); //将调用std::vector<int>::push_back
}
void stack::pop() {
    pop_back(); //将调用std::vector<int>::pop_back
}
```

```

void stack::swap(stack &s) {
    std::vector<int>::swap(s);
}

bool stack::operator<(const stack &s) const {
    return *this < static_cast<std::vector<int>>(s);
}

bool stack::operator>(const stack &s) const {
    return s < *this;
}

bool stack::operator<=(const stack &s) const {
    return !(s < *this);
}

bool stack::operator>=(const stack& s) const {
    return !(*this < s);
}

bool stack::operator==(const stack &s) const {
    return !operator!=(s); // 利用已经声明的 stack::operator!=
}

bool stack::operator!=(const stack &s) const {
    return *this < s || s < *this; // 比较运算符的优先级高于逻辑或运算符
}

```

## 9.4 多级继承

任何一个自定义类型都可以作为基类，被其它类继承，从而成为其它类的一部分<sup>19</sup>。对于公有继承来说，这相当于是对基类对象所拥有的共性，添加了一些派生类对象额外的特性；对于私有继承来说，这相当于是把一个基类的对象内嵌到派生类中，作为它的一个成员。

接下来我们还可以把派生类再作为基类，用它来继承新的类。我们把这种操作叫作多级继承（Multilevel inheritance）。

```

struct A { /*...*/ };

struct B : A { /*...*/ }; // B 继承自 A
struct C : B { /*...*/ }; // C 继承自 B
class D : C { /*...*/ }; // D 继承自 C

```

看上去好像很复杂，但是读者只需要把握刚才那点就可以理顺各种关系。对于这个继承关系来说，B 继承了 A 的共性又有自己的特性；C 继承了 B 的共性并有了自己的特性；D 稍有不同，它私有继承自 C，表示的是 D 中内嵌了一个 C 对象作为其成员。

当然，这是一个虚构的例子，不那么形象。现在我们就来看两个真实的例子。

### 生物分类法

图 9.5 是赤狐（Red fox, *Vulpes vulpes*）在生物分类法中的位置。它属于真核域 Eukaryota - 动物界 Animalia - 脊索动物门 Chordata - 哺乳纲 Mammalia - 食肉目 Carnivora - 犬科 Canidae - 狐属 *Vulpes*。

这里面有好复杂的层级，但我们不难看出，它们之间是通过广义的“属与种”关系串连起来的：

<sup>19</sup>有一种例外：标注为 **final** 的类不可以被继承。但本书不打算讲解 **final** 的相关内容。

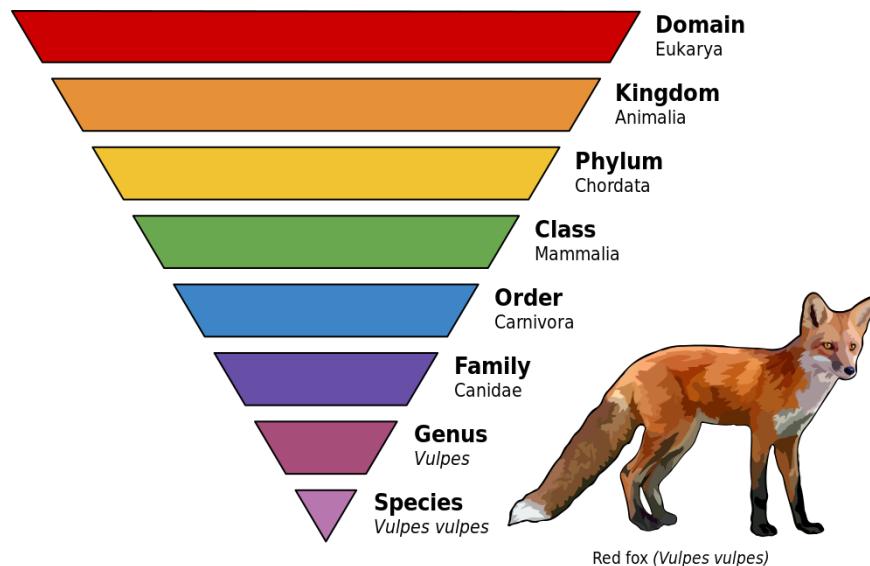


图 9.5: 赤狐的生物分类  
图片来源: Wikipedia

- 狐属是犬科的一种，它既有犬科的共性，又有狐属的独特性——比如蓬松的尾巴；
- 犬科是食肉目的一种，它既有食肉目的特性，又有犬科的独特性——比如较长的颅骨；
- .....
- 动物界是真核域的一种，它既有真核域的特性，又有动物界的独特性——比如运动能力。

所以我们可以用一套复杂的公有继承关系来描述生物分类。

```

class Eukarya { // 真核域
protected:
    class Nucleus { /*...*/ } nucleus; // 真核域生物均有细胞核
    //...
};

class Animalia : public Eukarya { // 动物界， 公有继承自真核域
public:
    virtual void move(); // 动物界生物均有运动能力（我们会在下一章讲virtual）
    //...
};

class Chordata : public Animalia { // 脊索动物门， 公有继承自动物界
protected:
    class Notochord { /*...*/ } notochord; // 脊索动物门生物均有脊索
};

class Carnivora : public Chordata { // 食肉目， 公有继承自脊索动物门
    //...
};

class Canidae : public Carnivora { // 犬科， 公有继承自食肉目
    //...
};

class Vulpes : public Canidae { // 狐属， 公有继承自犬科
    //...
};

```

```

};

class Vulpes_vulpes : public Vulpes { //赤狐种，公有继承自狐属
    //...
};

}

```

像这样,我们就可以通过继承关系建立起一个复杂的树状结构。所以一个赤狐种的对象具有从 `Eukarya` 类到 `Vulpes` 类的所有公有和受保护成员。

如果我们还想要添加新的分支类,只需要通过继承关系把它们接到适当的类中,就可以拥有这个类及其基类的全部成员。举个例子说,犬科还有一条分支是犬属 `Canis` - 狼种 `Canis lupus` - 家犬亚种 `Canis lupus familiaris`。所以我们可以在 `Canidae` 之下再继承一个分支。

```

class Canis : public Canidae { //犬属，公有继承自犬科
    //...
};

class Canis_lupus : public Canis { //狼种，公有继承自犬属
    //...
};

class Canis_lupus_familiaris : public Canis_lupus { //家犬亚种，公有继承自狼种
    //...
};

```

这样一来,赤狐种 `Vulpes_vulpes` 和家犬亚种 `Canis_lupus_familiaris` 就在相同的特征方面继承同一个类,而在各自的特性上又有自己的一套。就这样,面对这样错综复杂的关系,C++ 可以通过继承功能,把它们梳理得十分清晰。

## C++ 流输入/输出库

C++ 的流输入/输出库是一个系统,包含 `iostream`, `fstream`, `sstream` 等许多库文件,`std::istream`, `std::ostream` 等许多我们已经熟知的类模版或类。它们之间有一套复杂的继承关系,见图 9.6。其中,`std::istream` 是 `std::basic_istream` 类模版的一个实例,定义为 `std::basic_istream<char>`;`std::ostream` 是 `std::basic_ostream` 类模版的一个实例,定义为 `std::basic_ostream<char>`。

我们曾经用 `std::cout.setf(std::ios_base::boolalpha)` 来改变 `std::cout` 对 `bool` 数据的输出方式,这里的 `setf` 和 `fmtflags` 其实都是定义在 `std::ios_base` 类中的。`std::ostream` 间接继承了 `std::ios_base` 类,所以它可以把 `std::ios_base::setf` 当作自己的成员函数来调用。

我们也曾用过 `std::cin.clear()` 这样的写法,这里的 `clear` 是定义在 `std::basic_ios<CharT,Traits>` 类中的一个成员函数。`std::istream` 类继承了 `std::basic_ios<CharT,Traits>`,所以它也能调用这个成员函数。

敏锐的读者可能发现了,图 9.6 中的 `std::basic_iostream` 类同时继承了 `std::basic_ostream` 和 `std::basic_istream` 类。这种操作叫作多重继承,我们会在第十章中讲解有关问题。

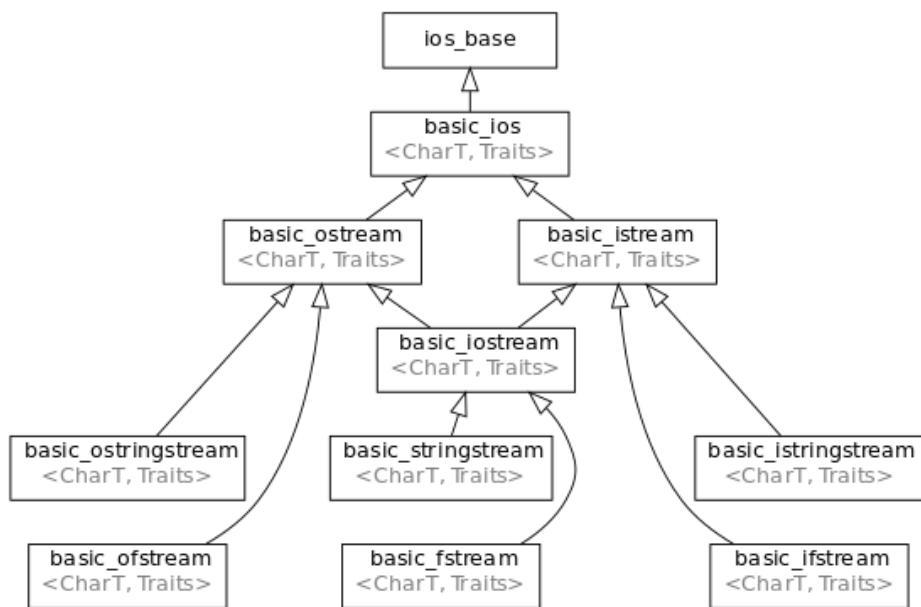


图 9.6: C++ 流输入/输出库中的继承关系

# 第十章 继承中的常见问题

在上一章中，我们讲解了继承的基本语法和基本概念。本章我们继续深入，以期进一步搞清楚继承概念之下的许多常见问题——

- 继承关系中的类型转换是怎么进行的？又有什么规则？
- 多态又是怎么一回事？虚函数到底在做什么？
- 如果描述基类所需要的成员比描述派生类的还要多，那怎么办？
- 多重继承是什么？怎么理解？又有什么用？
- 棱形继承关系中，基类的成员重复了怎么办？

这些问题在实际编程中很常见，所以我们需要好好研究一下，以防将来真的用的时候你突然一拍脑门说：“哎呀，我没学过这玩意。”那就有点麻烦了。

本章的内容量非常大。硬度和第八章相仿，也请读者做好心理准备。

## 10.1 继承中的类型转换

无论对于何种继承方式，我们都可以作这样的理解：一个派生类对象当中内嵌了一个基类对象。这个基类对象只拥有基类的成员；而派生类的对象拥有基类成员在内的所有成员。在上一章的代码中，我们也反复看到了派生类对象到基类对象的隐式类型转换。那么，这个转换在什么条件下是可以的，什么条件下是禁止的？基类对象能转换为派生类对象吗？对于指针和引用来说，情况是否又会有些不同呢？这些问题的答案就在本节当中。

### 对象的类型转换

在继承时，如果我们没有人为提供转换构造函数或者自定义转换函数，那么编译器就会生成一个隐式的，从派生类到基类的类型转换函数。这个类型转换函数允许派生类的对象通过 `static_cast` 转换为基类的对象。

```
struct Base { /*...*/ };
struct Derived : Base { /*...*/ }; // 内置了一个Derived到Base类型的转换函数
```

但是，这个转换函数的可见性是取决于继承方式的：

- 如果 `Derived` 类公开继承自 `Base` 类，那么这个转换函数相当于是 `Derived` 的公有成员。
- 如果 `Derived` 类受保护继承自 `Base` 类，那么这个转换函数相当于是 `Derived` 的受保护成员。
- 如果 `Derived` 类受私有继承自 `Base` 类，那么这个转换函数相当于是 `Derived` 的私有成员。

对于上一章中的 Husky 类的对象来说，这个类型转换函数是 Husky 类的公有成员，所以我们可以在任何一个地方把 Husky 类的对象转换成 Dog 类。

而上一章中的 stack 则不同，我们只能在 stack 类当中进行类型转换；而在类之外，这个转换函数不可见，我们不能进行 stack 到 std::vector<int> 的类型转换。

那么，基类对象能否隐式类型转换为派生类对象呢？答案是否定的。原因也很简单：无论何种继承方式，派生类对象所需要的信息量都比基类对象更多<sup>1</sup>。也就是说，如果要把派生类对象转换为基类对象，我们不需要提供额外信息，单纯把内嵌于其中的基类对象挖出来就足够了；但如果把基类对象转换为派生类对象，我们就需要提供额外的信息，否则编译器就不知道要如何补全缺失数据。

所以，如果我们希望允许基类到派生类之间的类型转换，就必须自行设计转换构造函数或者自定义转换函数。还是以我们写的 stack 为例，我们就在 stack 类中定义了一个 explicit 的转换构造函数。

```
public:
explicit stack(const std::vector<int>&); // 接收一个std::vector<int>对象
```

这样一来我们就可以用显式类型转换把基类 std::vector<int> 对象转换成派生类 stack 对象了。

## 指针/引用的向上类型转换

对于指针/引用类型来说，情况会稍显复杂。

我们还是来想一下内存吧。对于一个派生类的对象来说，在它的内存空间中，既有属于基类的成员对象，又有属于派生类的成员对象。比如下面这段代码，作为基类成员的 Base::a 在 Derived 对象的内存空间之中拥有自己的一席之地。

```
class Base {
    int a;
};

class Derived : public Base{
    int b;
};

int main() {
    std::cout << sizeof(Base) << 'u' << sizeof(Derived);
}
```

这段代码的运行结果如下：

---

4 8

---

你瞧，Derived 所占用的内存空间大小为 8 字节，这说明它包含了 Base::a 和 Derived::b 两个成员。那么当我们对一个 Derived\* 指针取内容时，程序将会取该字节起的 8 个字节作为一个 Derived 对象。

那么假如我把它转换成 Base\* 指针然后取内容呢？程序只会取该字节起的 4 个字节作为一个 Base 对象。

所以我们不难发现，因为 Base 对象是内嵌在 Derived 对象当中的，我们就可以用 Base\* 指针来取 Derived 对象中的基类部分，如图 10.1 所示。所以对于公有继承来说，我们可以把派生类的指针隐式类型转换为基类的指针。

---

<sup>1</sup>至少是不少于。

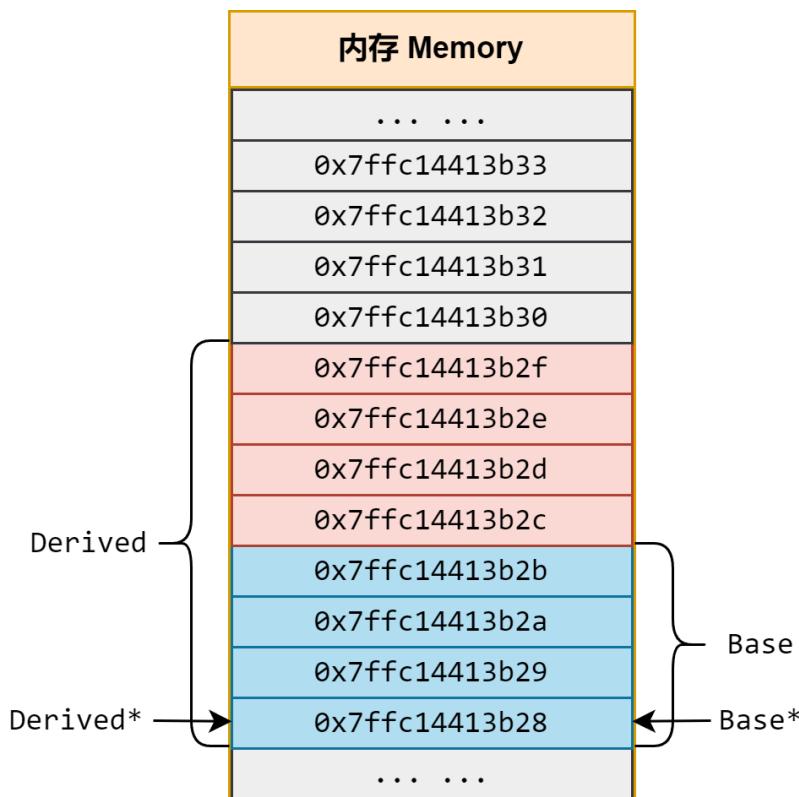


图 10.1: 用基类指针指向内嵌于派生类中的基类对象

```
Derived d {/*...*/};
Base *b = &d; //&d 被隐式类型转换为 Base* 类型，并赋值给 b
```

至于私有继承和受保护继承，它们的规律和对象之间的类型转换很相似——受保护继承的派生类指针只有在派生类及“派生类的派生类”中可以转换为基类指针；私有继承的派生类指针只能在这个类中转换为基类指针。

无论继承权限如何，我们都把这种派生类指针到基类指针的类型转换称为向上类型转换（Up-casting）<sup>2</sup>。向上类型转换总是安全的，因为基类指针访问的范围总是不大于派生类指针，所以就不会发生访问内存“越界”这样的问题。

引用也可以进行类型转换，道理相同：派生类的对象或者引用类型可以向上类型转换成基类的引用。所以如果我们要写一个对所有狗类都通用的函数，我们就不必为 Husky, Retriever 等类型写一大堆重载，只需要写 Dog 一个类就行了。

```
void fun(const Dog &dog) { //可以接收 Dog 类及其 public 派生类的对象
    //...
}

int main() {
    Husky mine {/*...*/};
    fun(mine); //Husky 对象到 Dog 引用的向上类型转换
}
```

这里我们看到的都是指针/引用的隐式类型转换。如果要做显式类型转换的话，我们用 `static_cast` 来实现就可以了。

<sup>2</sup>我们在习惯上会把继承关系图中的基类画在上方，派生类画在下方，因此得名。

## 指针/引用的向下类型转换

派生类的指针/引用可以转换为基类的引用，只要在特定继承方式下这种类型转换是可见的就行。相反地，基类的指针/引用也可以类型转换为派生类。但是相较于前者，这种基类到派生类的转换不仅多了限制，还可能存在危险。

基类指针/引用到派生类指针/引用的类型转换必须显式地进行——换句话说，你必须知道你正在做什么，不能稀里糊涂地就把类型转换给执行了。

为什么编译器要把这个功能限制地如此严格呢？这是因为，这种基类指针/引用到派生类指针/引用的向下类型转换（Downcasting）存在着访问越界的潜在风险。

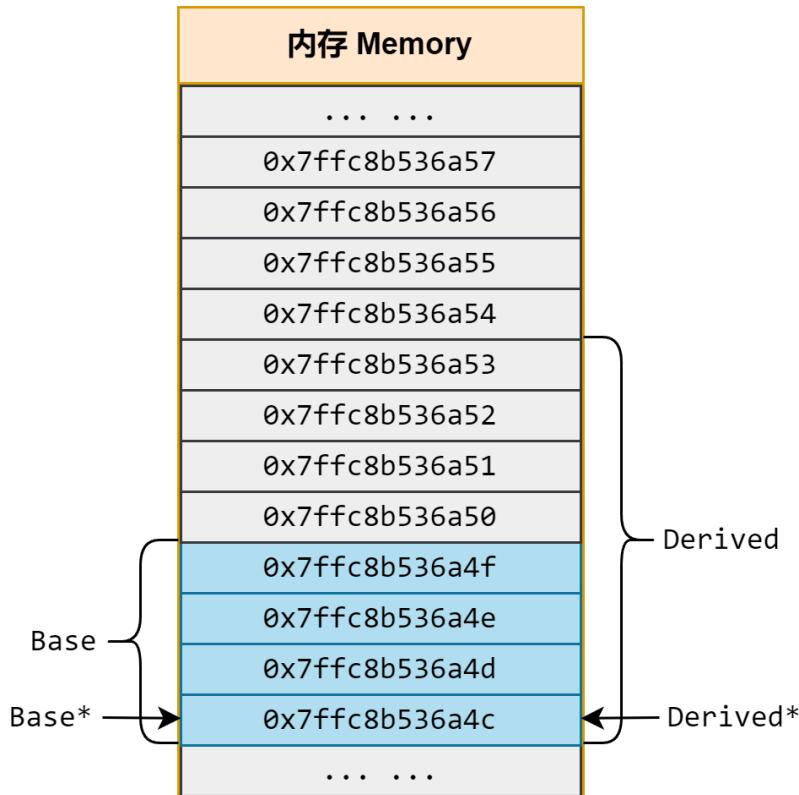


图 10.2: 用派生类指针指向基类对象，这是未定义行为

如图 10.2 所示，如果我们定义了一个 `Base` 类对象而用 `Derived*` 指针指向它，那么这个 `Derived*` 指针指向的内存范围除了这个 `Base` 对象以外，还有 4 个字节的其它空间。这 4 个字节可能存储了其它变量的部分信息，或者只是一些存储着杂乱无章信息的未使用空间。但无论如何，当我们试图访问这些空间中的内容时，将会得到不确定的结果——这是一种未定义行为。

那么什么情况下这种类型转换是有意义的呢？只有一种情况：这个指针原本就是指向派生类对象的，只不过它出于某种原因转换成了 `Base*` 而已。这样一来，我们就能确保这个基类指针到派生类指针的转换是安全的。但另一个问题在于，当我们拿到一个基类指针时，我们怎么才能知道这个基类指针到底是不是原本指向派生类对象的呢？很遗憾，因为指针的指向是可以任意改变的，所以我们不能在编译时确定它是不是指向派生类对象的。

`static_cast` 的解决方法是不解决——它只是死板地执行类型转换的操作而已，至于这里潜在的风险，它是不会管的。也正因如此，在向下类型转换的过程中，我们再使用 `static_cast` 已经不合适，必须寻求一种能在运行时判断类型并给出相应转换结果的方案，这就是动态类型转换 `dynamic_cast`。

动态类型转换可以保证程序能够安全地进行向下类型转换。当我们试图把基类指针转换为派生类指针时，程序会进行运行时检查：如果这个基类指针是指向派生类的，那么它可以成功地进行类型

转换；否则，这个类型转换只能返回一个 `nullptr`。当我们试图把基类引用转换为派生类引用时，程序会进行运行时检查：如果这个基类引用是对派生类对象的引用<sup>3</sup>，那么它可以成功地进行类型转换；否则，就会抛出 `std::bad_cast` 异常<sup>4</sup>。

`dynamic_cast` 的用法和 `static_cast` 非常相仿，但是我们先别急着用它。动态类型转换要求基类必须是多态的——接下来的一节马上就来讲解这个问题。

## 10.2 虚函数与多态

在上一节中我们遗留了有关动态类型转换的问题。不过请读者先不要迷失在语法之中，我们先把思绪拉回来，思考一下我们使用向下类型转换的目的。

### 为什么要向下类型转换？

我们说过，如果一个基类指针原本就指向一个基类对象的话，那么向下类型转换没有任何意义；只有当基类指针原本就指向一个派生类对象时，我们使用向下类型转换才有意义。

但是，你有没有在心里纳闷过哪怕一秒——这么转上去再转下来有意思吗？如果我们一开始就已经有一个派生类的指针/对象，那么为什么要先把它向上转换成基类的指针/引用，再向下转回去呢？我们直接用原来的指针/引用不就完了？

```
void fun(Base *pb) {
    Derived *pd {static_cast<Derived*>(pb)}; //Base* 类的pb再转换成Derived*
    //...
}

int main() {
    Derived de;
    fun(&de); //传入参数时&de被转换成Base*
}
```

这是何苦呢？如果只是为了完成这个功能，我们直接把形参改成 `Derived*` 就好了。

```
void fun(Derived *pd) { //更省事了
    //...
}

int main() {
    Derived de;
    fun(&de);
}
```

所以如果单纯是为了使用某个派生类的对象，那么咱犯不着还要通过基类指针/引用倒一遍。

那么向下转换是为了使用某个派生类独有的成员对象吗？问题同样在于，如果这个成员对象是派生类独有的，那么我们应该传入派生类的指针/引用，而不是拿基类来做文章；如果这个成员对象是所有派生类都有的呢，我们应该在设计之初就把它写到基类的成员对象当中，使它成为所有派生类的共性，而不是像刚才说的那样，把它写成每个派生类“共同的独特性”。

```
class Base {
protected:
    int common; //共性，使用它不需要向下类型转换
};
```

<sup>3</sup>其本质说白了还是指针的指向。引用都是靠指针来实现的。

<sup>4</sup>我们会在第十二章中讲解异常。

```
class Derived : public Base {
    int special; //独特性，如果是为了使用它，就不应当传入基类指针
};
```

我们发现，这个时候也是用不到向下类型转换的——不是不能用，而是太没必要了。

读者可能想到了另一种情况——静态成员。确实，不同的派生类可能有同名的静态成员。如果我们希望这个静态成员在不同派生类中有不同的值（举个例子吧，`_object_number` 表示这个类的对象数），那么我们就不能把它定义在基类当中——因为静态成员对象有内部链接，同一个类只能有同一个静态成员。那么不同派生类的同名静态成员就要定义在各自的派生类中。

```
class Base {
public:
    static std::size_t _object_number; //Base类的静态成员对象
protected:
    int common; //共性
}
class Derived : public Base {
public:
    static std::size_t _object_number; //Derived类的静态成员对象
}
```

这里候如果我们还想要根据 `Base*` 指针来访问派生类独有成员的话，就需要用向下类型转换了。

但是——实际上，如果我们想要访问静态成员的话，可以直接用作用域解析操作来完成。

```
std::cout << Derived::_object_number;
```

所以说这里其实也没有类型转换什么事。

最后还有一种可能，那就是成员函数。试想一种情况——我们需要为不同的派生类写同样名字的函数；但是，这些函数的定义又有所不同，所以我们不能把它们作为共性，合并到基类当中。

```
struct Shape { //这个类用来表示各种几何图形
    static constexpr double Pi {3.1415926}; //常量表达式Pi
    static constexpr double Deg2Rad(double deg) { //常成员函数，用于角度转弧度
        return deg * Pi / 180;
    }
    static constexpr double Rad2Deg(double rad) { //常成员函数，用于弧度转角度
        return rad * 180 / Pi;
    }
};
class Triangle : public Shape { //三角形是一类几何图形
    double _a;
    double _b;
    double _c;
public:
    Triangle(double a, double b, double c)
        : _a {a}, _b {b}, _c {c} {}
    double perimeter() const { return _a + _b + _c; } //周长
    double area() const { //面积
        double s {(_a + _b + _c) / 2};
        return std::sqrt(s * (s - _a) * (s - _b) * (s - _c));
    }
};
```

```

    }
};

class Circle : public Shape { // 圆形是一类几何图形
    double _r;
public:
    Circle(double r) : _r {r} {}
    double perimeter() const { return 2 * Pi * _r; } // 周长
    double area() const { return Pi * _r * _r; } // 面积
};

class Parallelogram : public Shape { // 平行四边形是一类几何图形
    double _a;
    double _b;
    double _theta; // 表示一个夹角的角度值
public:
    Parallelogram(double a, double b, double theta)
        : _a {a}, _b {b}, _theta {theta} {}
    double perimeter() const { return 2 * (_a + _b); } // 周长
    double area() const { return _a * _b * std::sin(Deg2Rad(_theta)); } // 面积
    // 注意C++标准库中三角函数接收的参数都是弧度值
};

```

读者可以看到，这里的三个派生类，每个类的 `perimeter` 函数和 `area` 函数都不尽相同——所以我们不能直接把这些函数写到 `Shape` 类中。所以在面对基类指针/引用时，我们进行向下类型转换就是必要的了。

## 多态与动态类型转换

基类指针/引用到派生类指针/引用的动态类型转换有一个前置条件，即基类必须是多态（**Poly-morphism**）的。在继承场合下，多态意味着一个基类指针会指向它的派生类对象，并且允许程序在运行时判断这个指针的实际指向，进而转换为指向其派生类的指针。

```

Shape *ps[3] {
    new Triangle{2,3,2},
    new Circle{2},
    new Parallelogram{1,5,90}
}; // 定义一个指针数组用来分配新内容
std::cout << dynamic_cast<Triangle*>(ps[0])->area() << std::endl;
std::cout << dynamic_cast<Circle*>(*ps[1]).area() << std::endl;
for (auto p : ps)
    delete p; // 记得回收

```

保证了基类的多态性后，我们就可以通过动态类型转换来实现这个功能了。无论指针还是引用均可以如此。

最简单的多态化方法就是把基类的析构函数变成 `virtual` 的虚函数。

```
virtual ~Shape() {} // 虚函数
```

只要有哪怕一个虚函数，这个类就是多态的——所以把析构函数定义成虚函数是我所认为的最佳方法。我们稍后再讲虚函数的有关细节。

如果我们需要写一个函数，它可以输出任何一个 `Shape` 派生类对象的面积信息，目前的方案就是把这个派生类对象的地址通过向上类型转换变成 `Shape*`——这样我们只需要写一个函数就可以了，不用写一大堆重载。

```
void output_shape_info(const Shape *sh) {
    //...待补充
}
```

这个方法还有点粗糙，因为我们需要在函数体中再判断它到底指向哪个派生类。为此我们就不得不再写一大堆代码。

```
double shape_area(const Shape *sh) {
    if (dynamic_cast<Triangle*>(sh)) //如果sh不指向Triangle对象，返回值为nullptr
        return dynamic_cast<Triangle*>(sh).area(); //调用Triangle::area()
    if (dynamic_cast<Circle*>(sh)) //如果sh不指向Circle对象，返回值为nullptr
        return dynamic_cast<Circle*>(sh).area(); //调用Circle::area()
    if (dynamic_cast<Parallelogram*>(sh)) //同上
        return dynamic_cast<Parallelogram*>(sh).area(); //同上
}
```

这样看上去……好像和我们分别写三个重载的工作量也没大差啊。

```
void shape_area(const Triangle &tri) {
    return tri.area();
}

//还有两个重载
```

这样单纯地用动态类型转换并没有让我们写代码变得多么方便；反而，`dynamic_cast` 加上各种信息和判断就够我们喝一壶的了。所以我们还需要寻求更简化的方法。这也就是虚函数最大的妙用，我们现在就来讲它。

## virtual 虚函数

让我们从头开始思考关于 `shape_area` 函数的问题。

显然，因为不同派生类的 `area` 函数彼此有一些差异（而且它们多少用到了派生类当中的成员），所以我们不能把这些函数定义在基类中。这也就导致我们不能用 `sh->area()` 之类的写法——基类当中没有这个函数的定义。

那么我们在基类中添加一个 `area` 函数的定义，可否？答案是好像不行，因为这样一来，我们在使用 `sh->area()` 时也只能做到调用 `Shape::area()`，而根本就不是在调用 `Triangle::area()` 或者别的派生类成员函数。

而虚函数为我们提供了这种可能性——当我们用基类的指针/引用调用基类的虚函数时，程序会进行运行时检测，判断这个指针实际指向哪个类型的对象，然后调用这个类中的成员函数（如果存在的）。还是以 `Shape` 为例，我们可以在 `Shape` 当中声明虚函数 `perimeter` 和 `area`。这样，我们就可以用基类的指针调用 `area` 函数；而在运行时，程序会根据这个指针指向对象的类型，判断该用 `Triangle::area()`, `Circle::area()` 还是 `Parallelogram::area()`。

```
struct Shape { //这个类用来表示各种几何图形
    //...
    virtual double perimeter() const { return 0; }
    virtual double area() const { return 0; }
    //虚函数，如果通过指针/引用调用它，程序会进行运行时类型判断，决定调用哪个函数
```

```
virtual ~Shape() {}  
};
```

那么我们就可以用基类的虚函数作为媒介，通过基类指针调用派生类的成员函数。

```
Shape *ps[3] {  
    new Triangle{2,3,2},  
    new Circle{2},  
    new Parallelogram{1,5,90}  
};  
  
std::cout << ps[0]->area() << std::endl;  
//ps[0] 指向Triangle对象，所以调用Triangle::area()  
std::cout << ps[1]->area() << std::endl;  
//ps[1] 指向Circle对象，所以调用Circle::area()  
for (auto p : ps)  
    delete p;
```

瞧，连动态类型转换都可以省了。我们不再需要自行判断基类指针究竟指向什么，程序会帮我们判断。

## 虚函数的性质

在没有虚函数的情况下，编译器根据调用成员函数的对象/指针/引用就知道要调用哪个类的成员函数。举个例子，对于一个 `std::string` 类的对象 `str` 来说，当我们调用 `size()` 成员函数时，编译器根据对象的类型就可以推测出它要调用的是 `std::string::size()`，而不是 `std::vector<int>::size` 或者别的。

编译器知道了要调用哪个成员函数之后，它就会把这个函数与该语句绑定（或者说，链接）起来，这样程序就知道该调用什么函数了。

对于指针来说亦如此。如果没有虚函数的话，一切都很简单：对于 `Type*` 类型的指针，当我们使用 `->` 进行指针的成员访问时，编译器会把 `Type` 类型的成员与其绑定。这种绑定是在编译期完成的，我们把这种方式称为**早绑定 (Early binding)**。

虚函数为我们提供了另一种可能：用派生类的成员函数覆盖基类的成员函数。如果基类指针指向派生类对象，并且这个派生类有同名的成员函数，那么程序可以使用派生类成员覆盖基类成员。那么这两个条件——是否有同名函数，这是可以在编译时确定下来的；但是，基类指针究竟指向什么，这是不可能在编译时确定下来的<sup>5</sup>。正因如此，这个调用不能在编译期进行早绑定，而必须在运行期根据实际情况绑定到相应的成员函数中。我们把这种方式称为**迟绑定 (Late binding)**。用 `virtual` 声明的虚函数就拥有这种性质，它允许被派生类的同名成员函数覆盖。这样一来，当我们使用指针/引用来调用这个函数时，它就会被迟绑定。<sup>6</sup>

虚函数要求基类和派生类中的成员函数同名，所以构造函数不能被定义为虚函数。

然而，析构函数能被定义成虚函数。当我们用基类指针调用析构函数，或者是用 `delete` 回收基类指针指向的派生类内存空间时，派生类与基类的析构函数会被先后调用<sup>7</sup>。如果基类的析构函数不是虚函数，那么在回收内存空间时，程序将不会调用派生类的析构函数——这可能暗藏危险。所以我的建议是：只要你想写一个多态的基类，就请务必定义一个虚析构函数。

静态成员函数不能定义成虚函数。

<sup>5</sup>除非这个基类指针是 `constexpr` 之类的

<sup>6</sup>注意，如果我们使用基类的对象调用这个成员函数，那么它照样会被早绑定；只有用指针/引用才可以进行迟绑定。

<sup>7</sup>注意顺序。对于这种情况来说，派生类的析构函数先调用，然后才是基类的析构函数。这个顺序与派生类对象的析构很相似。

除构造函数、析构函数和静态成员函数之外的成员函数都可以用 **virtual** 关键字定义成虚函数。只要是虚函数，它就有了被派生类对象覆盖的可能<sup>8</sup>。前文定义的 `area` 成员函数便是如此。

虚函数描述的是单个成员函数的性质——某个成员函数是虚函数，并不意味着该类的其它成员函数都是虚函数。但是只要这个类有一个虚函数，它就是一个多态的类，其对象可以进行动态类型转换。

虚函数的性质可以顺着继承关系传递给派生类的同名函数。换句话说，只要 `Shape::area()` 是虚函数，那么 `Triangle::area()`, `Circle::area` 和 `Parallelogram::area()` 都是虚函数，而无论我们是否使用了 **virtual** 关键字。虽说如此，但是我依然建议读者在这些类中也加上 **virtual** 关键字，这样可以提高代码的可读性，避免我们在不知情的状况下犯一些不易查出的错误。

## 10.3 抽象基类与纯虚函数

我们在前面讲解继承时一致认为：派生类拥有基类的共性，同时又有自己的独特性。仅用那些共同成员来描述派生类对象是不够的，所以我们需要为派生类添加一些成员，从而更准确地描述派生类的对象。

但是实际编程中并不总是如此。有些时候，描述基类对象所需要的成员比描述派生类所需要的成员更多，以致我们不得想办法限制部分成员的值——

- 按理，实数应继承自复数（实数是复数的一部分）。但是复数需要至少两个成员（实部和虚部）来表示，而实数只需要一个（实部）。如果我们要写一个 `Complex` 类和一个 `Real` 类，那么必须在继承过程中把 `Complex` 类的虚部成员强制限制为 0——但这样只会引发无意义的内存空间浪费，我们的 `Real` 类总是存在着一个没有任何作用的成员。
- 按理，菱形应继承自平行四边形（各边长相等的平行四边形是菱形）。但是平行四边形需要至少三个成员（两边长及一角）来表示，而菱形只需要两个（一边长及一角）。如果我们要写一个 `Parallelogram` 类和一个 `Rhombus` 类，那么必须在继承过程中把 `Parallelogram` 的两边长成员强制限制为相同值——而且只要改变边长，就必须两个值一起修改。无论怎么说，这也太麻烦了点。

面对这样的问题，一种粗糙的处理方式是，把基类与派生类颠倒过来。既然实数只需要一个成员，复数需要两个成员，那么我们就让复数类继承实数类，这不就好了吗？

```
class Real {
    double _r; // 共同属于Real和Complex类的实部
};

class Complex : public Real {
    double _i; // 专属于Complex类的虚部
}
```

这样能解决代码上的困难，但是会造成逻辑上的困惑——复数怎么成了实数的一部分了？

为了更妥善地解决这个问题，我们需要使用**抽象基类**（Abstract base class, ABC）。

### 什么是抽象基类？

**抽象类**（Abstract class），简单说来就是不能定义对象的基类。我们曾通过把构造函数写在 **protected** 区或者 **private** 区的做法来防止在类（派生类）外定义对象，但这种做法仍不能避免我

<sup>8</sup>虚析构函数不是“被覆盖”了。它只是先调用了派生类的析构函数而已。

们在类内定义对象。抽象类则不同，我们不能在任何地方定义它的对象。与抽象类相对的叫作具体类（Concrete class），我们可以定义它们的对象。

抽象类生来就是用来当基类的，所以我们就叫它抽象基类。仍以实数、复数间的关系为例，原本它们是直接继承关系，但如果用直接继承的关系来描述，就会在代码层面引发诸多不便。为此，我们可以把实数和复数的共性（实部）提取出来，作为“抽象复数基类”的成员；然后让复数类和实数类都继承自它，如图 10.3 所示。

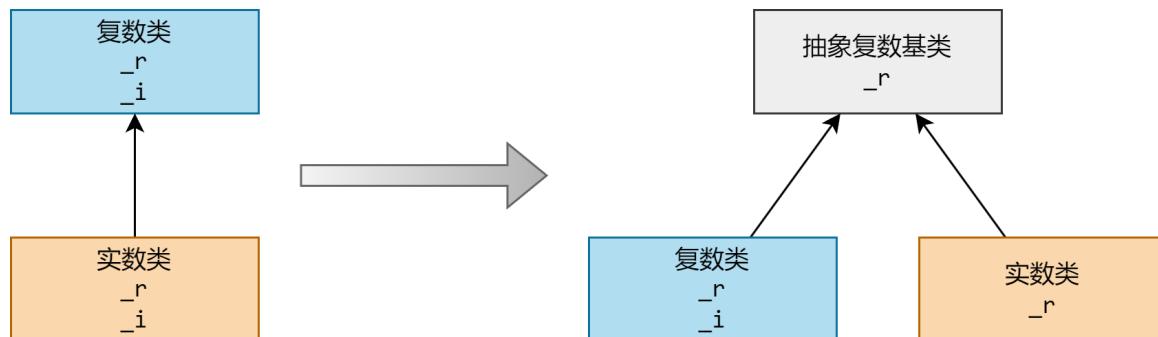


图 10.3: 用抽象基类表示 Complex 与 Real 的关系

这样，我们就在保证逻辑关系的条件下避免了让 Real 类多出一个 \_i 成员。

对于多级继承的情况来说也可以如此。图 10.4 就是一个多级继承的例子。

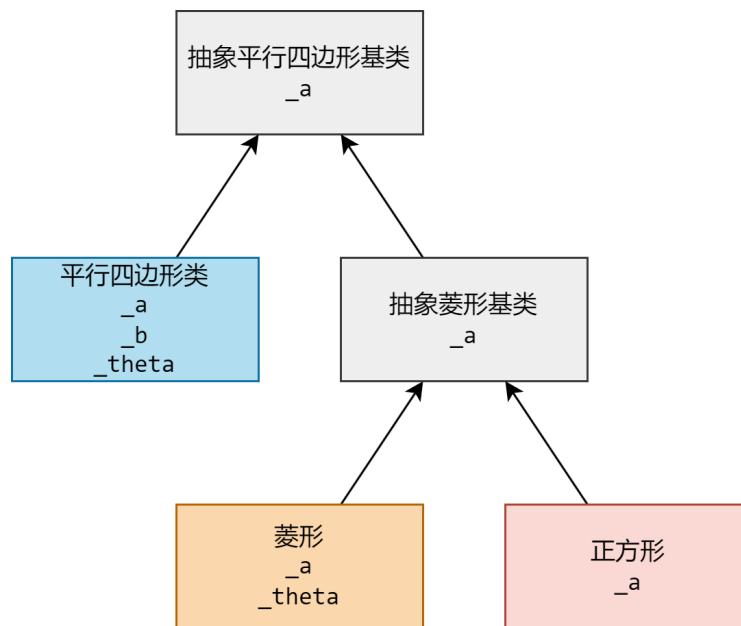


图 10.4: 用抽象基类表示平行四边形、菱形和正方形的关系

## 纯虚函数

含有纯虚函数（Pure virtual function）的基类就是抽象基类。

我们想，抽象基类没有对象，我们只能用抽象基类的指针/引用来指向其派生类。当调用基类的虚函数时，程序总是能找到对应派生类中的虚函数，并调用之。所以说抽象基类的虚函数没有什么实际作用，它的存在只是为了让编译器不报错——我们讲过，基类指针只能调用基类中声明过的函数；如果只在派生类中声明而未在基类中声明，我们是不能调用的。

既然抽象基类的虚函数不可能真的用到，那么我们当然不需要真的给它一个定义。C++ 中规定，如果给一个虚函数声明添加 =0 后缀<sup>9</sup>，那么这个函数就是纯虚函数，它只有名称意义上存在，而不需要实际意义上的定义。

```
struct Shape { //抽象基类，这个类用来表示各种几何图形
    //...
    virtual double perimeter()const = 0; //周长，纯虚函数
    virtual double area()const = 0; //周长，纯虚函数
};
```

抽象基类与纯虚函数之间是充要条件，即：只要这个基类中有一个纯虚函数，它就是抽象基类；只要这个基类中没有纯虚函数，它就不是抽象基类。

唯一的特殊情况还是析构函数。一个析构函数无论是不是纯虚函数，它都会被调用。所以我们定义成纯虚函数也没有什么意义，只是给自己找麻烦。因此对于析构函数，我的态度一般是：如果基类中有别的虚函数，那么析构函数最好也要是虚函数；但是析构函数不需要是纯虚函数，那没什么用。

## 实操：不同的几何图形

接下来我们就用目前为止学到的继承和虚函数方法表示不同几何图形之间的关系，并设计相应的成员函数来计算它们的周长和面积。如有必要，我们也使用抽象基类。

为了简便起见，我们只表示三角形、圆形、平行四边形、菱形和正方形。它们都直接或间接继承自 Shape 抽象基类，所以我们可以先把这个 Shape 写出来。

```
struct Shape { //抽象基类，这个类用来表示各种几何图形
    static constexpr double Pi {3.1415926}; //常量表达式Pi
    static constexpr double Deg2Rad(double deg) { //用于角度转弧度
        return deg * Pi / 180;
    }
    static constexpr double Rad2Deg(double rad) { //用于弧度转角度
        return rad * 180 / Pi;
    }
    virtual double perimeter()const = 0; //周长，纯虚函数
    virtual double area()const = 0; //周长，纯虚函数
    virtual ~Shape() {} //析构，虚函数
};
```

这里的 Shape 用 **struct** 来定义，这样可以少写一个 **public** 了。

接下来我们就用 Shape 派生其它类，如图 4.5 所示。

三角形和圆形的定义很简单，没什么可说的。

```
class Triangle : public Shape { //三角形
    double _a;
    double _b;
    double _c; //三角形的三条边
public:
    Triangle(double a, double b, double c)
        : _a {a}, _b {b}, _c {c} {} //构造函数
    virtual double perimeter()const { return _a + _b + _c; } //周长
```

---

<sup>9</sup>MSVC 还支持使用 **abstract** 关键字，不过它不属于 C++ 标准，也未必能在别的编译器中通过编译。

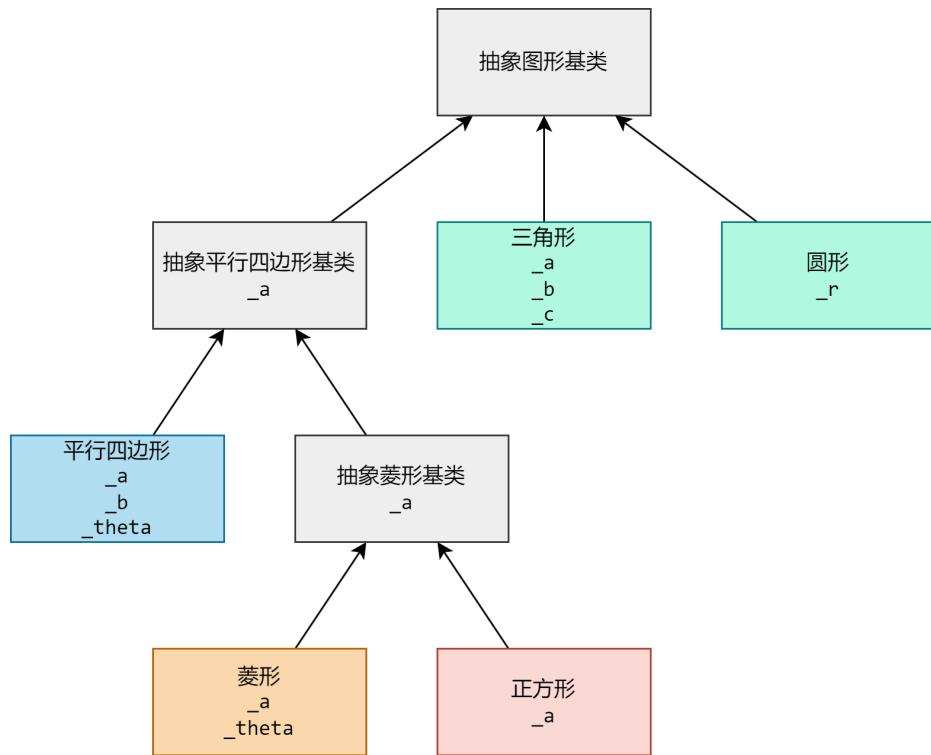


图 10.5: Shape 之下派生的各个类

```

virtual double area()const { //面积
    double s {(_a + _b + _c) / 2};
    return std::sqrt(s * (s - _a) * (s - _b) * (s - _c));
}
};

class Circle : public Shape { //圆形
    double _r; //圆的半径
public:
    Circle(double r) : _r {r} {}
    virtual double perimeter()const { return 2 * Pi * _r; } //周长
    virtual double area()const { return Pi * _r * _r; } //面积
};
```

至于平行四边形这边，就要更复杂一点。我们先把抽象平行四边形基类写出来吧。

```

class Parallelogram_abc : public Shape { // 抽象平行四边形基类
protected:
    double _a; //所有平行四边形都至少需要一条边长信息
public:
    Parallelogram_abc(double a) : _a {a} {} //构造函数
};
```

这里我们不需要再写纯虚函数 `perimeter` 和 `area` 了，因为纯虚函数 `Shape::perimeter()` 和 `Shape::area()` 也是 `Parallelogram_abc` 的成员函数。所以 `Parallelogram_abc` 也就自动地变成抽象基类。作为抽象基类，它只有一个属于平行四边形、菱形和正方形的共性，那就是一条边长 `_a`。

接下来我们定义平行四边形类，它在抽象平行四边形基类的基础上，增加了一条边长和一个角度。

```

class Parallelogram : public Parallelogram_abc { //平行四边形
    double _b;
    double _theta;
public:
    Parallelogram(double a, double b, double theta)
        : Parallelogram_abc {a}, _b {b}, _theta {theta} {}
    double perimeter()const { return 2 * (_a + _b); }
    double area()const { return _a * _b * std::sin(Deg2Rad(_theta)); }
};

```

这里的周长和面积仍然是虚函数，但是我们不用把 `virtual` 写出来。在 `Parallelogram` 的构造函数函数中，`_a` 的初始化要交给 `Parallelogram_abc` 来完成。这也就意味着，虽然我们不能定义抽象基类的对象，但是调用抽象基类的构造函数还是有可能的。

接下来是抽象菱形基类的定义。

```

struct Rhombus_abc : Parallelogram_abc { //抽象菱形基类
    Rhombus_abc(double a) : Parallelogram_abc {a} {} //构造函数
};

```

这里我们不需要添加新的成员对象，只需要写一个构造函数就够了。至于纯虚函数，它可以继承自 `Parallelogram_abc`，我们无须再写。

下面分别定义菱形和正方形类就可以了，很简单。

```

class Rhombus : public Rhombus_abc { //菱形
    double _theta;
public:
    Rhombus(double a, double theta) : Rhombus_abc {a}, _theta {theta} {}
    double perimeter()const { return 4 * _a; }
    double area()const { return _a * _a * std::sin(Deg2Rad(_theta)); }
};

struct Square : Rhombus_abc { //正方形
    Square(double a) : Rhombus_abc {a} {}
    double perimeter()const { return 4 * _a; }
    double area()const { return _a * _a; }
};

```

我们突然发现，`Rhombus::perimeter()` 和 `Square::perimeter()` 的定义是完全相同的——实际上菱形与正方形的周长公式真的完全一致。因此我们还可以把它们作为共性，定义到抽象菱形基类当中，也就是把上述代码改成这样：

```

struct Rhombus_abc : Parallelogram_abc { //抽象菱形基类
    double perimeter()const { return 4 * _a; }
    Rhombus_abc(double a) : Parallelogram_abc {a} {} //构造函数
};

class Rhombus : public Rhombus_abc { //菱形
    double _theta;
public:
    Rhombus(double a, double theta) : Rhombus_abc {a}, _theta {theta} {}
    double area()const { return _a * _a * std::sin(Deg2Rad(_theta)); }
};

```

```
struct Square : Rhombus_abc { // 正方形
    Square(double a) : Rhombus_abc {a} {}
    double area()const { return _a * _a; }
};
```

而当调用 Rhombus 或者 square 的 perimeter() 函数时，编译器自然能根据名称查找规则找到 Rhombus\_abc::perimeter() 来。总之，不必为此操心。

## 完整代码

以下是本例的完整代码。因为所有成员函数都定义在类内了，所以我只提供一个头文件内容。读者可以自行写源文件并验证其效果。

代码 10.1: Header.h

```
#pragma once
#include <cmath>

struct Shape { // 抽象基类，这个类用来表示各种几何图形
    static constexpr double Pi {3.1415926}; // 常量表达式 Pi
    static constexpr double Deg2Rad(double deg) { // 用于角度转弧度
        return deg * Pi / 180;
    }
    static constexpr double Rad2Deg(double rad) { // 用于弧度转角度
        return rad * 180 / Pi;
    }
    virtual double perimeter()const = 0; // 周长，纯虚函数
    virtual double area()const = 0; // 面积，纯虚函数
    virtual ~Shape() {} // 析构，虚函数
};

class Triangle : public Shape { // 三角形
    double _a;
    double _b;
    double _c; // 三角形的三条边
public:
    Triangle(double a, double b, double c)
        : _a {a}, _b {b}, _c {c} {} // 构造函数
    virtual double perimeter()const { return _a + _b + _c; } // 周长
    virtual double area()const { // 面积
        double s {(_a + _b + _c) / 2};
        return std::sqrt(s * (s - _a) * (s - _b) * (s - _c));
    }
};

class Circle : public Shape { // 圆形
    double _r; // 圆的半径
public:
    Circle(double r) : _r {r} {}
    virtual double perimeter()const { return 2 * Pi * _r; } // 周长
    virtual double area()const { return Pi * _r * _r; } // 面积
};
```

```
class Parallelogram_abc : public Shape { // 抽象平行四边形基类
protected:
    double _a; // 所有平行四边形都至少需要一条边长信息
public:
    Parallelogram_abc(double a) : _a {a} {} // 构造函数
};

class Parallelogram : public Parallelogram_abc { // 平行四边形
    double _b;
    double _theta;
public:
    Parallelogram(double a, double b, double theta)
        : Parallelogram_abc {a}, _b {b}, _theta {theta} {}
    double perimeter()const { return 2 * (_a + _b); }
    double area()const { return _a * _b * std::sin(Deg2Rad(_theta)); }
};

struct Rhombus_abc : Parallelogram_abc { // 抽象菱形基类
    double perimeter()const { return 4 * _a; }
    Rhombus_abc(double a) : Parallelogram_abc {a} {} // 构造函数
};

class Rhombus : public Rhombus_abc { // 菱形
    double _theta;
public:
    Rhombus(double a, double theta) : Rhombus_abc {a}, _theta {theta} {}
    double area()const { return _a * _a * std::sin(Deg2Rad(_theta)); }
};

struct Square : Rhombus_abc { // 正方形
    Square(double a) : Rhombus_abc {a} {}
    double area()const { return _a * _a; }
};
```

## 精讲篇



# 附录 A C++ 运算符基本属性

表 A.1 列出了截至 C++17 的所有运算符的优先级、结合性和重载要求。<sup>1</sup>

表 A.1: 截至 C++17 的所有运算符

优先级	运算符	运算符含义	结合性	重载要求
1	scope::a	作用域解析	从左到右	不可重载
2	var++ var--	后缀自增/自减		特殊参数要求 <sup>2</sup>
	type() type{}	函数风格类型转换 <sup>3</sup>		可重载
	fun()	函数调用		只能是成员函数
	arr[index]	下标运算 <sup>4</sup>		只能是成员函数
	obj.member	成员访问		不可重载
	p_obj->member	指针的成员访问		返回类型受限 <sup>5</sup> 只能是成员函数 不能是静态函数
3	++var --var	前缀自增/自减	从右到左	可重载
	+var -var	一元加/减(正/负号)		可重载
	!var ~var	逻辑非/按位取反		可重载
	(type)var	C 风格类型转换		可重载
	*pointer	取内容(解引用)		可重载
	&var	取地址		可重载
	new new[]	动态内存分配		可重载
	delete delete[]	动态内存回收		可重载
	sizeof a <sup>6</sup>	求内存空间大小		不可重载
4	obj.*p	成员指针访问	从左到右	不可重载
	p_obj->*p	指针的成员指针访问		可重载
5	lhs*rhs lhs/rhs lhs%rhs	乘法/除法/模运算		可重载 <sup>7</sup>
6	lhs+rhs lhs-rhs	加法/减法		可重载

<sup>1</sup>值得注意的是，虽然双引号"" 不被视为运算符，但它可以用在重载自定义字面量的语法中。本书不予以介绍，对此感兴趣的读者可以阅读 [用户自定义字面量-cppreference](#)。

<sup>2</sup>重载后缀自增/自减运算符需带 `int` 型参数。参数可以不具名，事实上也无需使用。这样的语法是为了与前缀形式相区别。

<sup>3</sup>函数风格类型转换、C 风格类型转换和 `static_cast` 在静态类型转换时是等价的。换言之，只需定义一个转换函数，就可以使用这三种类型转换语法。

<sup>4</sup>对于内置类型数组 `arr` 的下标运算，`arr[index]` (和 `index[arr]`) 都会解析为 `*(arr+index)` (和 `*(index+arr)`)，而自定义类型的下标重载版本不会如此。

<sup>5</sup>返回类型必须是裸指针或对象。返回对象时，必须返回同样重载了运算符 `->` 的对象。

<sup>6</sup>不使用括号的 `sizeof a` 只适用于数据或对象；如果要对类型求内存空间大小，必须用 `sizeof(type)` 形式。

<sup>7</sup>注意，一些重载运算符对参数有特殊要求：必须接收至少一个自定义类型的参数，可以是类或枚举类。比如说，`double operator%(double, double)` 就是不允许的。并非所有重载都有这个限制，比如下标运算可以接收 `int` 类型的参数。

优先级	运算符	运算符含义	结合性	重载要求	
7	<code>lhs&lt;&lt;rhs</code> <code>lhs&gt;&gt;rhs</code>	左移位/右移位	从左到右	可重载	
8	<code>lhs&lt;rhs</code> <code>lhs&lt;=rhs</code>	小于/小于或等于		可重载	
	<code>lhs&gt;rhs</code> <code>lhs&gt;=rhs</code>	大于/大于或等于		可重载	
9	<code>lhs==rhs</code> <code>lhs!=rhs</code>	相等运算符/不等运算符		可重载	
10	<code>lhs&amp;rhs</code>	按位与		可重载	
11	<code>lhs^rhs</code>	按位异或(互斥或)		可重载	
12	<code>lhs rhs</code>	按位或(可兼或)		可重载	
13	<code>lhs&amp;&amp;rhs</code>	逻辑与		可重载 <sup>8</sup>	
14	<code>lhs  rhs</code>	逻辑或(可兼或)		可重载	
15	<code>condition?exp1:exp2</code>	条件运算符	从右到左	不可重载	
	<code>throw exception</code>	抛出异常		不可重载	
	<code>lhs=rhs</code>	直接赋值		只能是成员函数	
	<code>lhs+=rhs</code> <code>lhs-=rhs</code>	复合赋值		只能是成员函数	
	<code>lhs*=rhs</code> <code>lhs/=rhs</code>				
	<code>lhs%=rhs</code> <code>lhs =rhs</code>				
16	<code>exp1,exp2</code>	逗号	从左到右	可重载	

另有这些运算符：`static_cast`, `dynamic_cast`, `const_cast`, `reinterpret_cast`, `typeid`, `noexcept`。它们在使用时必须对操作数加括号(相比之下, `sizeof` 是可以不对操作数加括号的), 因而不受优先级、结合性问题困扰, 不列入表 A.1。以上这些运算符均不可重载, 但你可以通过自定义转换函数的方式来使 `static_cast` 变得可用。

<sup>8</sup>逻辑与/逻辑或函数有短路求值特点, 如对于 `false&&f()`, 在左操作数为 `false` 时已经能确定整个表达式为 `false`, 故不需要再求 `f()`, 于是右操作数及其副作用都不会发生。但这两个运算符经过重载后并不保留短路求值特性, 左右操作数都会被计算。

## 附录 B ASCII 码表 (0 到 127)

ASCII 字符分为控制字符和可显示字符两部分。

表 B.1 是 33 个 ASCII 控制字符，其中 DEC 代表十进制形式，HEX 代表十六进制形式，CN 代表脱出字符<sup>1</sup>，ES 代表转义字符<sup>2</sup>，CH 代表字符含义。

表 B.1: 33 个 ASCII 控制字符

DEC	HEX	CN	ES	CH	DEC	HEX	CN	ES	CH
0	00	<sup>^</sup> @	\@	NUL 空	16	10	<sup>^</sup> P		DLE 退出数据链
1	01	<sup>^</sup> A		SOH 标题开始	17	11	<sup>^</sup> Q		DC1 设备控制 1
2	02	<sup>^</sup> B		STX 正文开始	18	12	<sup>^</sup> R		DC2 设备控制 2
3	03	<sup>^</sup> C		ETX 正文结束	19	13	<sup>^</sup> S		DC3 设备控制 3
4	04	<sup>^</sup> D		EOT 传送结束	20	14	<sup>^</sup> T		DC4 设备控制 4
5	05	<sup>^</sup> E		ENQ 询问	21	15	<sup>^</sup> U		MAK 反确认
6	06	<sup>^</sup> F		ACK 确认	22	16	<sup>^</sup> V		SYN 同步空闲
7	07	<sup>^</sup> G	\a	BEL 响铃	23	17	<sup>^</sup> W		ETB 传输块结束
8	08	<sup>^</sup> H	\b	BS 退格	24	18	<sup>^</sup> X		CAN 取消
9	09	<sup>^</sup> I	\t	HT 横向制表	25	19	<sup>^</sup> Y		EM 媒介结束
10	0A	<sup>^</sup> J	\n	LF 换行	26	1A	<sup>^</sup> Z		SUB 替换
11	0B	<sup>^</sup> K	\v	VT 纵向制表	27	1B	<sup>^</sup> [	\e <sup>3</sup>	ESC 退出
12	0C	<sup>^</sup> L	\f	FF 换页	28	1C	<sup>^</sup> \		FS 文件分隔符
13	0D	<sup>^</sup> M	\r	CR 回车	29	1D	<sup>^</sup> ]		GS 组分隔符
14	0E	<sup>^</sup> N		SO 移出	30	1E	<sup>^</sup> ^		RS 记录分隔符
15	0F	<sup>^</sup> O		SI 移入	31	1F	<sup>^</sup> _		US 单无分隔符
127	7F	<sup>^</sup> ?		DEL 删除					

表 B.2 是 95 个 ASCII 可打印字符，其中 DEC 表示十进制形式，HEX 表示十六进制形式，CH 代表字符。

<sup>1</sup> 脱出字符 (Caret notation)，是 ASCII 控制字符的一种记号，常用于在终端当中输入控制字符。多数情况下，脱出字符由 Ctrl+ 某键输入，而不是敲下^加一个字符。在显示时，脱出字符被视为单个字符，而非两个。

<sup>2</sup> 转义字符 (Escape sequence)，是一个字符序列，它以反斜线开头，用特定的字母组合来表达一些特殊含义。在输出转义字符时，它们不会按照原形式输出，而是被译为另一类字符（比如控制字符）。

<sup>3</sup> C++ 和许多语言标准都没有明确规定将\e作为转义字符，但是有些编译器，如 GCC，允许使用这种语法。

表 B.2: 95 个 ASCII 可打印字符

DEC	HEX	CH	DEC	HEX	CH	DEC	HEX	CH	DEC	HEX	CH
32	20	<sup>4</sup>	48	30	Ø	64	40	@	80	50	P
33	21	!	49	31	1	65	41	A	81	51	Q
34	22	" <sup>5</sup>	50	32	2	66	42	B	82	52	R
35	23	#	51	33	3	67	43	C	83	53	S
36	24	\$	52	34	4	68	44	D	84	54	T
37	25	%	53	35	5	69	45	E	85	55	U
38	26	&	54	36	6	70	46	F	86	56	V
39	27	' <sup>6</sup>	55	37	7	71	47	G	87	57	W
40	28	(	56	38	8	72	48	H	88	58	X
41	29	)	57	39	9	73	49	I	89	59	Y
42	2A	*	58	3A	:	74	4A	J	90	5A	Z
43	2B	+	59	3B	;	75	4B	K	91	5B	[
44	2C	,	60	3C	<	76	4C	L	92	5C	\ <sup>7</sup>
45	2D	-	61	3D	=	77	4D	M	93	5D	]
46	2E	.	62	4E	>	78	4E	N	94	5E	^
47	2F	/	63	4F	?	79	4F	O	95	5F	_
96	60	`	104	68	h	112	70	p	120	78	x
97	61	a	105	69	i	113	71	q	121	79	y
98	62	b	106	6A	j	114	72	r	122	7A	z
99	63	c	107	6B	k	115	73	s	123	7B	{
100	64	d	108	6C	l	116	74	t	124	7C	
101	65	e	109	6D	m	117	75	u	125	7D	}
102	66	f	110	6E	n	118	76	v	126	7E	~
103	67	g	111	6F	o	119	77	w			

<sup>4</sup>此处为空格符。<sup>5</sup>字符 " 在 C++ 代码中需用转义字符 '\\"' 表达。<sup>6</sup>字符 ' 在 C++ 代码中需用转义字符 '\'' 表达。<sup>7</sup>字符 \ 在 C++ 代码中需用转义字符 '\\\' 表达。

# 附录 C 相关数学知识

本附录列出了部分有关的数学知识。这些知识并非必要的内容，但掌握它们可以让你对 C++ 语言有更深入的理解。

## C.1 数制转换

### 简介与概念引入

图 C.1 是一个数字时钟的界面，它指示的日期是 2016 年 08 月 29 日，时间是 20:34:42。



图 C.1: 一个简单的数字时钟  
图片来源: 维基共享资源

每一秒，这个数字时钟显示的时间都会发生变化，变化主要体现在“秒”这一位上。我们很容易就能想象出来，下一步它会变成 43，再下一秒它会变成 44，就这样每一秒都增加 1，直到 59——这是一个例外，下一步它不是变成 60，而是变成了 00。同时发生变化的还有“分”这一位，它会变成 35。

如果“分”位不发生变化，我们会在 20:34:59 的下一秒看到 20:34:00。这会给我们造成一种错觉，好像我们回到了 60 秒之前。但是“分”位发生了变化，这就很明确地告诉我们，现在是 20:35:00，刚才是 20:34:00，这俩不一样。

此后的每一分钟都是如此。每当“秒”走到 59 的时候，下一步它就会变成 00，同时“分”增加 1，直到“分”也遇到了 59，这时它再加就有困难了，因为“分”这一位没有 60，所以它只能变回 00，同时让“时”增加 1。这就是一个进位（Carry）过程。

为什么要进位？这是因为，只用一位数字能表示的范围太小，一旦我们进行计数，就很容易出现不够的情况。比如说时钟，如果它只“秒”位，那么它就会每 60 秒一循环，从 00 加到 59 又变成了 00<sup>1</sup>，这怎么合理呢？<sup>2</sup>但是加上了“分”这一位之后，我们只需要在每次“秒”超过 59 不能再加的时

<sup>1</sup>这是一种溢出（Overflow）现象。

<sup>2</sup>一个现实的例子就是干支纪年法。天干地支轮相更替，60 年作一循环，但它没有进位机制，只有一个干支位，所以我们不能单纯根据一个“丙申”判断它是 1176 年还是 2016 年还是别的年份，必须依赖其它信息作为参考。

候，把“秒”变回 00，并且在“分”位上进位，就可以正确记录它的变化了。所以我们可以说明，秒是“逢 60 进一”的。

仅有“分”位还是不够，它也有 60 分钟一循环的问题，所以我们还有“时”。每当分超过了 59 不能再加的时候，就可以把“分”变回 00，并且在“时”位上进位，这样又可以正确记录它的变化了。所以说，分也是“逢 60 进一”的。

同样的道理，我们可以看出，时是“逢 24 进一”的。至于“日”，它的进位规则就比较麻烦了，本书就不再细究。

我们在日常生活中最常用的是 10 进制。对于“个位”来说，它只能表示从 0 到 9 的 10 种情况。在计数的时候，一旦超过了 9，我们就需要进位：让“十位”加 1<sup>3</sup>，同时个位变回 0。所以说，个位是“逢 10 进一”的。

两位数只能表示 100 种情况，我们依然可能遇到不够的情况。当我们加到超过 99 的时候，就需要再进位了。这个进位过程可以分为两步：

1. 个位进位。进位之后个位变为 0，而十位加 1，变为“9+1”。我们发现，这个值超过了十位能表示的范围，所以十位也需要进位。
2. 十位进位。进位之后十位变为 0，而百位加 1，变为 1。至此进位完成。

当我们熟悉了整个过程之后，我们就根本不用分成两步来进位了，直接下意识得到  $99+1=100$  就行。总之，十位也是“逢 10 进一”的。

一个十进制整数的每一位都满足这样“逢 10 进一”的规则，所以我们可以仿照此理来进行推导。于是我们把计数推广到加法（加法就是重复若干次的计数过程），再把加法推广到乘法（若干次同样的加法过程），再把乘法推广到乘方（若干次同样的乘法过程），乃至高德纳箭头<sup>4</sup>。图 C.2 展示了这样一个过程，我们就是从这里开始学习数学的。

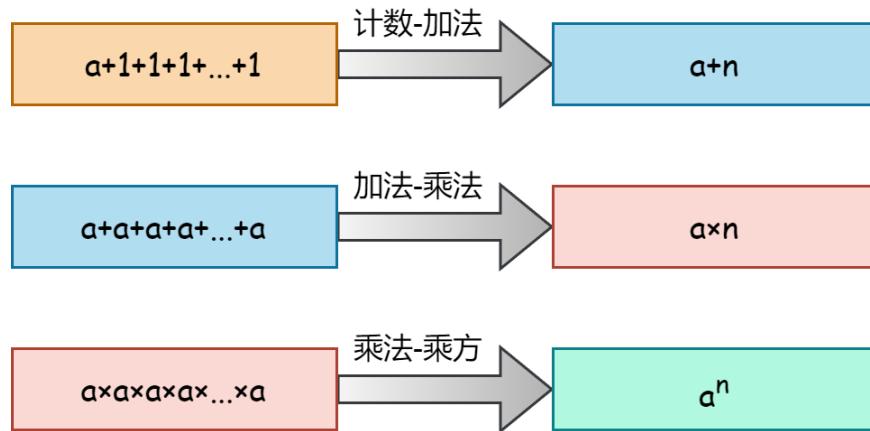


图 C.2: 从计数到乘方

我们在小学时都背过加法表和乘法表，它就是 10 进制整数的计算规则。但是如果换一套进制呢，情况就会发生变化了。图 C.3 就是一个 12 进制乘法表，仅供读者开拓眼界。

## 数的形式化表示

有这样一个十进制的三位数，它的百位是  $a$ ，十位是  $b$ ，个位是  $c$ ，怎么把这个数表示出来呢？我们可以用  $100a + 10b + c$  的形式来表示。

<sup>3</sup>原来的十位是 0，我们省略不写。

<sup>4</sup>高德纳箭头 (Knuth's up-arrow notation)，由高德纳在 1976 年提出，它是一种表示巨大整数的方法，可以视作是对重复乘方的迭代扩展。

	2	3	4	5	6	7	8	9	A	B	10
2	4	6	8	A	10	12	14	16	18	1A	20
3	6	9	10	13	16	19	20	23	26	29	30
4	8	10	14	18	20	24	28	30	34	38	40
5	A	13	18	21	26	2B	34	39	42	47	50
6	10	16	20	26	30	36	40	46	50	56	60
7	12	19	24	2B	36	41	48	53	5A	65	70
8	14	20	28	34	40	48	54	60	68	74	80
9	16	23	30	39	46	53	60	69	76	83	90
A	18	26	34	42	50	5A	68	76	84	92	A0
B	1A	29	38	47	56	65	74	83	92	A1	B0
10	20	30	40	50	60	70	80	90	A0	B0	100

图 C.3: 一个 12 进制乘法表

图片来源: 维基共享资源

同样的道理, 如果我们知道了某个整数的每一位, 那我们就可以用每一位乘  $10^n$  再求和的形式把它表示出来。

$$36 = 3 \times 10^1 + 6 \times 10^0$$

$$1024 = 1 \times 10^3 + 0 \times 10^2 + 2 \times 10^1 + 4 \times 10^0$$

$$32768 = 3 \times 10^4 + 2 \times 10^3 + 7 \times 10^2 + 6 \times 10^1 + 8 \times 10^0$$

所以我们可以把它抽象一下: 对于任意一个整数  $a$ , 它可以用它的每一位  $a_n$  乘以  $10^n$  求和表示成

$$a = a_0 \times 10^0 + a_1 \times 10^1 + a_2 \times 10^2 + \dots + a_n \times 10^n + \dots$$

其中  $a_0$  是个位,  $a_1$  是十位,  $a_2$  是百位, 以此类推。我们还可以用求和符号简写成这样:

$$a = \sum_{n=0}^{+\infty} a_n 10^n$$

如果要表示一个小数呢? 我们就需要把  $n$  扩展到负数范围中来。

$$3.14159 = 3 \times 10^0 + 1 \times 10^{-1} + 4 \times 10^{-2} + 1 \times 10^{-3} + 5 \times 10^{-4} + 9 \times 10^{-5}$$

$$0.5772 = 0 \times 10^0 + 5 \times 10^{-1} + 7 \times 10^{-2} + 7 \times 10^{-3} + 2 \times 10^{-4}$$

所以我们还可以把这个数写成这样:

$$a = \dots + a_2 \times 10^2 + a_1 \times 10^1 + a_0 \times 10^0 + a_{-1} \times 10^{-1} + a_{-2} \times 10^{-2} + \dots$$

其中的  $a_{-1}$  是十分位,  $a_{-2}$  是百分位,  $a_{-3}$  是千分位, 以此类推。我们还可以用求和符号简写成这样:

$$a = \sum_{n=-\infty}^{+\infty} a_n 10^n$$

以上全都是基于十进制来讨论的。如果换一个进制呢？我们只需要把 10 改换成对应的数就可以了。这是二进制：

$$a = \sum_{n=-\infty}^{+\infty} a_n 2^n$$

这是八进制：

$$a = \sum_{n=-\infty}^{+\infty} a_n 8^n$$

这是十六进制<sup>5</sup>：

$$a = \sum_{n=-\infty}^{+\infty} a_n 16^n$$

假如我们有一个十进制<sup>6</sup>的  $a = (1024)_{10}$  和一个十六进制的  $b = (400)_{16}$ ，如何判断它们是否相等呢？很简单，我们只需要验证

$$\sum_{n=-\infty}^{+\infty} a_n 10^n = \sum_{m=-\infty}^{+\infty} b_m 16^m$$

就可以了。等号左边是  $1 \times 10^3 + 2 \times 10^1 + 4 \times 10^0 = 1024$ ，<sup>7</sup>而右边是  $4 \times 16^2 = 1024$ ，所以等号成立， $(1024)_{10} = (400)_{16}$  就是相等的。

而  $(153)_8$  和  $(211)_7$  不相等，这是因为

$$(153)_8 = 1 \times 8^2 + 5 \times 8^1 + 3 \times 8^0 = 107$$

$$(211)_7 = 2 \times 7^2 + 1 \times 7^1 + 1 \times 7^0 = 106$$

这里的 107 和 106 不相等。

所以说  $(a)_R = \sum_{n=-\infty}^{+\infty} a_n R^n$  就是我们对  $(a)_R$  这个数的形式化表示。只要知道这个数的每一位，我们就能确定这个数的值。

不过这么写还是有一点麻烦，所以我们还可以自行搞一套符号来简化  $\sum_{n=-\infty}^{+\infty} a_n R^n$ ，我们可以写成这样：

$$(a)_R = \sum_{n=-\infty}^{+\infty} a_n R^n = [\dots, a_2, a_1, a_0; a_{-1}, a_{-2}, \dots]_R$$

其中的分号用来分隔个位与十分位。举几个例子：

$$(10)_{10} = [1, 0;]_{10}$$

$$(1010)_2 = [1, 0, 1, 0;]_2$$

$$(0.75)_{10} = [0; 7, 5]_{10}$$

$$(0.c)_{16} = [0; 12]_{16}$$

---

<sup>5</sup>在十六进制中，用一位阿拉伯数字不足以表示一位数，所以引入  $a$ （也可大写）表示 10， $b$  表示 11， $c$  表示 12， $d$  表示 13， $e$  表示 14， $f$  表示 15。

<sup>6</sup>为了把不同进制下的数字区分开，我们在数字之外套上括号并用下标注明是何种进制。

<sup>7</sup>对于十进制数，可以不用这么麻烦，直接 1024 就行。

$$(1a.c3)_{16} = [1, 10; 12, 3]_{16}$$

这里的  $(10)_{10}$  与  $(1010)_2$  是相等的,  $(0.75)_{10}$  与  $(0.c)_{16}$  也是相等的, 读者可以自行计算验证。<sup>8</sup> 而  $(1a.c3)_{16}$  的表示可以参考图 C.4 来理解。

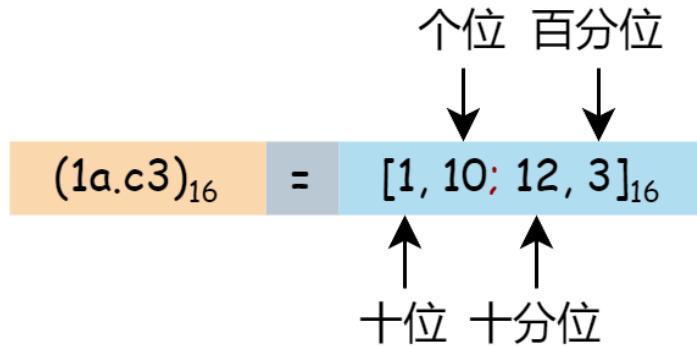


图 C.4:  $(1a.c3)_{16}$  的形式化表示

## 进位与借位

现在让我们用形式化的表示方法来计算一下  $(c)_{16} + (f)_{16}$  的值。

$$\begin{aligned}(c)_{16} &= [12;]_{16} \\ (f)_{16} &= [15;]_{16} \\ [12;]_{16} + [15;]_{16} &= [27;]_{16}\end{aligned}$$

现在我们得到了一个数, 它的个位是 27, 这已经超过十六进制的最大限制 15 了, 那么我们怎么办? 当然是进位了。

$$[27;]_{16} = [1, 27 - 16;]_{16} = [1, 11;]_{16} = (1b)_{16}$$

我们怎么判断这个结果是不是正确呢? 很简单, 用十进制的计算结果  $(27)_{10}$  和  $(ab)_{16}$  比较一下就好了, 也就是验证

$$\sum_{n=-\infty}^{+\infty} a_n 10^n = \sum_{n=-\infty}^{+\infty} b_n 16^n$$

最后我们会发现  $1 \times 16^1 + 11 \times 16^0 = 27$  确实成立, 所以计算结果无误。

这里的进位过程非常简单, 就是“后一位减 16, 同时前一位加 1”。如果换到十进制下, 那就是“后一位减 10, 同时前一位加 1”。

$$[9, 9, 10;]_{10} = [9, 10, 0;]_{10} = [10, 0, 0;]_{10} = [1, 0, 0, 0;]_{10}$$

所以说同样是  $(1000)_{10}$  这个数字, 我们写成  $[1, 0, 0, 0;]_{10}$  和  $[9, 9, 10;]_{10}$ , 甚至写成  $[9, 9, 9; 9, 9, 10]_{10}$  都是对的。正确的写法有无数种, 但最简单的写法还是进位之后的写法。

而在计算过程当中, 我们没那么多限制, 可以任意选择自己喜欢的写法。比如说, 我们做减法运算时, 就可能涉及到借位 (Borrow) 操作。这就是在不同形式之间进行变化, 从而简化计算的方法。

<sup>8</sup> 我们把 [ ] 内的数字都用十进制来表示, 这样会方便我们计算。毕竟我们还是最熟悉十进制。

举个例子，我们要计算  $(12.6)_8 - (7.7)_8$ ，这时我们就可以通过把  $[1, 0; 6]_8$  转换成我们方便计算的形式。注意它是八进制的，所以借位的规则是“后一位加 8，同时前一位减 1”。

$$\begin{aligned}[1, 2; 6]_8 - [7; 7]_8 \\= [10; 6]_8 - [7; 7]_8 \\= [9; 14]_8 - [7; 7]_8 \\= [2; 7]_8\end{aligned}$$

所以  $(12.6)_8 - (7.7)_8 = (2.7)_8$ 。

以上是进 1 的情况，实际计算时我们可能进 2，进 3 或者进 4，例如  $[56, 111; 54]_{10}$ 。这种情况下我们最好用带余除法来进位。先看十分位：

$$54 \div 10 = 5 \dots 4$$

所以我们进 5，也就是十分位减  $5 \times 10 = 50$ ，同时个位加 5，变成  $[56, 116; 4]_{10}$ 。再看个位：

$$116 \div 10 = 11 \dots 6$$

所以我们进 11，也就是个位减  $11 \times 10 = 110$ ，同时十位加 11，变成  $[67, 6; 4]_{10}$ 。再看十位：

$$67 \div 10 = 6 \dots 7$$

所以我们进 6，也就是十位减  $6 \times 10 = 60$ ，同时百位加 6，变成  $[6, 7, 6; 4]_{10}$ ，这样整理完了就会得到  $(676.4)_{10}$ 。

换一个  $R$  进制，思路也不变，我们只需要把除数从 10 改成  $R$  就行了。所以  $[0; 500]_{16}$  就可以这样处理：

$$\begin{aligned}[0; 500]_{16} \\500 \div 16 = 31 \dots 4 \Rightarrow [31; 4]_{16} \\31 \div 16 = 1 \dots 15 \Rightarrow [1, 15; 4]_{16}\end{aligned}$$

最后把  $[1, 15; 4]_{16}$  写成十六进制数的形式  $(1f.4)_{16}$  就可以了。

## 数制转换的一般方法

接下来我们考虑如何进行数制转换。假如我们有一个  $R$  进制的数  $a$ ，希望转换成  $T$  进制的数  $x$ ，那么只需要找到一个  $\dots, x_2, x_1, x_0; x_{-1}, x_{-2}, \dots$ ，能满足式子

$$\sum_{n=-\infty}^{+\infty} a_n R^n = \sum_{m=-\infty}^{+\infty} x_n T^n$$

就可以了。

在这里，我们把  $a$  拆成整数部分和小数部分来考虑。对于整数部分，我们用到的思路和进位相似；而对于小数部分，我们用到的思路和借位相似。在分别处理完整数和小数的转换之后，我们只需把结果相加即可。

### 整数部分

如果某个  $R$  进制整数要转换成十进制，那就非常简单，直接用  $\sum_{n=0}^{+\infty} a_n R^n$  算出来就行。以下是例子：

$$\begin{aligned}(4d9)_{16} &= 4 \times 16^2 + 13 \times 16^1 + 9 \times 16^0 = (1241)_{10} \\(1001011)_2 &= 1 \times 2^6 + 1 \times 2^3 + 1 \times 2^1 + 1 \times 2^0 = (75)_{10} \\(64)_8 &= 6 \times 8^1 + 4 \times 8^0 = (52)_{10}\end{aligned}$$

如果要把某个十进制整数转换成  $R$  进制，我们就可以先把它放在  $x$  的个位上，然后用  $R$  进制进位的方式来处理它。例如，要把  $(382)_{10}$  转换成十六进制，步骤如下：

$$\begin{aligned}[382;]_{16} \\382 \div 16 &= 23 \dots 14 \Rightarrow [23, 14;]_{16} \\23 \div 16 &= 1 \dots 7 \Rightarrow [1, 7, 14;]_{16} \\1 \div 16 &= 0 \dots 1 \Rightarrow [1, 7, 14;]_{16}\end{aligned}$$

最后把  $[1, 7, 14;]_{16}$  表达成十六进制的形式  $(17e)_{16}$  即可。

要把  $(395)_{10}$  转换成八进制，步骤如下：

$$\begin{aligned}[395;]_8 \\395 \div 8 &= 49 \dots 3 \Rightarrow [49, 3;]_8 \\49 \div 8 &= 6 \dots 1 \Rightarrow [6, 1, 3;]_8 \\6 \div 8 &= 0 \dots 6 \Rightarrow [6, 1, 3;]_8\end{aligned}$$

最后把  $[6, 1, 3;]_8$  表达成八进制的形式  $(613)_8$  即可。

要把  $(41)_{10}$  转换成二进制，步骤如下：

$$\begin{aligned}[41;]_2 \\41 \div 2 &= 20 \dots 1 \Rightarrow [20, 1;]_2 \\20 \div 2 &= 10 \dots 0 \Rightarrow [10, 0, 1;]_2 \\10 \div 2 &= 5 \dots 0 \Rightarrow [5, 0, 0, 1;]_2 \\5 \div 2 &= 2 \dots 1 \Rightarrow [2, 1, 0, 0, 1;]_2 \\2 \div 2 &= 1 \dots 0 \Rightarrow [1, 0, 1, 0, 0, 1;]_2 \\1 \div 2 &= 0 \dots 1 \Rightarrow [1, 0, 1, 0, 0, 1;]_2\end{aligned}$$

最后把  $[1, 0, 1, 0, 0, 1;]_2$  表达成二进制的形式  $(101001)_2$  即可。

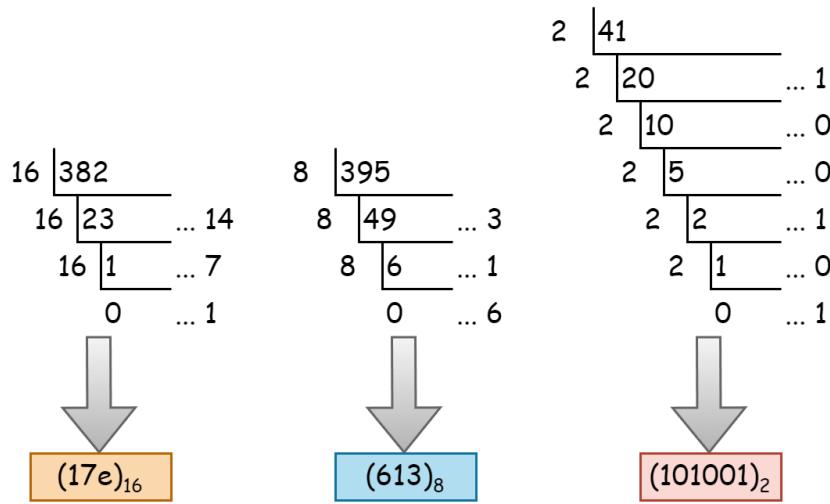
我们不难发现，整个过程的关键其实就是反复对着上一步的商值除以  $R$ ，然后把每一步的余数按照个位、十位、百位的顺序排起来。实际操作中，我们更倾向于用短除法来进行转换，如图 C.5 所示。

使用短除法时只需注意，先算出来的余数在个位，后算出来的在十位，以此类推，从低到高。

当我们需要把  $R$  进制数转换成  $T$  进制时，又该如何做呢？一般我们使用十进制作为中介<sup>9</sup>，先把  $R$  进制转换为十进制，再把它转换为  $T$  进制。这样的好处是，我们所有的过程计算都可以基于十

---

<sup>9</sup>但当我们进行二进制、八进制和十六进制之间的互相转换时，有更简单的方法。详见后文。

图 C.5: 通过短除法把十进制数转换成  $R$  进制

进制，尤其是乘和除。如果不使用十进制作为中介，那么我们在转换时就需要背  $R$  进制的乘法表了。

### 小数部分

接下来我们考虑小数部分。

如果某个  $R$  进制小数要转换成十进制，那就非常简单，直接用  $\sum_{n=-\infty}^{-1} a_n R^n$  算出来就行。以下是例子：

$$(0.1011)_2 = 1 \times 2^{-1} + 1 \times 2^{-3} + 1 \times 2^{-4} = 0.6875$$

$$(0.c8)_{16} = 12 \times 16^{-1} + 8 \times 16^{-2} = 0.78125$$

$$(0.5)_8 = 5 \times 8^{-1} = 0.625$$

如果要把某个十进制整数转换成  $R$  进制，我们就需要用借位的方法来处理它了。我们还是先把它放在个位上，然后用  $R$  进制借位的方式来处理它。例如，要把 0.78125 转换成十六进制，我们就先写成  $[0.78125;]_{16}$ 。

怎么借位呢？试想，如果我们要从前一位借 1，那么就要向后一位加 16；如果我们要从前一位借 2，那么就要向后一位加 32，……。那么我们现在想从前一位借 0.78125 呢？自然是把它乘上 16 加到后一位中了。

$$0.78125 \times 16 = 12.5 \Rightarrow [0; 12.5]_{16}$$

接下来我们发现十分位上还有小数，那么我们就借 0.5，把它乘上 16 加到后一位去——只有这样才能把十分位变成整数。

$$0.5 \times 16 = 8 \Rightarrow [0; 12, 8]_{16}$$

这样就完成了。最后只需要把它整理成  $(0.c8)_{16}$  即可。

再举一个例子，把  $(0.669)_{10}$  转换成二进制的操作是

$$[0.669;]_2$$

$$0.669 \times 2 = 1.338 \Rightarrow [0; 1.338]_2$$

$$\begin{aligned}
 0.338 \times 2 &= 0.676 \Rightarrow [0; 1, 0.676]_2 \\
 0.676 \times 2 &= 1.352 \Rightarrow [0; 1, 0, 1.352]_2 \\
 0.352 \times 2 &= 0.704 \Rightarrow [0; 1, 0, 1, 0.704]_2 \\
 &\dots \dots
 \end{aligned}$$

小数的进制转换是有可能算不尽的，所以我们根据实际的精度需要来近似就行。这个数的二进制结果在  $(0.1010)_2$  到  $(0.1011)_2$  之间，我们可以取  $(0.101)_2$  作为近似。

再举一例，把  $(0.35)_{10}$  转换成八进制的步骤可以这样：

$$\begin{aligned}
 &[0.35;]_8 \\
 0.35 \times 8 &= 2.8 \Rightarrow [0; 2.8]_8 \\
 0.8 \times 8 &= 6.4 \Rightarrow [0; 2, 6.4]_8 \\
 0.4 \times 8 &= 3.2 \Rightarrow [0; 2, 6, 3.2]_8 \\
 0.2 \times 8 &= 1.6 \Rightarrow [0; 2, 6, 3, 1.6]_8
 \end{aligned}$$

总而言之，就是把整数部分作为这一位，而把小数部分乘以  $R$  加到下一位，这样“借位”。

至于  $R$  进制转  $T$  进制，思路还是以十进制作为桥梁，先转成十进制，再转成  $T$  进制。

## 二、八、十六进制间的特殊转换方法

前文介绍的一般方法，已经可以用于在二进制、八进制和十六进制数之间进行转换。但对于它们三者之间的转换来说，用十进制作为桥梁还是太麻烦了一点。在这里，我们有更好的方案。

以二进制数  $a$  转换成八进制数  $x$  为例，我们可以进行如下数学推导（看不懂请跳过）：

$$\sum_{n=-\infty}^{+\infty} a_n 2^n = \sum_{m=-\infty}^{+\infty} (a_{3m+2} 2^2 + a_{3m+1} 2^1 + a_{3m} 2^0) 8^m = \sum_{m=-\infty}^{+\infty} x_m 8^m$$

所以只要

$$x_m = a_{3m+2} 2^2 + a_{3m+1} 2^1 + a_{3m} 2^0$$

就可以保证这个转换是正确的。

具体来说，如何操作呢？我们可以在分号左右两边，每三位一组（如果不到三个数字，不妨在两侧补 0），把它转换成八进制。以  $(10101.1101)_2$  为例，我们把  $[1, 0, 1, 0, 1; 1, 1, 0, 1]_2$  以分号为界，向左向右每三位一组，转换成八进制，这就是图 C.6 中从上到下的过程。

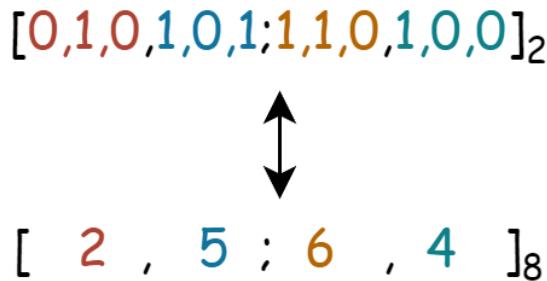


图 C.6: 二进制与八进制的转换

那么拿到  $(25.64)_8$  之后我们也可以反过来，把每一位转换成三位一组的二进制形式，也就是图 C.6 中从下到上的过程。

十六进制也是同理，

$$\sum_{n=-\infty}^{+\infty} a_n 2^n = \sum_{m=-\infty}^{+\infty} (a_{4m+3} 2^3 + a_{4m+2} 2^2 + a_{4m+1} 2^1 + a_{4m} 2^0) 16^m = \sum_{m=-\infty}^{+\infty} x_m 8^m$$

所以只要

$$x_m = a_{4m+3} 2^3 + a_{4m+2} 2^2 + a_{4m+1} 2^1 + a_{4m} 2^0$$

就可以保证这个转换是正确的。

具体操作和二进制转八进制相似，只是三位一组变成了四位一组。还以  $(10101.1101)_2$  为例，我们把  $[1, 0, 1, 0, 1; 1, 1, 0, 1]_2$  以分号为界，向左向右每四位一组（如果不到四个数字，不妨在两侧补 0），转换成十六进制，这就是图 C.7 中从上到下的过程。

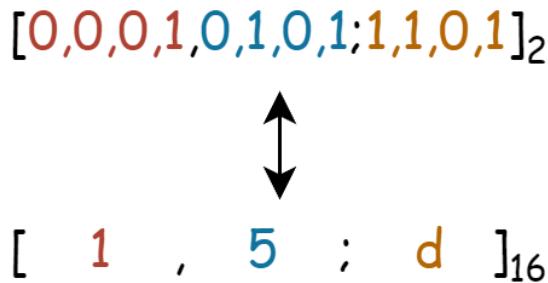


图 C.7: 二进制与十六进制的转换

那么拿到  $(15.d)_{16}$  之后我们也可以反过来，把每一位转换成四位一组的二进制形式，也就是图 C.7 中从下到上的过程。

如果要在八进制与十六进制之间进行转换，最简单的方式是以二进制为桥梁，先把  $(15.d)_{16}$  转换成  $(10101.1101)_2$ ，再转换成  $(25.64)_8$ 。

## C.2 布尔代数基础

跋

編程技巧馬行空，語法知識也錯綜。

偶遇疑難求上下，遭逢困惑問西東。

鑽研練就超群技，實踐能成鬼斧工。

道遠路長君莫嘆，錦囊盡在此書中。

