C++ Prime Plus 学习笔记

Calm Liu

2020.03.18

目录

预备知识

§ 1.4 程序创建的技巧

§ 1.4.1 创建源代码文件

在不同的平台上对于编写的 C++ 代码有不同的后缀名要求，见表1.1。

C++ 实现	源代码文件的扩展名
UNIX	C，cc，cxx，c
GUN C++	C，cc，cxx，cpp，c++
Digital Mars	cpp，cxx
Borland C++	cpp
Watcom	cpp
Microsoft Visual C++	cpp，cxx，cc
Freestyle CodeWarrior	cp，cpp，cc，cxx，c++

表 1.1: 源代码文件的扩展名

§ 1.4.2 编译和链接

Linux 下的编译和链接一般用 g++，常用的参数有：

编译

1. `g++ -E Test.cpp -o Test.i`，进行宏的替换，还有注释的消除，还有找到相关的库文件，生成.i 文件。
2. `g++ -S Test.cpp -o Test.s`，生成汇编文件，.s 文件。

3. `g++ -c Test.cpp -o Test.o`，生成目标代码（即机器码）文件，.o 文件。

链接

1. `g++ Test.o -o Test.exe`，链接单个目标文件，生成可执行文件。
2. `g++ Test1.o Test2.o Test3.o -o Test.exe`，链接多个目标文件，生成可执行文件。

命令参数

1. `g++ -c main.cpp -o hello.o`，-o <filename>：输出对应名称的文件。
2. `g++ -c main.cpp -I /usr/local -o hello.o`，-I <path>：把 path 指定的路径添加到头文件的搜索范围中。

其实 Linux 对于文件的后缀名没有强制规定，文件的后缀名只是提供了一个当没有指定的应用运行时能够提供默认应用的功能，对于这个文件本身来说并没有直接指定它的类型，Linux 下可以通过`file`命令来查看文件类型。

Windows 下的 **MinGW** 其实就是 Minimalist GNU For Windows。它实际上是将经典的开源 C 语言编译器 GCC 移植到了 Windows 平台下，并且包含了 Win32API 和 MSYS，因此可以将源代码编译生成 Windows 下的可执行程序，又能如同在 Linux 平台下时，使用一些 Windows 不具备的开发工具。

Debug 版本和 Release 版本的区别：

Debug 通常称为调试版本，通过一系列编译选项的配合，编译的结果通常包含调试信息，而且不做任何优化，以为开发人员提供强大的应用程序调试能力。

Release 通常称为发布版本，是给用户使用的，一般客户不允许在发布版本上进行调试。所以不保存调试信息，同时，它往往进行了各种优化，以期达到代码最小和速度最优。为用户的使用提供便利。

开始学习 C++

§ 2.1 进入 C++

§ 2.1.1 main() 函数

关于`int main()`和`void main()`，不过在 C99[2] 标准中，只有以下两种定义方式是正确的：

1. `int main(void)`
2. `int main(int argc, char* argv[])`

处理数据

§ 3.1 简单变量

§ 3.1.1 整型 short，int，long 和 long long

在不同的系统中，short，int，long，long long 的长度可能都是不同的，C++ 对其的标准为：

- short 至少 16 位
- int 至少与 short 一样长
- long 至少 32 位，且至少与 int 一样长
- long long 至少 64 位，且至少与 long 一样长

以下程序运行在 **CentOS** 7 64 位机上

```cpp
#include <iostream>
#include <climits>

int main(int argc, char* argv[]) {
   std::cout << sizeof(int) << std::endl; // 4
   std::cout << INT_MAX << std::endl; // 2147483647
   return 0;
}
```

climits 中还定义了许多符号常量，都是 limits.h 的套壳，具体可以查看 limits.h[1]文件，表3.1也给出了一些。

[1]linux 下一般在`/usr/include/limits.h`

符号常量	表示
CHAR_BIT	char 的位数
CHAR_MAX	char 的最大值
CHAR_MIN	char 的最小值
SCHAR_MAX	signed char 的最大值
SCHAR_MIN	signed char 的最小值
UCHAR_MAX	unsigned char 的最大值
SHRT_MAX	short 的最大值
SHRT_MIN	short 的最小值
USHRT_MAX	unsigned short 的最大值
INT_MAX	int 的最大值
INT_MIN	int 的最小值
UNIT_MAX	unsigned int 的最大值
LONG_MAX	long 的最大值
LONG_MIN	long 的最小值
ULONG_MAX	unsigned long 的最大值
LLONG_MAX	long long 的最大值
LLONG_MIN	long long 的最小值
ULLONG_MAX	unsigned long long 的最大值

表 3.1: climits 中的符号常量

§ 3.1.6 整型字面值

表示不同进制数的前缀，对于字符常量用 \ 代替 `0`：

- 二进制以 `0b` 开头
- 八进制以 `0` 开头
- 十六进制以 `0x` 开头

§ 3.1.7 C++ 如何确定常量的类型

放在数字常量后面的字母用于表示类型，整数后面的 l 或 L 表示该整数为 long 常量，u 或 U 后缀表示 unsigned int 常量，ul（不区分顺序和大小写）表示 unsigned long 常量，在 C++11 标准中，新加入了一些表示数字类型的后缀，具体见表3.2。

后缀字母[2]	**类型**
u	表示 unsigned int 类型
l	表示 long 类型或 long double 类型
ll	表示 long long 类型
ul	表示 unsigned long 类型
ull	表示 unsigned long long 类型
f	表示 float 类型

表 3.2: 数值后缀代表的类型

§ 3.1.8 char 类型：字符和小整数

论`\n`和`endl`的区别[3]：\n 仅代表换行的转义字符，endl 除了代表换行，还紧跟着清出缓冲槽。

cout 的意思是 console-output：控制台输出，就拿`cout << "Hello World!" << endl;`来说，cout 代表后面的内容输出到控制台的一个缓冲槽，而不是控制台，当遇到 endl 或者其他 flush 之类的命令或函数时，缓冲槽里的内容会按照顺序输出到控制台，再由控制台进行转义字符的识别打印。所以当你输入`cout << "Hello World!" << "\n";`时，其实就相当于输入`cout << "Hello World!" << "\n" << flush;`，程序会自动刷新输出流，所以只要你程序不炸，用 \n 比 endl 会快一点（不刷新输出流会快那么一点吧），但你如果程序够大并且明确要刷新输出流，就用 endl 吧。

C++ 支持直接输入 **Unicode** 码点，通用字符名可以以`\u`或`\U`打头，\u 后面跟的是 4 个十六进制位，\U 后面跟的是 8 个十六进制位，这里参见 C++ prime plus 英文版 987 页 [There are two forms of UCN sequences. The first is \u hexquad , where hexquad is a sequence of four hexadecimal digits; \u00F6 is an example. The second is \U hexquadhexquad ; \U0000AC01 is an example. Because each hexadecimal digit corresponds to four bits,

[2] 不区分顺序和大小写

[3] 具体可以查看 endl 与 \n 的区别这篇文章

the \u form can be used for codes representable by a 16-bit integer, and the \U form can be used for codes representable by a 32-bit integer.]，中文译版在第 52 页写的是 8 个十六进制位和 16 个十六进制位，应该是写错了。

```cpp
#include <iostream>
#include <string>

int main(int argc, char* argv[]) {
   std::string str = "\u2660\u2663\u2665\u2666";
   std::cout << str << std::endl; // ♠♣♥♦
   return 0;
}
```

一个 char 型占 1 个字节，也就是 2 个十六进制位，一般用来存放 ASCII 或者扩展的 ACSII，C++ 内部还有一个内建的`wcha_t`类型，在 CentOS 7 平台上测试其占 4 个字节，也就是 8 个十六进制位，可见其底层类型应该是 unsigned int，在另外一些系统中其底层类型可能为 unsigned short。如果你要用 wcha_t 类型，由于 cin 和 cout 都将输入和输出都视为 char 流，不适合用来处理 wcha_t 类型，iostream 头文件的最新版中有`wcin`和`wcout`工具专门用来处理 wcha_t 类型。另外，在一个字符常量前加 L 可以将其声明为宽字符常量，字符串也是一样。

```cpp
#include <iostream>

int main(int argc, char* argv[]) {
   wchar_t ch = L'\u262F';
   std::cout << sizeof(L'\u262F') << std::endl; // 4
   std::cout << ch << std::endl; // 9775
   std::wcout << ch << std::endl; // \
   return 0;
}
```

在这里我用的字符是 U+262F[☯]，可以看到宽字符常量占 4 个字节，cout 输出流将其视为 4 个 1 字节的字符常量（即 0x0000262F），转换成十进制时会忽略前面的 0，输出便是 9775，这里 wcout 的输出就是 ACSII 码的第 97 个，也就是\，想让 wcout 的输出正确的 Unicode 字符相当复杂[4]，而且不同的编译器实现也都不一样（虽然 C++ 标准制定的很好，那有人不遵守咋办嘛）。

Unicode 到目前为止已经定义了 17 个平面，除了第一个基本平面（简称 BMP 图3.1，其

[4]具体可以查看 cout，wcout 无法正常输出中文字符问题的深入调查这篇文章

码点从 U+0000 到 U+FFFF）用一个 wcha_t 型存就会浪费两个字节，其他 16 个辅助平面（其码点从 U+010000 到 U+10FFFD）用一个 wcha_t 型刚好存下，如果你的程序用到了非常多的国际字符，用 wcha_t 型可能就会存在空间浪费的情况（这里谈一下 UTF-8 吧，我觉得 UTF-8 流行的原因就是因为它兼顾了效率和容量，首先它实现了 Unicode 中的所有码点，其次它的变长编码保证了对于高频字符存储空间没有浪费，大概就像哈夫曼编码（Huffman Coding）一样吧），C++11 中新增了`char16_t`和`char32_t`类型，这两种类型都是无符号的，其中前者长 16 位，后者长 32 位。C++11 标准使用前缀 u 表示 char16_t 类型字符常量和字符串常量（例如 u '\u262F'），用前缀 U 表示 char32_t 类型字符常量和字符串常量（例如 U '\U0001F602'），这两个类型应该和 UTF-16 编码和 UTF-32 编码相对应吧，但鉴于 UTF-16 是变长编码（好吧，就是两个字节根本不够用），~~我也不知道有没有关系~~（在 Google 风格指南里明确了它们用于非 UTF-8 文本，但不推荐你使用这两个类型）。

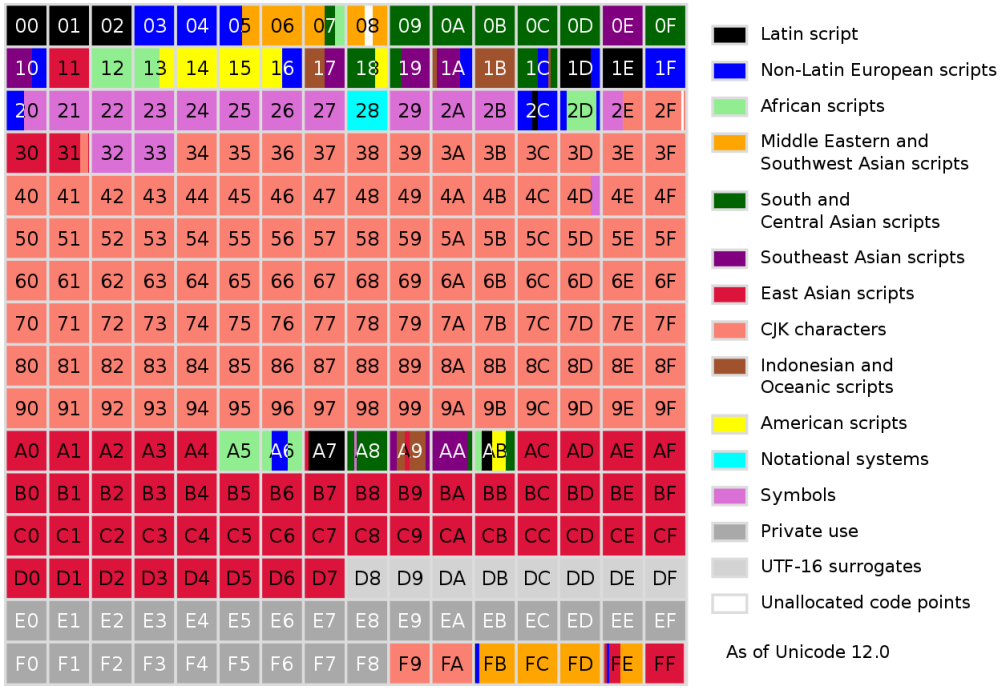

图 3.1: A graphical representation of Unicode's Basic Multilingual Plane(BMP)

搞了怎么多类型，有 char, wcha_t, char16_t, char32_t, 而且有的用起来还十分的不方便（这里提名 wcin 和 wcout），所以在上面那个程序里用的是`std::string`，其实 C++ 标准库已经帮我们实现好了 string 类，而且它相比`char* []`非常智能（到后面`std::cin.get()`的时候就会知道），如果你不是一定要用 char，用 string 会方便地多。

§ 3.2 const 限定符

C++ 中`#define`和`const`的区别：

- 用`#define MAX 255`定义的常量是没有类型的，所给出的是一个数，编译器只是把所定义的常量值与所定义的常量的名字联系起来，#define 所定义的宏变量在预处理的时候进行替换，在程序中使用到该常量的地方都要进行拷贝替换。用`const float MAX = 255`定义的常量有类型名字，存放在内存的静态区域中，在程序运行过程中 const 变量只有一个拷贝，而 #define 所定义的宏变量却有多个拷贝，所以宏定义在程序运行过程中所消耗的内存要比 const 变量的大得多
- 用 #define 定义的常量是不可以用指针变量去指向的，用 const 定义的常量是可以用指针去指向该常量的地址的
- 用 #define 可以定义一些简单的函数，const 是不可以定义函数的

在编译时，编译器通常不为 const 常量分配存储空间，而是将它们保存在符号表中，这使得它成为一个编译期间的常量，没有了存储与读内存的操作，使得它的效率也很高。宏定义的作用范围仅限于当前文件，且只作替换，不做计算，不做表达式求解。默认状态下，const 对象只在文件内有效，当多个文件中出现了同名的 const 变量时，等同于在不同文件中分别定义了独立的变量。如果想在多个文件之间共享 const 对象，必须在变量定义之前添加 extern 关键字（在声明和定义时都要加）。const 有数据类型，编译时会进行类型检查，而 #define 无类型，也不进行类型安全检查，可能会产生意想不到的错误，因此如果你不是明确知道自己在干什么，尽量只使用 const 常量而不使用宏常量。

§ 3.3 浮点数

§ 3.3.2 浮点类型

先说一下计算机中浮点数的存法吧，IEEE-754[1] 规定，对于 32 位的浮点数，最高的 1 位是符号位 S，接着的 8 位是指数 E，剩下的 23 位为有效数字 M，就拿图3.2中的这个浮点数来

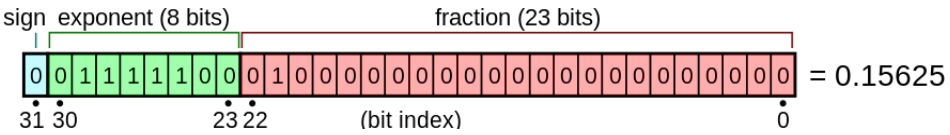

图 3.2: 0b01111100010000000000000000000000

说，其真值的计算公式为 $X = (-1)^S * M * 2^{E-127}$，其中 M 为尾数，127 称为偏移量，这只是一个最普通的偏移类型的浮点数，IEEE-754 还规定了很多类型的浮点数，在这里就不一一赘述了。还有就是我们生活中的很多十进制小数转换成浮点数的时候都会丢失精度，32 位的丢失的多一点，64 位的丢失的少一点罢了，就拿 0.7 这个数来说，十进制小数转换成二进制的规则是：小数点前的部分**除 2 取余逆序排列**小数点后的部分**乘 2 取整顺序排列**，所以 0.7 转

换成二进制小数就是 0.101100110...，你拿多少位的浮点数存都会有丢失精度的情况，只不过是丢失的多少罢了。

§ 3.4 C++ 算数运算符

§ 3.4.4 类型转换

C++11 对于类型转换有一个表，编译器会查这个表进行类型转换：

1. 如果有一个操作数的类型是 long double，则将另一个操作数转换为 long double
2. 否则，如果有一个操作数的类型是 double，则将另一个操作数转换为 double
3. 否则，如果有一个操作数的类型是 float，则将另一个操作数转换为 float
4. 否则，说明操作数都是整型，因此执行整型提升
5. 在这种情况下，如果两个操作数都是有符号或无符号的，且其中一个操作数的级别比另一个低，则转换为级别高的类型
6. 如果一个操作数是有符号的，另一个操作数是无符号的，且无符号操作数的级别比有符号操作数高，则将有符号操作数转换为无符号操作数所属的类型
7. 否则。如果有符号类型可表示无符号类型的所有可能取值，则将无符号操作数转换为有符号操作数所属的类型
8. 否则，将两个操作数都转换为有符号类型的无符号版本

上述都是在表达式中的类型转换，如果你在赋值时将一个值赋给一个取值范围更小的值，可能就会出现潜在的数值转换问题，见表3.3。

转换	**潜在的问题**
将较大的浮点类型转换为较小的浮点类型，如将 double 转换为 float	精度（有效位数）降低，值可能超出目标类型的取值范围，在这种情况下，结果将是不确定的
将浮点类型转换为整型	小数部分丢失，值可能超出目标类型的范围，在这种情况下，结果将是不确定的
将较大的整型转换为较小的整型，如将 long 转换问 short	原来的值可能超出目标类型的取值范围，通常只复制右边的字节

表 3.3: 潜在的数值转换问题

复合类型

§ 4.2 字符串

§ 4.2.1 拼接字符串常量

对于`char boss[8] = "Bozo"`这样的字符串赋值方法，后面的四个字符均会被赋成‘\0’。另外，字符串常量（使用双引号）不能与字符常量（使用单引号）互换，字符常量如 'S' 只是 83 的另一种写法，因此`char str = 'S'`表示将 83 赋给 str。但 "S" 不是字符常量，它表示的是两个字符（S 和 \n）组成的字符串，并且 "S" 表示的是这个字符串所在的内存地址，所以你如果写出`char str = "S"`这样的代码，编译器是要报错的，因为没有这种类型转换。

顺便说一说那个著名的“烫烫烫”和“屯屯屯”，在 Debug 模式下，VS 会把未初始化的栈内存全部填成 0xCC。会把未初始化的堆内存全部填成 0xCD。但是 Release 模式下不会有这种附加动作，原来那块内存里是什么就是什么（我在 CentOS 下用 g++ 编译输出发现每次输出的字符串都不一样，并且大多都是乱码，到这里我也不知道为什么了，可能要牵扯到 DRAM 原理了）。未初始化的变量会被系统赋初值为 0xCC，超过了 ASCII 码 0-127 这个范围，因此这个字符串被系统当成了宽字符组成的字符串，即两个字节数据组成一个字符，而 0xCCCC 表示的宽字符在 GBK 编码中正好是乱码中的那个烫字，0xCDCD 在 GBK 编码中就是屯字。

当然，我们也可以追踪一下这种行为，可以在 VS 中打断点调试然后查看反汇编代码，栈中的填充很容易追踪，堆中的填充就不太好追踪了。Linux 下你可以使用 objdump 命令来查看可执行文件的反汇编代码。

§ 4.2.4 每次读取一行字符串输入

对于普通的输入就比如`std::cin >> str`，std::cin 通过使用空白（空格，制表符和换行符）来确定字符串的结束位置，也就是说你想在一个字符串中输入空格用普通的 std::cin 是做不到的。istream 中的类如`std::cin`提供了两个面向行的类成员函数：`getline()`和

`get()`。这两个函数都读取一行输入，直到达到换行符。然而，随后 getline() 将丢弃换行符，而 get() 将换行符保留在输入序列中。

getline() 函数在 istream 中的原型为`__istream_type& getline(char_type* __s, streamsize __n)`（书上说是在后面会讲还可以有第三个可选参数的 getline()），它接受一个 char 型数组的指针和一个 streamsize（顾名思义就是字符串长度，就是流长嘛），当它读到换行符，且此时长度没有超过字符串的长度或者 streamsize，它就会把换行符换成 '\0'，如果超过了 streamsize 或者字符串的长度，它就会在 streamsize-1 的地方截断，让后把第 streamsize 位置为 '\0'，注意此时你的缓冲区是有那些被截断的字符的，如果你想要清空它们，可以使用`std::cin.clear()`，后面讲 **EOF** 的时候会再说这个函数。

get() 函数与 getline() 函数工作方式相近，它们接受的参数相同，解释参数的方式也相同，并且读取方式也相同，但是 get() 并不读取并丢弃换行符，而是将其留在输入流内，也就意味着当你第一次调用 std::cin.get(str, ArSize) 后，第二次调用这个函数会得到一个空的字符串（其实如果你不去管那个换行符，以后再读取都是空字符串），这就是因为换行符依旧在输入流内，而下一个 get() 函数认为已经读到行尾了。但不带任何参数的 std::cin.get() 可以读取下一个字符（C++ 的函数重载让 cin 变得非常的智能），因此可以拿它来清理这个换行符。由于 std::cin.get(str, ArSize) 返回的是一个 cin 对象，你可以使用`std::cin.get(str, ArSize).get()`的方法。同理，getline() 函数也可以这样调用，`std::cin.getline(str1, ArSize).getline(str2, ArSize)`表示把输入流中连续的两行分别读入 str1 和 str2 两个字符串。

可以看到，使用 getline() 函数更加方便，而使用 get() 函数能够让我们更加精细的去调控需要读取的字符。当 get() 函数读取到空行时，还会设置失效位 (failbit)，会将接下来的输出阻断，当然，你也可以用 std::cin.clear() 来重置输入流。

这里顺便讲一下关于回车换行的问题，在屏幕还不普及的时代，结果输出经常是依赖于所谓的电传打印机，打印头沿着打印杆从左向右移动并打印出一个个字符，当碰到一个回车符时 (CR，0x0D，\r)，打印机就指示打印头重新回到最左边的位置上，这即是传统意义上的回车了。回车符后常跟着一个换行符 (LF，0x0A，\n)，打印机收到换行符就会指示滚筒滚动，这样，打印头就对准了纸张上的新的一行，如果没有换行，新的打印输出就会重叠在上一行上，有时走纸不顺畅时也会造成这种后果。在 Windows 系统上，回车键会产生两个字符 CRLF，一起表示换行，Unix/Linux 之类的则单独用 LF 表示换行，而苹果的 MacOS 则单独用 CR 来表示换行，其实这里最符合传统的应该是微软，毕竟和电传打印机的回车换行是一致的，但是用一个字符表示换行符能够减少文件大小，这也是 Unix/Linux 和 MacOS 用一个字符来表示换行符的原因，至于为什么一个用回车，一个用换行，可能为了独树一帜吧。

§ 4.3 string 类简介

§ 4.3.5 其他形式的字符串字面值

同理，对于字符常量有用的前缀，对于字符串也可以用，L 代表 w_char，u 代表 char_16，U 代表 char_32。C++11 还支持 UTF-8 编码，你可以使用 u8 前缀来表示这种类型的字符串字面值，就比方说零宽无间断间隔符 [Unicode zero-width no-break space character]，它的编码是 '\xEF\xBB\xBF'，用 u8 前缀就可以写成 u8 '\uFEFF'。

C++11 还新增了原始 (Raw) 字符串，就是禁止转义啦，原始字符串需要以 R 开头，并且用 "(和)" 作为定界符，就比如`std::cout << R"(\n)" << std::endl`会直接输出 \n，这里这样定义定界符是为了能让你在字符串里直接输入 "，并且在输入原始字符串时，按回车键会在里面加入回车字符。当然，如果你想在原始字符串中包含 "(或)"，原始字符串允许你在 " 和 (之间包含其他字符，只要你也以这些字符结尾即可。同时，这些前缀可以混合搭配，并且不区分顺序。

§ 4.4 结构简介

§ 4.4.1 在程序中使用结构

说一下考试很喜欢考的结构体占用的内存问题，这个就牵扯到内存对齐的事情了，一般编译器都会自动帮你把内存对齐的（如果你想手动指定对齐或者禁止对齐请使用`#pragma pack(n)`，n 为对齐的宽度），下面这个例子运行在 CentOS 64 位机上。

```
#include <iostream>

int main(int argc, char* argv[]) {
   struct {
      int a;
      double b;
      short c;
   } A;
   struct {
      int a;
      short b;
      double c;
   } B;
```

```
14      std::cout << sizeof(A) << std::endl; // 24
15      std::cout << sizeof(B) << std::endl; // 16
16      return 0;
17  }
```

在结构体中，从结构体的首地址开始，假设地址从 0 开始。对结构体 A 来说，a 占 4 个字节，占从 0~3 的字节，b 是 double 类型占 8 个字节，占从 8~15 的字节，c 占两个字节，从 16~17 的字节。对结构体 B 来说，a 占 4 个字节，从 0~3，b 占两个字节从 4~6；c 占 8 个字节从 8~15。这就是内存对齐，对齐规则是按照成员的声明顺序，依次安排内存，其偏移量为成员大小的整数倍，0 看做任何成员的整数倍，最后结构体的大小为最大成员的整数倍（所以这里的 A 的大小是 24，而不是 18）。在 C++ 中规定了空结构体和空类的内存所占大小为 1 字节，因为 C++ 中规定，任何不同的对象不能拥有相同的内存地址。

系统进行内存对齐，可以提高 CPU 处理速率，而这项任务就交给编译器进行相应的地址分配和优化，编译器会根据提供参数或者目标环境进行相应的内存对齐，当然内存对齐也有硬件方面的原因，有些硬件规定了读取地址，如果指令地址和其规定地址不一致，则该硬件可能会发生崩溃等未知情况。

§ 4.7 指针和自由存储空间

§ 4.7.1 声明和初始化指针

前面已经说过，无论是在 32 位还是 64 位机上，int 型都是 4 个字节，而 int 型指针变量（其实无论什么型的指针都是一样，无非就是存了一个内存地址变量）在 32 位机上是 4 个字节，在 64 位机上是 8 个字节（不用担心内存地址存不下的情况，因为你根本弄不到那么大的内存，linux 的 64 位机的虚拟地址只有 48 位，为了节约空间），所以在 64 位机上指针的内建类型应该是 unsigned long。这里顺便讲一下一个 C++ 程序在内存中的空间分配，具体见图4.1[3]。

栈是向低地址扩展的数据结构，是一块连续的内存区域，这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，当申请的空间超过栈的剩余空间时，将提示溢出。因此，用户能从栈获得的空间较小。堆是向高地址扩展的数据结构，是不连续的内存区域。因为系统是用链表来存储空闲内存地址的，且链表的遍历方向是由低地址向高地址。由此可见，堆获得的空间较灵活，也较大。栈中元素都是一一对应的，不会存在一个内存块从栈中间弹出的情况，这里再说下去要讲到 C++ 里内存分配了，这些还是放到后面讲到 malloc() 的时候再说吧[1]。

[1]具体可以查看 Linux C 内存分配这篇文章

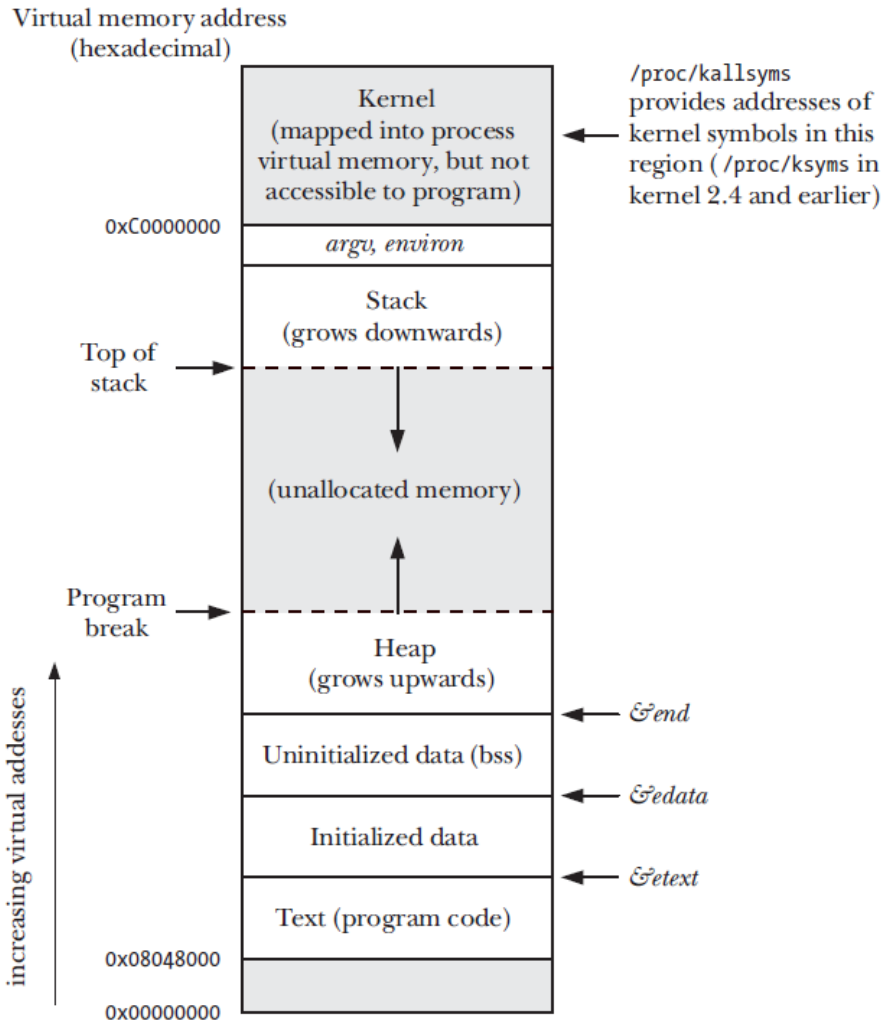

图 4.1: Typical memory layout of a process on Linux/x86-32

§ 4.7.2 指针的危险

在你没有将一个指针指向某一块你分配好的内存之前，不要对这个指针解除引用并赋值（也就是说，不要用一个右值来初始化指针），下面这个程序会说明这件事情：

```
1  #include <iostream>
2
3  int main(int argc, char* argv[]) {
4     long* fellow;
5     *fellow = 8888;
6     std::cout << *fellow << std::endl;
7     return 0;
8  }
```

这个程序能够正常过编译，但是运行时会报如下`[1] 14741 segmentation fault ./init_ptr`的错误，其中 14741 代表进程号，而 segmentation fault 就是段错误，所谓的段错误就是指访问的内存超过了系统所给这个程序的内存空间，通常这个值是由 gdtr（全局描述符寄存器）来保存的，它是一个 48 位的寄存器，其中的 32 位是保存由它指向的 gdt 表（全局描述符表），后 13 位保存相应于 gdt 的下标，最后 3 位包括了程序是否在内存中以及程序的在 CPU 中的运行级别，指向的 gdt 是由以 64 位为一个单位的表，在这张表中就保存着程序运行的代码段以及数据段的起始地址以及相应的断限和页面交换还有程序运行级别和内存粒度等信息，一旦一个程序发生了越界访问，CPU 就会产生相应的异常保护，于是 segmentation fault 就出现了。

有许多情况会导致 SIGSEGV，上面一种就是引用了空指针导致访问了不存在的内存而报错[2]。

§ 4.7.3 指针和数字

如果你想手动指定指针指向的内存地址，C++ 在类型一致方面要求更加严格，因此你需要显式地将数字转换成地址类型，你可以将`(int*) 0×B8000000`赋给一个 int 型指针。

§ 4.7.4 使用 new 来分配内存

new 分配的内存通常与常规变量声明分配的内存块不同，新建的变量和指针的值都储存在栈 (stack) 中，而 new 分配的内存是在堆 (heap) 或自由储存区 (free store) 中。如果你 new 请求内存失败，C++ 将返回 0，即生成了一个空指针，书上说后面会讨论这一异常。

§ 4.7.5 使用 delete 释放内存

使用 new 分配的内存在用完后需要用 delete 运算符归还这一块内存，你可以直接 delete 一个指针，这会释放这个指针指向的内存，而不会删除这个指针。如果你忘记 delete 一块 new 出来的内存，就会发生内存泄漏（当你编写一些复杂的程序时，内存泄漏可能会难以发现，好吧，VS 有性能探查器，可以帮你检测内存泄漏，毕竟宇宙第一 IDE）。但是尽管你 delete 掉这一块内存，你会发现它依旧是可以访问的，释放的内存系统可能不会直接使用，但后面会发生什么就没人知道了，如果你不想让这种事情发生，当你 delete 后请将指针置为 NULL。

§ 4.7.6 使用 new 来创建动态数组

在编译时给数组分配内存称为静态联编 (static binding)，这意味着数组实在编译的时候加入到程序中的。使用 new 时，你可以在程序运行时确定数组的长长度，这称为**动态联编 (dynamic binding)**，生成的数组叫做动态数组。

[2]具体可以查看 Segmentation Fault 错误原因总结这篇文章

为动态数组分配内存的通用格式为：`type_name * pointer_name = new type_name [num_elements]`，同样释放内存的格式也改为`delete [] pointer_name`，还有就是你没法用 sizeof 来确定动态分配的数组的长度，它只会返回其中一个元素的长度。

§ 4.8 指针，数组和指针算术

§ 4.8.2 指针小结

对指针来说，它的类型决定了每次偏移的地址量，即当你将 pointername ± 1 时指针移动的位数。因此对于一个数组，`arrayname[4]`和`*(arrayname + 4)`是一个意思。

对数组取地址时，数组名会被解释称其第一个元素的地址，即`&array_name[0]`，而直接对数组名取地址 (`&array_name`) 会返回同样的地址，但这次返回的是整个数组的地址，下面这个程序会解释这个现象：

```
#include <iostream>

int main(int argc, char* argv[]) {
    short pt[10];
    short *pd = new short [10];
    std::cout << &pt << std::endl; // 0x7ffec62c9010
    std::cout << &pt + 1 << std::endl; // 0x7ffec62c9024
    std::cout << pt << std::endl; // 0x7ffec62c9010
    std::cout << pt + 1 << std::endl; // 0x7ffec62c9012
    std::cout << &pd << std::endl; // 0x7ffec62c9008
    std::cout << &pd + 1 << std::endl; // 0x7ffec62c9010
    std::cout << pd << std::endl; // 0xd8a010
    std::cout << pd + 1 << std::endl; // 0xd8a012
    delete [] pd;
    return 0;
}
```

上面程序中 pt + 1 就是很普通的指针加法，而 &pt + 1 则直接跳到了整个数组的尾部，这也说明了对数组名取地址最然返回的也是第一个元素的地址，但其实它的长度是整个数组。另一个差别就是，要得到第一个元素的值，只需要对 pt 解除一次引用，但需要对 &pt 解除两次引用，即`**&pt == *pt == pt[0]`。

§ 4.8.4 使用 new 创建动态结构

对于新创建的动态结构或者指向结构的指针，你可以使用`pointer_name→member_name`，`struct_name.member_name`或者`*(pointer_name).member_name`的方式来访问里面的成员，其中 → 叫做间接成员运算符，而. 为成员运算符。

§ 4.10 数组的替代品

§ 4.10.1 模板类 vector

模板类 vector 类似于 string 类，也是一种动态数组，并且通过 new 和 delete 来管理内存，你可以使用`std::vector<type_name> vector_name(elenemnt_number)`，其中 element_number 可以是一个在编译期间不能确定的变量，好了，书上说的就只有怎么多了。

既然讲到了 vector，顺便说一下 C++ 的容器（不是 Docker 的那个容器），书上应该会在后面开一章来讲这个，这里就先随便讲一下吧。在 C++ 中容器的定义为：在数据存储上，有一种对象类型，它可以持有其它对象或指向其它对像的指针，这种对象类型就叫做容器。每一种容器都有其优点和缺点，所以为了应付程序中的不同需求，STL 实现了七种基本容器类型。

§ 4.10.2 模板类 array（C++11）

你可以通过`std::array<type_name elenemnt_number> array_name`来新建一个 array 对象。

§ 4.10.3 比较数组，vector 对象和 array 对象

无论是数组，vector 对象和 array 对象，都可以使用标准数组表示法来访问各个元素（但其实这些容器都有自己实现的访问方法），对于数组的越界访问，C++ 是不会禁止的，也就是说你可能会通过数组的下标来访问一些数组外的内存地址，这种用容器都是可以避免的。

循环和关系表达式

§ 5.4 基于范围的 for 循环（C++11）

C++11 新增了一种基于**范围 (ranged-based)** 的 for 循环，这简化了对于数组或容器类（对于容器可能用迭代器会更好）的循环，即：

```cpp
#include <iostream>
#include <vector>

int main(int argc, char* argv[]) {
   int arr[] = {1, 2, 3};
   std::vector<std::string> vec = {"abc", "def"};
   for (int i : arr) {
      std::cout << i << std::endl; // 1 2 3
   }
   for (std::vector<std::string>::iterator itor = vec.begin();
        itor != vec.end(); itor++) {
      std::cout << *itor << std::endl; // abc def
   }
   return 0;
}
```

§ 5.5 循环和文本输入

§ 5.5.1 使用原始的 cin 进行输入

这里底下都是要解决循环读取来自键盘的文本输入，并以一个你喜欢的方式结束读取的问题。你首先肯定想到利用 while 循环和某一个特殊字符来完成这个工作，即：

```cpp
#include <iostream>

int main(int argc, char* argv[]) {
   char ch;
   std::cin >> ch;
   while (ch != '#') {
      std::cin >> ch;
   }
   return 0;
}
```

上面的这个程序确实能够完成不断从键盘中读入文本，并以 # 结束的工作，但你很容易发现，你没办法输入空格或者换行符，因为这些字符都会被 cin 过滤，你输入的信息会先送入缓冲区，当你按下回车，这些字符才会被发送给 cin，这也就是你甚至能够在 # 后面输入字符，虽然这并没有什么用。

§ 5.5.2 使用 cin.get(char) 进行补救

如果你想逐个字符地检查，`std::cin.get(ch)`可以帮你实现这一点，它会读取输入流中的下一个字符，即使它是空格，即：

```cpp
#include <iostream>

int main(int argc, char* argv[]) {
   char ch;
   std::cin.get(ch);
   while (ch != '#') {
      std::cin.get(ch);
   }
   return 0;
}
```

§ 5.5.3 使用哪一个 cin.get()

对于 std::cin.get() 这个函数到现在应该已经出现过了 3 个版本，总共有 4 个重载的版本：

1. int cin.get()
2. istream& cin.get(char& var)

3. istream& cin.get(char* s, streamsize n)
4. istream& cin.get(char* s, streamsize n, char delim)。

它们的作用我们前面已经分别讲过，其中 streamsize 在 Linux 下被定义为 long int 型，最后那个 delim 参数为中止字符，具体你可以看 istream.tcc[1]这个文件里面那些好长的模板类函数。后面三个类型的返回值都是 cin 对象，也就是说，你能够写出像`std::cin.get(ch1).get(ch2)`这种代码，它将两个字符连续地读入 ch1 和 ch2。

§ 5.5.4 文件尾条件

使用 # 来结束输入总很难让人满意，因为你有的时候可能回想输入 #（其实就是不够优雅），这个时候你可以尝试使用**检测文件尾 (EOF)** 的方法来告知程序结束输入。EOF 是 end of file 的缩写，Wiki 上对其的定义为 a condition in a computer operating system where no more data can be read from a data source，就是文件尾的一个特殊标识符，当然你也可以通过键盘来模拟 EOF，在 Linux 下是 Ctrl + D，在 Windows 下是 Ctrl + Z 和 Enter。在检测到 EOF 后，cin 会将 **eofbit** 和 **failbit** 都设置为 1，你可以通过`std::cin.eof()`和`std::cin.fail()`两个函数来监测是否达到了文件尾，如果 eofbit 被设置为 1，则 eof() 会返回 bool 值 true，如果 eofbit 或 failbit 被设置为 1，则 fail() 均会返回 bool 值 true，使用 fail() 会比 eof() 更广泛，因为 failbit 还能监控到其他故障（磁盘故障）。

当 cin 检测到 EOF 后，会设置 eofbit 和 failbit，此后 cin 将会停止读取输入，再次调用也不管用（对于文件输入，这有助于程序读取不会超出文件尾）。但对于键盘输入，你可能后面想要再次调用 cin，这时可以用`std::cin.clear()`来清除 EOF 标记，但在有些系统中，可能 Ctrl + Z 就会结束输入输出，用 clear() 也无法恢复。

同时，由于 std::cin.get(char) 的返回值为 cin，你可以将这个直接作为循环的条件，也就是说上面的 while 语句可以改写成`while (std::cin.get(ch))`，而且这样可以去掉 while 前和 while 内的 get() 语句。

§ 5.5.5 另一个 cin.get() 版本

为了使用 syd::cin.get(), 需要知道它对于 EOF 的处理。当该函数到达 EOF 时，将返回一个用符号常量 EOF 表示的特殊值，通常这个值为-1，因为没有 ASCII 码为-1 的字符，也就不会与任何输入的字符弄混，所以你只需要在程序中使用 EOF 即可。对于`std::cin.get()`和`std::cin.get(ch)`的区别，见表5.1。

有关能够让你完全正确输入的程序（针对那种不想好好的用你的程序而在键盘上狂敲 EOF 的用户），会在 §6.7 给出。当然，你也可以用 std::cin.get() 函数来重写上面的程序，即:

[1]linux 下一般在`/usr/include/C++/[0~9]/istream.tcc`

属性	cin.get(ch)	ch = cin.get()
传递输入字符的方式	赋给参数 ch	将函数返回值赋给 ch
用于字符输入时函数的返回值	istream 对象（执行 bool 转换后为 true）	int 型的字符编码
到达 EOF 时函数的返回值	istream 对象（执行 bool 转换后为 true）	EOF

表 5.1: cin.get(ch) 与 cin.get()

```
#include <iostream>

int main(int argc, char* argv[]) {
   char ch;
   while ((ch = cin.get()) != EOF) {
      // do something
   }
   return 0;
}
```

对于上面程序的 while 判断中的`(ch = cin.get()) != EOF`，由于赋值语句的值就是左操作数的值，所以这里 ch = cin.get() 返回的就是 ch 的值，这也是这里为何可以进行判断的原因。

§ 5.6 嵌套循环和二位数组

假设你现在有一个二位数组`int arr[4][5]`，你想访问里面的第 3 排第 4 列的那个元素，现在你有以下四种办法（排名不分先后）：

- arr[2][3]
- *(arr + 2)[3]
- *(*(arr + 2) + 3)
- *(*arr + 13)

也就是说，对于一个二位数组，其实它的本质是一个有行数个元素的一维指针数组，其中每一个指针指向每行的有列数个元素的一位数组的第一个元素，所以对于二位数组名，它应该是一个指向一个指针数组的指针。

分支语句和逻辑运算符

§ 6.2 逻辑表达式

§ 6.2.3 用 && 来设置取值范围

如果你想判断一个值大于一个数并且小于一个数，不要把代码写成`0 < score < 100`（当然如果是 python 你可以这么做也最好怎么做），因为编译器通过从左向右的结合法则会把上面的代码当成`(0 < score) < 100`来执行，也就是说不管如何前面表达式的值要么是 0，要么是 1，都会小于 100，所以正确的方法应该是`socre > 0 && score < 100`。

§ 6.2.6 逻辑运算符细节

对于复合比较的语句，&& 运算符的优先级要高于 || 的优先级，并且 C++ 确保程序从左向右计算逻辑表达式，并且在知道答案后立即停止，也就是说你可以写出`x =≠ 0 && 1.0 / x > 100`这样的代码而不用担心编译器会报错。

§ 6.4 ?: 运算符

条件运算符 (?:)，又叫三元运算符，该运算符的通用格式为 expression1 ? expression2 : expression3，当 expression1 为真时，表达式返回 expression2 的值，反之返回 expression3 的值。

§ 6.6 break 和 continue 语句

break 和 continue 语句都能使程序跳过当前循环中一部分代码，可以在 switch 语句中使用 break 语句来跳过剩余的 case 以及 default，当 **break** 用于循环中时，它会跳过循环的

剩余部分去执行下面的语句，而 **continue** 用于循环中时会跳过循环体的剩余部分，而开始新一轮的循环。当然，如果你偏爱 goto，它也能完成工作，只是会降低代码的可读性罢了。

§ 6.7 读取数字的循环

现在你接到了一个来自甲方的任务，是要写一个能够录入分数的程序，但你面向的客户也许根本不知道什么是键盘，所以你要保证你的程序不会有任何 bug（因为书上在这段程序中用到了 continue，所以在这里才讲到，但我感觉书上的程序还是有 bug，所以我改写了一下）。

```
#include <iostream>
#include <limits>

int main(int argc, char* argv[]) {
   short score;
   std::cin.sync();
   while (!(std::cin >> score) || (score < 0 || score > 100) ||
         std::cin.get() != '\n') {
      std::cin.clear();
      std::cin.ignore(std::numeric_limits<std::streamsize>::
         max(), '\n');
      std::cout << "Invalid input!" << std::endl;
   }
   return 0;
}
```

这里首先使用 std::cin.sync() 函数来清空 cin 缓冲区里面未读取的信息，接下来的!(std::cin » score) 对应输入非法字符（包括检测到 EOF）的情况，(score < 0 || score > 100) 对应当你输入的分数不合法的情况，而 std::cin.get != '\n' 则会检查当你输入类似于像 85c 这样的字符时 cin 会帮你自动截断当成 85 的情况，一旦当你输入的类型有问题时，首先使用 std::cin.clear() 来重置 EOF 改变的标志位，接着用 std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); 来清空缓冲区剩余的字符（这里使用 numeric_limits 宏来获取输入流中字符数量上限，将 '\n' 作为 delim 告知函数清除到换行符），最后告知用户 Invalid input!

§ 6.8 简单文件输入/输出

§ 6.8.3 读取文本文件

当你用你定义的 std::ifstream 对象（暂且就叫它 fcin）来打开文件时，可以使用`fcin.is_open()`来判断文件是否被正常地打开，如果文件被成功打开，方法 is_open() 会返回 true，反之则会返回 false，如果你想让程序在打开文件失败后中止，你可以使用 **cstdlib** 中定义地 exit() 函数，在该头文件中，还定义了一个用于通操作系统通信的宏 EXIT_FALIURE，也就是说，你可以使用`exit(EXIT_FALIURE)`来终止程序。

```
#include <iostream>
#include <fstream>
#include <cstdlib>

int main(int argc, char* argv[]) {
   char ch;
   std::ifstream fcin;
   fcin.open("file");
   if (fcin.is_open() && fcin.peek() ≠ EOF) {
      while ((ch = fcin.get()) ≠ EOF) {
         // do something
      }
   }
   else {
      exit(EXIT_FAILURE);
   }
   fcin.close();
   return 0;
}
```

在上面的程序中用到的 peek() 函数能够读取并返回下一个字符，但并不提取该字符到输入流中，也就是说你可以用这个函数来窥探输入流中的下一个字符，但并不将它提出。这里使用 peek() 函数来判断 EOF 而不用 eof() 函数是因为 eof() 返回 true 的条件是读到**文件结束符 (0xff)**，而不是文件内容的最后一个字符，也就意味着用 eof() 来判断总会迟滞一位。如果你在这里的 if 判断用`fcin.eof()`，你会发现它无法判断出空文件，因为当你打开一个文件时，文件指针位置为 0，并不是指向第 1 个字符，所以这里用的 peek() 会返回输入流中可用的下一个字符（就是文件指针指向的下一位），对于空文件来说就是 EOF。

如果你在这里的 while 判断用的是`!fcin.eof()`，而把 fcin.get(ch) 写在循环里，你

会发现文件的最后一个字符被读取了两次，就是因为当输入流中的 EOF 被读入时并不会改变 ch 的值，而是将 eofbit 置为 1。

但如果你想输入的不是 char 型的话，peek() 会表现很差，具体会在后面章节详细说明。

函数——C++ 的编程模块

§ 7.3 函数和数组

§ 7.3.2 将数组作为参数意味着什么

如果你想在函数之间传递数组时，不要尝试同时传递数组指针和数组长度（就像`void fillArray(int arr[size]);`这样），而应该将数组指针和数组长度作为两个分别的参数来传递，即应该`void fillArray(int arr[], int size);`，因为你传递的数组指针本身并不能说出整个数组有多长。

§ 7.3.4 使用数组区间的函数

在上面的办法中，我们使用了一个指针来标识数组的开头，用另一个参数来标识数组长度，如果你对这种办法感到不满，也可以用双指针（一个标识数组开头，一个标识数组结尾）来传递数组，这样也可以达到一样的效果。但在这个过程中，你要记得对于比方说`int arr[size]`的数组来说，指针 arr + size 指向的是最后一个元素的后面一个的位置，指针 arr + size - 1 才指向最后一个元素。

§ 7.3.5 指针和 const

将 const 用于指针可以有两种办法，第一种是将指针指向一个常规变量，这样会使通过这个指针访问的地址内的值不可修改，第二种是将它设置成一个常量指针，这样会使你不能修改这个指针指向的地址。

对于第一种情况，将一个指针指向一个常规变量，也就是`const int* pt = &age`，这个常规变量 age 可以是 const 也可以不是 const。对于 const int age，反正大家都改不了，但对于 int age，只有通过 pt 指针访问的 age 才不能改（也就是 *pt）。并且，你不能把一个 const 变量赋给一个没有 const 的指针，原因就是它本身就不准改。但当关系到了二层间接关

系，情况就变得复杂了，这个时候你连将非 const 变量赋给带有 const 的指针都将变得违法，因为一旦允许，你就可以写出下面的代码：

```
#include <iostream>

int main(int argc, char* argv[]) {
    const int **pt2;
    int *pt1;
    const int age = 22;
    pt2 = &pt1;
    *pt2 = &age;
    *pt1 = 21;
    return 0;
}
```

上面这段程序是过不了编译的，编译器会告诉你`error: assigning to 'const int **' from incompatible type 'int **'`，原因也很简单，在一层间接的规则下，你似乎可以用两层间接来合法修改一个 const 的值，这肯定是不被允许的，所以将一个非 const 变量赋给 const 指针在两层间接下会报错，这个行为也会影响数组，因为如果这是一个指针数组，它也需要解除两层引用，这个也被认为是两层引用，同样也会报错。当然，在你确定你的函数不需要修改传来的指针里的数据或者你想保护原始数据，尽量在指针前加上 const。

对于第二种情况，将一个指针设置成常量指针，也就是`int* const pt = &age`，这样会使这个指针与 age 变量绑定，你就不能再修改 pt 指针指向的内存地址。

当然，你也可以使用两个 const，也就是`const int* const pt = &age`，这个时候，不论 pt 还是 *pt 都是 const[1]。

§ 7.9 函数与 array 对象

对于一个指向 array 数组的指针，比如说`std::array<double, size>* pt`，如果要数组某一个元素的值，应该`(*pt)[i]`，因为 *pt 为这种对象，(*pt)[i] 才是该对象里面的元素。

[1]具体可以参见 const (computer programming) 这篇文章

§ 7.11 函数指针

§ 7.11.1 函数指针的基础知识

在 C++ 中，函数也可以被当成参数传递，函数名就是函数的地址，只要你不带 ()，就不会 call 这个函数，而是代表这个函数所在的地址。同时，你也可以声明函数指针，比如对于函数`double pam(int)`，你可以声明一个形如`double (*pt)(int)`的函数指针，这里 *pt 的括号不能省略，否则你就在定义一个返回值为 double* 的函数。如果你有一个函数接收 pt 指针作为参数，那它的原型应形如`void estimate(int lines, double (*pt)(int ));`。对于一个指向函数的指针来说，你可以通过`(*pt)`来调用这个函数，也可以直接用 pt 来调用这个函数，对于这个问题，书上是这样解释的（为避免翻译造成的意义不准确，这里用了英文原版）：

Holy syntax! How can pf and (*pf) be equivalent? One school of thought maintains that because pf is a pointer to a function, *pf is a function; hence, you should use (*pf)() as a function call. A second school maintains that because the name of a function is a pointer to that function, a pointer to that function should act like the name of a function; hence you should use pf() as a function call. C++ takes the compromise view that both forms are correct, or at least can be allowed, even though they are logically inconsistent with each other. Before you judge that compromise too harshly, reflect that the ability to hold views that are not logically self-consistent is a hallmark of the human mental process.

§ 7.11.3 深入探讨函数指针

对于一个很复杂的函数来说，你如果想声明一个指向这个函数的函数指针会变得十分复杂，就比方说对于如下三个函数：

- const double* fun1(const double arr[], int size);
- const double* fun2(const double* arr, int size);
- const double* fun3(const double [], int);

你如果想声明一个指向由这三个函数构成的一个数组（`const double* (*pa[3])( const double*, int) = fun1, fun2, fun3;`）的指针，这个指针的声明会变得十分复杂，正确的格式应该是`coust double* (*(*pt)[3])(const double*, int) = &pa;`，因为 *pt 是这个数组，*pt[1] 是指向 fun1 的函数指针，而函数指针需要再解除一层引用，所以你可以通过`(*(*pt)[1](av, 3))`的方式来调用函数。当然，C++11 有自动类

型推断的功能，你也可以这样声明`auto pt = &pa`，这能让你的工作变得简单，但你要保证你给的类型是你想要的。

§ 7.11.4 使用 typedef 进行简化

如果你想多次声明一个类型，而它的类型名又太长，这时候你可以通过 typedef 进行简化，来创建一个类型别名，即`typedef typeName yourName`。

函数重载

§ 8.1 C++ 内联函数

对于一些长度较短的函数（Google 风格指南里对可以内联的函数要求为不超过十行，不包含 for 以及 switch 语句），可以进行内联处理，你可以在函数声明以及定义前加上 **inline** 关键字来声明一个内联函数，通常的做法是省略原型，将整个定义写在函数的声明位置。内联函数以代码复制为代价，省去了函数参数压栈和弹出的时间，从而提高函数的执行效率。

有些函数即使声明为内联的也不一定会被编译器内联，比如虚函数和递归函数就不会被正常内联。通常，递归函数不应该声明成内联函数（递归调用堆栈的展开并不像循环那么简单，比如递归层数在编译时可能是未知的，大多数编译器都不支持内联递归函数）。虚函数内联的主要原因则是想把它的函数体放在类定义内，为了图个方便，抑或是当作文档描述其行为，比如精短的存取函数。而有的函数即使你没有显式地声明为内联函数如果你开了优化编译器可能也会帮你内联（编译器将内联扩展选项和关键字视为建议，不能保证所有函数都将以内联方式展开，您可以禁用内联扩展，但是不能强制编译器内联特定函数），MSVC 编译器可以指定 /Ob 参数来控制内联展开，gcc 只有在开了-O 优化才会处理内联。在语句层面的 inline 优化根本没有编译器的优化强大，所以 inline 关键字差不多可以被弃用了。

对于内联和宏的区别，宏是由预处理器对宏进行替代，而内联函数是通过编译器控制来实现的。而且内联函数是真正的函数，只是在需要用到的时候，内联函数像宏一样的展开，所以取消了函数的参数压栈，减少了调用的开销。你可以像调用函数一样来调用内联函数，而不必担心会产生于处理宏的一些问题，它们的代码效率是一样，但是内联函数要优于宏定义，因为内联函数遵循的类型和作用域规则，它与一般函数更相近，也有严格的参数检查。

§ 8.2 引用变量

§ 8.2.1 创建引用变量

对于 & 符号，还有另外一个含义，就是你可以用它来声明一个引用变量，比方说`int& rodents = rats;`，在这里 rodents 是 rats 的别名，它们的值和地址都是相同的，其实你就可以把左值引用看做是指针的语法糖，当然引用更加接近 const 指针，必须在创建的时候进行初始化，并且只能与一个变量绑定，不允许绑定新的变量，也就是说，你声明一个 const 指针也能够达到同样的效果。

§ 8.2.2 将引用作为函数参数

对于左值引用（书上在这里都是在讨论左值引用），Google 风格指南指出，所有按引用传递的参数都必须加上 const，确实普通的左值引用传参在函数里可能会导致歧义，对于一个封装好的函数而言，用户可能会难以辨别哪些参数是拷贝传递哪些参数是引用传递的，所以如果你明确要修改变量的值，请用指针，虽然这可能会产生二级指针这样导致代码阅读困难的东西，但总比引起误解来的好，你如果只是想传递一个不需要修改的数组或结构，就声明一个 const 引用。除非你明确知道自己在干什么（标准库中使用左值引用做参数的 API 还是存在的），否则不要用非 const 的引用来作为函数参数。

§ 8.2.3 引用的属性和特别之处

如果实参类型与引用参数不匹配时，C++ 将生成临时变量，对于带 const 的引用参数，编译器在以下两种情况生成临时变量：

1. 实参的类型正确，但不是左值
2. 实参的类型不正确，但可以转换为正确的类型

左值 (lvalue) 参数是可以被引用的数据对象，包括变量，数组元素，结构成员，引用和解除引用的指针等等，简单来说就是一个有明确存储地址的一个值，反之非左值就是右值，即一些表达式以及不能取地址的值（在 C 语言中的左值指的就是可以出现在表达式左边的值，但现在 const 变量也被视为左值，虽然它是一个不可修改的左值）。

如果实参的类型不匹配造成的类型转换，编译器将创建类型正确的临时变量，但如果这里的参数类型不带 const，编译器将会报错 [a reference of type "int &" (not const-qualified) cannot be initialized with a value of type "long"]，只有对于 const 类型的引用，编译器才会允许进行正确的类型转换。

§ 8.2.4 将引用用于结构

这里讨论有关返回引用的问题，你可以把函数的返回值设为引用，但你应该避免返回的引用对象在函数运行完就不复存在的情况，因此最好返回函数调用前就已经存在的对象的引用，或者你也可以返回在函数中 new 的堆空间，但这样可能会导致在函数运行完毕后忘记释放空间，从而导致内存泄漏。普通的返回会将函数的返回值先复制到一个临时位置，让后再做操作，而返回引用仅仅是挪一挪指针而已，这样的操作效率会搞得多。

§ 8.3 默认参数

你可以在声明函数原型时对参数添加一些默认值，这样在函数没有接收到某些参数时仍能通过默认值运行。但是在 Google 风格指南指出：缺省参数会干扰函数指针，导致函数签名 (function signature) 与调用点的签名不一致，即在一个现有函数添加缺省参数，就会改变它的类型，那么调用其地址的代码可能会出错。此外，缺省参数会造成臃肿的代码，毕竟它们在每一个调用点都有重复（调用函数的代码表面上看来省去了不少参数，但编译器在编译时还是会在每一个调用代码里统统补上所有默认实参信息，造成大量的重复）。因此在需要用到默认参数的地方，建议你都用函数重载，因为函数重载均没有这些问题。

这里提到了一个函数签名的概念，函数签名包含了一个函数的信息，包括函数名，参数类型，所在的类和名称空间及其他信息，函数签名用于识别不同的函数。注意区别这个概念和名称修饰的区别，名称修饰是在编译器及链接器处理符号时，使用某种名称修饰的方法，使得每个函数签名对应一个修饰后名称 (decorated name)。编译器在将 C++ 源代码编译成目标文件时，会将函数和变量的名字进行修饰，形成符号名，也就是说，C++ 的源代码编译后的目标文件中所使用的符号名是相应的函数和变量的修饰后名称。C++ 编译器和链接器都使用符号来识别和处理函数和变量，所以对于不同函数签名的函数，即使函数名相同，编译器和链接器都认为它们是不同的函数。

§ 8.4 函数重载

§ 8.4.1 重载示例

函数重载又叫做函数多态，它可以利用函数的参数列表，也叫做**函数特征标 (function signature)** 来区分同一个函数名称的不同版本，首先函数特征标不区分是否为引用类型，因为对于函数调用而言，编译器无法区分你传入的参数是需要拷贝传递还是引用传递，其次在匹配函数时，并不区分函数的返回值，也就是说你不能够通过返回值来重载函数，但不同特征标的函数可以有不同的返回值

类设计和 STL 经常使用引用参数，其中的重载的类型匹配有三种：左值引用参数与可修改的左值参数能够匹配，const 左值引用参数能够与可修改的左值，const 左值和右值参数匹配，而右值引用参数只能与右值相匹配，在这里 const 左值引用能够与三者都匹配，如果重载使用这三种参数的函数，编译器将选择最匹配的版本来调用。

§ 8.4.2 何时使用函数重载

书上在这里的说法是能够用默认参数解决的问题就不要上函数重载了，这里应该面向一些小的项目，函数重载还会多浪费内存，而上面的 Google 风格指南应该是面向大的 project，建议你用函数重载。

§ 8.5 函数模板

§ 8.5.3 显式具体化

对于模板的显式具体化，C++98 标准用以下的规则：

1. 对于给定的函数名，可以有非模板函数，模板函数和显式具体化模板函数以及它们的重载版本。
2. 显式具体化的原型和定义应以 template<> 打头，并通过名称来指出类型。
3. 具体化优先于常规模板，而非模板函数优先于具体化和常规模板。

§ 8.5.4 实例化和具体化

实例化 (instantiation)，这个词的意思就像你从类里面实例化出一个对象一样，当编译器使用模板为特定类型生成函数定义的过程就是在实例化，从一个高度抽象的模板中生成一个具体的函数，也就是模板实例。最初的编译器只支持**隐式实例化 (implicit instantiation)**，也就是你只能让编译器去选择该怎么来实例化一个模板，但现在的 C++ 还可以**显式实例化 (explicit instantiation)** 一个模板，这意味着你可以直接命令编译器来创建特定类型的模板实例，即`template void fun<int>(int);`，当然你也可以简写，<int> 可以写成 <> 甚至去掉，编译器会推导出推导出模板实参。书上把这句话放在了 main() 函数内，但是这样似乎做会报错，Intellisence 会警告你 explicit instantiation is not allowed in the current scope，所以应该是只能在函数外进行显式实例化。具体显式实例化有什么用，好像是没什么用的（应该是我水平不够，看不出来它有什么用）。当然你也可以在程序中使用函数来创建显式实例化，比如说这样去调用函数`fun<int>(1)`，如果你这里的实例化类型与参数类型不匹配，编译器就会将其转换为正确的类型，如果这里不能正确转换，编译器会报错。

这里还有一个概念就是**显式具体化 (explicit specialization)**，与显式实例化不同，显式具体化的函数需要有自己的函数声明，因为它们本来就是为了解决某些模板解决不了的类型。你可以这样声明`template <> void fun<int>(int);`，当然你也同样可以简写，<int> 可以写成 <> 或者没有。如果你试图在同一个文件中将同一个类型显式实例化和显式具体化，编译器将报错。

§ 8.5.5 编译器选择使用哪个函数版本

对于函数重载，函数模板，函数模板重载导致有多个备选函数时，编译器会进行一个叫**重载解析 (overloading resolution)** 的过程，其步骤如下：

1. 创建候选函数列表，其中包含与被调用函数名称相同的重载函数和模板函数。
2. 使用候选函数列表创建可行函数列表。这些都是参数数目正确的函数，为此有一个隐式转换序列，其中包括实参类型与相应形参类型完全匹配的情况。
3. 确定是否有最佳的可行函数。如果有则使用，否则将调用出错。

在确定最佳函数时，编译器会根据一下顺序来挑选函数：

1. 完全匹配，但允许一些无关紧要的转换（见表8.1），但常规函数优先于模板
2. 提升转换，例如 char 和 short 转换为 int，float 转换为 double
3. 标准转换，例如 int 转换为 char，long 转换为 double
4. 用户定义的转换，例如类声明的转换

从实参	**到形参**
Type	Type&
Type&	Type
Type[]	*Type
Type(argument-list)	Type(*)(argument-list)
Type	const Type
Type	volatile Type
Type*	const Type*
Type*	volatile Type*

表 8.1: 完全匹配允许的无关紧要的转换

如果这些函数都是完全匹配，则选择顺序为：非模板函数 > 显式具体化 || 显式具体化 > 隐式实例化。如果两个完全匹配的函数都是隐式实例化模板函数，则选择最具体 (most specialized) 的模板函数，这里的最具体指的是编译器推断使用哪种类型时执行的转换最少。用于找出最具体的模板的规则叫做函数模板的部分排序规则 (partial ordering rules)，这一部分比较复杂，参见 cppreference[重载决议]。

§ 8.5.6 模板函数的发展

C++11 新增了一个 **decltype** 关键字，你可以用一个表达式来声明变量的类型，这在编写模板时非常好用，因为有时你要声明的类型可能暂时无法确定，这时就可以用表达式来代替，即`decltype(expression) var;`。编译器在确定 var 的类型时会遍历一个核对表，其简化版如下：

1. 如果 expression 是一个没有被括号括起的标识符，则 var 的类型与改标识符的类型相同，包括 const。
2. 如果 expression 是一个函数调用，则 var 的类型与函数的返回类型相同（注意这里并不会真的去调用函数，编译器只是回去查看函数的原型来确定返回类型）。
3. 如果 expression 是一个被括号括起的左值，则 var 为指向其类型的引用。
4. 如果前面的条件均不满足，则 var 与 expression 的类型相同。

有时，函数的返回值类型也无法确定，为此 C++11 加入了**后置返回类型 (trailing return type)**，你可以这样来声明一个模板：`template <typename T, typename U> auto add(T x, U y) → decltype(x + y);`,但是 Coogle 风格指南指出这样的写法可能让某些读者陌生，它建议采用以前的 std::declval() 函数，其声明为：`template <typename T, typename U> decltype(std::declval<T&>() + std::declval<U&>()) add(T x, U y);`，但是对于 Lambda 表达式来说，后置返回类型是显式指定 Lambda 表达式返回值的唯一方式。

参考文献

[1] IEEE. **IEEE Standard for Interval Arithmetic**, June 2015.

[2] ISO/IEC. **ISO/IEC 9899:1999**, May 2005. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf.

[3] Michael Kerrisk. **The Linux programming interface**. No Starch Press, 2010.

[4] Stephen Prata. **C++ Primer Plus 中文版**. C 和 C++ 实务精选. 人民邮电出版社, 6th edition, June 2012.