

# ACM-ICPC 培训资料汇编 (1)

## 新生常见程序问题与操作指南

(版本号 1.0.0)

哈尔滨理工大学 ACM-ICPC 集训队

2012 年 12 月

# 序

2012 年 5 月，哈尔滨理工大学承办了 ACM-ICPC 黑龙江省第七届大学生程序设计竞赛。做为本次竞赛的主要组织者，我还是很在意本校学生是否能在此次竞赛中取得较好成绩，毕竟这也是学校的脸面。因此，当 2011 年 10 月确定学校承办本届竞赛后，我就给齐达拉图同学很大压力，希望他能认真训练参赛学生，严格要求受训队员。当然，齐达拉图同学半年多的工作还是很有成效，不仅带着黄李龙、姜喜鹏、程宪庆、卢俊达等队员开发了我校的 OJ 主站和竞赛现场版 OJ，还集体带出了几个比较像样的新队员，使得今年省赛我校取得了很好的成绩（当然，也承蒙哈工大和哈工程关照，没有派出全部大牛来参赛）。

在 2011 年 9 月之前，我对 ACM-ICPC 关心甚少。但是，我注意到我校队员学习、训练没有统一的资料，也没有按照竞赛所需知识体系全面系统培训新队员。2011-2012 年度的学生教练们做了一个较详细的培训计划，每周都会给 2011 级新队员上课，也会对老队员进行训练，辛辛苦苦忙活了一年——但是这些知识是根据他们个人所掌握情况来给新生讲解的，新生也是杂七杂八看些资料和做题。在培训的规范性上欠缺很多，当然这个责任不在学生教练。2011 年 9 月，我曾给老队员提出编写培训资料这个任务，一是老队员人数少，有的还要去百度等企业实习；二是老队员要开发、改造 OJ；三是培训新队员也很耗费精力，因此这项工作虽很重要，但却不是那时最迫切的事情，只好被搁置下来。

2012 年 8 月底，2012 级新生满怀梦想和憧憬来到学校，部分同学也被 ACM-ICPC 深深吸引。面对这个新群体的培训，如何提高效率和质量这个老问题又浮现出来。市面现在已经有了各种各样的 ACM-ICPC 培训教材，主要算法和解题思路都有了广泛深入的分析和讨论。同时，互联网博客、BBS 等中也隐藏着诸多大牛对某些算法的精彩论述和参赛感悟。我想，做一个资料汇编，采撷各家言论之精要，对新生学习应该会有较大帮助，至少一可以减少他们上网盲目搜索的时间，二可以给他们构造一个相对完整的知识体系。

感谢 ACM-ICPC 先辈们作出的杰出工作和贡献，使得我们这些后继者们可以站在巨人的肩膀上前行。

感谢校集训队各位队员的无私、真诚和抱负的崇高使命感、责任感，能够任劳任怨、以苦为乐的做好这件我校的开创性工作。

唐远新

2012 年 10 月

# 编写说明

本资料为哈尔滨理工大学 ACM-ICPC 集训队自编自用的内部资料，不作为商业销售目的，也不用于商业培训，因此请各参与学习的同学不要外传。

本册内容由杨和禹、姜喜鹏等分别编写和校核。

本分册内容部分采编自各 OJ、互联网和部分书籍。在此，对所有引用文献和试题的原作者表示诚挚的谢意！

由于时间仓促，本资料难免存在表述不当和错误之处，格式也不是很规范，请各位同学对发现的错误或不当之处向[acm@hrbust.edu.cn](mailto:acm@hrbust.edu.cn)邮箱反馈，以便尽快完善本文档。在此对各位同学的积极参与表示感谢！

哈尔滨理工大学在线评测系统（Hrbust-OJ）网址：<http://acm.hrbust.edu.cn>，欢迎各位同学积极参与AC。

国内部分知名 OJ：

杭州电子科技大学：<http://acm.hdu.edu.cn>

北京大学：<http://poj.org>

浙江大学：<http://acm.zju.edu.cn>

以下百度空间列出了比较全的国内外知名 OJ：

[http://hi.baidu.com/leo\\_xxx/item/6719a5ffe25755713c198b50](http://hi.baidu.com/leo_xxx/item/6719a5ffe25755713c198b50)

哈尔滨理工大学 ACM-ICPC 集训队

2012 年 12 月

# 目 录

第一章 认识ACM与集成开发环境Code::blocks.....	6
1.1 认识ACM.....	6
1.2 了解集成开发环境Code::blocks.....	6
1.3 获取并安装Code::blocks.....	7
1.4 使用Code::blocks建立C/C++源文件.....	7
1.5 代码的编译.....	10
1.6 在Code::blocks中快速复制粘贴代码.....	10
1.7 Code::blocks使用问题.....	10
1.8 本章总结.....	12
第二章 ACM在线评测系统的使用说明.....	13
2.1 注册一个在线评测系统帐号.....	13
2.2 在线评测系统的FAQ.....	13
2.3 在线提交问题.....	15
2.4 在线评测系统的其它功能.....	16
2.5 本章总结.....	17
第三章 代码书写规范与标准.....	18
3.1 代码书写规范的重要性.....	18
3.2 代码的缩进格式与空格.....	19
3.3 函数及变量的命名规则.....	22
3.4 对与代码书写规范的总结.....	22
3.5 本章总结.....	23
第四章 C/C++初步基础的常见错误与ACM例题分析.....	24
4.1 标识符与主函数的问题.....	24
4.2 符号问题.....	24
4.3 变量的赋值与初始化.....	25
4.4 声明、定义与重定义问题.....	25
4.5 数据范围问题.....	25
4.6 运算符优先级问题.....	26
4.7 条件运算符.....	26
4.8 表达式的表示.....	26
4.9 除法问题.....	27
4.10 空语句问题.....	27
4.11 本章总结.....	28
第五章 ACM中的输入输出.....	29
5.1 输入输出函数与ACM.....	29
5.2 格式控制符.....	29
5.3 取址运算符与输入输出函数.....	29
5.4 单个字符的读入与getchar函数.....	30
5.5 gets函数与字符串.....	31
5.6 本章总结.....	31
第六章 条件结构与循环结构.....	33
6.1 if条件语句的使用.....	33
6.2 if与else配对问题.....	33
6.3 判断变量与零值或其它值的关系.....	34
6.4 循环语句.....	35
6.5 循环语句的效率问题.....	35
6.6 break与continue语句.....	36
6.7 分析死循环问题.....	36

6.8 本章总结.....	36
第七章 函数的使用.....	36
7.1 函数的定义与声明.....	37
7.2 函数体的书写规范.....	37
7.3 主函数的问题.....	37
7.4 函数的调用.....	38
7.5 函数的形参问题.....	38
7.6 本章总结.....	40
第八章 数组与C/C++编程.....	41
8.1 数组的性质.....	41
8.2 数组的定义及初始化问题.....	41
8.3 数组的下标问题.....	41
8.4 数组大小的讨论.....	41
第九章 字符串及字符串的处理.....	43
9.1 字符串与数组.....	43
9.2 字符串的处理.....	43
第十章 有关指针的讨论.....	44
10.1 指针的初始化问题.....	44
10.2 指针与动态内存申请.....	44
10.3 数组与指针.....	45
10.4 字符串与指针.....	45
10.5 八到十章的总结.....	46
第十一章 学习路线.....	47
第十二章 ACM新手常见基本问题集锦.....	49
12.1 评测结果与评测环境.....	49
12.2 C语言常见编码问题与技巧.....	53
第十三章 Linux使用简介.....	55
13.1 Linux简介.....	55
13.2 Linux下常用命令简介.....	61
13.3 编写程序.....	66

# 第一章 认识 ACM 与集成开发环境 Code::blocks

## 1.1 认识 ACM

ACM 不仅仅是一个程序竞赛，因为 ACM 竞赛是在考察一个程序员的综合素质。首先 ACM 对参赛选手的很多知识方面都会进行考察，其比赛题目涉及到的知识点范围非常的广。从一个程序员的基础方面（如程序语言的基础、数据结构）到一些较为复杂的算法（如图论、数论、动态规划、计算几何、组合数学等）都有考察。因此，一个 ACM 的大牛一定是基础扎实，在各方面的知识都有一定的了解而且还会有自己的强项。而经过 ACM 竞赛考验过的程序员无疑是众多的程序员中的佼佼者，当你坚持到最后就会发现自己的编程水平确实比一般的程序员高出不少。

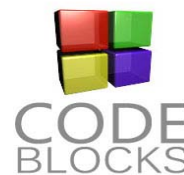
ACM 竞赛中不但是要考察每个选手的个人能力，而且还对团队合作很重视。比赛中，每个队伍必须由三个人组成，但是三个人只能使用一台计算机在有限的时间内做题。这就需要三个人之间的配合，三个人怎么样才能做到发挥出更好的效果。如果三个配合的不够默契，就很有可能造成队伍的做题效率低下等问题。因此 ACM 也是一个考验团队合作的比赛。

ACM 对同学们能力的提高是非常有帮助的，这不会局限于一个人拿到了多少各级竞赛的奖励。它对一个程序员的影响是深远的，如果一个程序员在专心的投入 ACM 的话，那么几年以后他的编程水平和算法的功底是大多数同行无法相比的，同时编出的程序质量也普遍比较高，这样的人往往会在某些技术方面成为大牛，而且各个公司都会非常欢迎这些算法功底深厚的程序员。对于那些打算考验的朋友们来说，把适当的经历投入在 ACM 上也是有很多益处的。ACM 对程序员代码编写能力的提高会有很大的帮助，往往一些名校的研究生复试题目也可以见到一些 ACM 的相对基础的题目。因此即使是打算考验的朋友们，在 ACM 上投入一定的精力也是一个正确的选择。

每年，哈尔滨理工大学将举办一次校赛，校赛的名次将作为参加省赛的参考，而之后的东北区赛就需要参赛队员们的继续努力。要想参加 ACM ICPC 的亚洲区赛，那么就需要付出更多的努力了。我们之所以对 ACM 如此的重视，不光是因为比赛的成绩，更重要的是提高同学们自身能力，这样在以后的学习和工作之中才能成为一个佼佼者。最后，哈尔滨理工大学 ACM 集训队队员祝同学们能在 ACM 的学习过程中找到乐趣。

## 1.2 了解集成开发环境 Code::blocks

ACM 竞赛中，最常用的集成开发环境就是 Code::blocks 了。这款集成开发环境体积小，占用系统资源也相对较小，而且非常容易上手通常。各级别 ACM 竞赛中往往会提供 Code::blocks。由于比赛中，ACM 官方并不承认 VC++ 而是承认 GCC、G++ 和 JAVA。因此对于我们这些 C/C++ 程序员来说通常会选择使用 Code::blocks，默认的编译器一般会选择 GCC 编译器。为了便于统一解答问题，我们强烈建议同学们使用 Code::blocks 这款集成开发环境。《哈尔滨理工大学 ACM 入门指南》中的所有样例代码都是在 Code::blocks 集成环境并使用 GCC 编译器编译的。



对于曾经非常经典的 Visual C++ 6.0（简称 VC 6.0）我们不建议使用，因为 VC 6.0 与 C 语言和 C++ 标准实际上有很大差距，而且默许了很多不规范的书写。因此长期使用 VC 6.0 很容易养成不良的编程习惯，不良的编程习惯对于一个程序员的负面影响是非常大的。因此使用 Code::blocks 是一个不错的选择。

## 1.3 获取并安装 Code::blocks

### 1.3.1 获取 Code::blocks

建议同学们从”2012 级 ACM 爱好者“的群共享中获取 Code::blocks 的安装包（安装包大小约 70MB），我们保证了群共享中安装文件的可靠性。而从其它途径下载 Code::blocks 往往同学们可能会下载到没有包含编译器的 Code::blocks。另外如果从他人的电脑里直接拷贝 Code::blocks 安装后的文件也可能出现一些预想不到的问题。

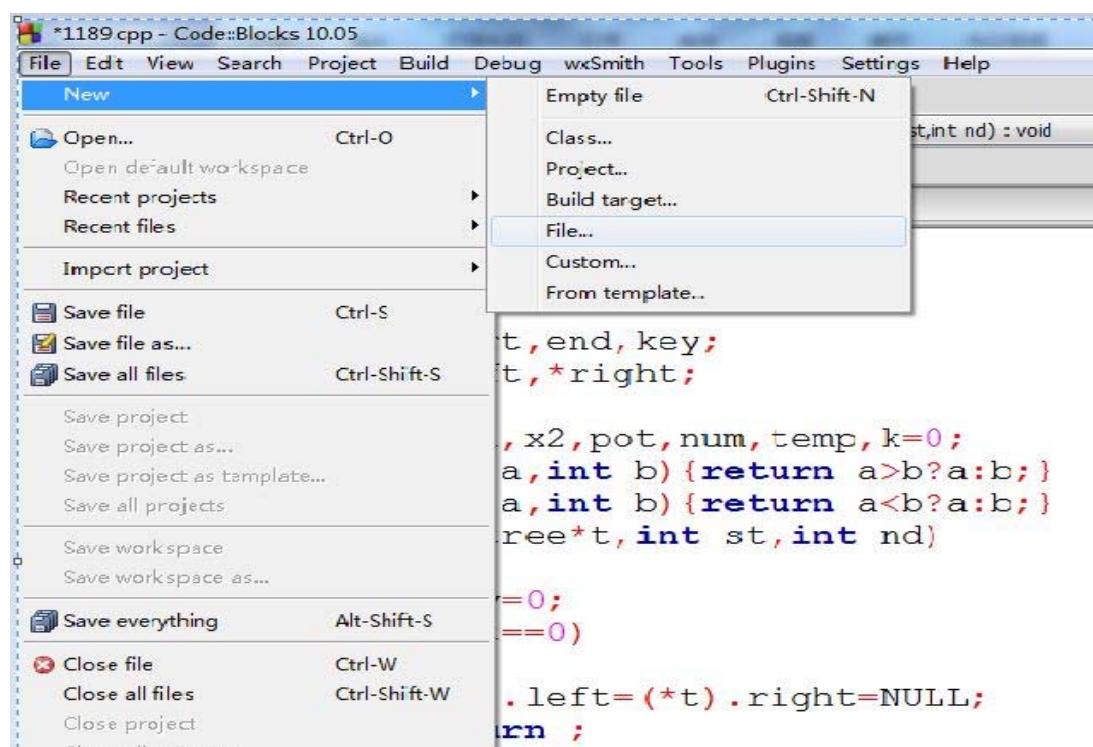
### 1.3.2 安装 Code::blocks

双击安装包，按照提示进行安装，建议不要将 Code::blocks 安装到 C 盘。在安装结束后，第一次运行 Code::blocks 时会弹出一些窗口，我们只需点击“Yes”或“Close”按钮即可，不必理会那些看不懂的英文信息。在英文版的 Code::blocks 的使用过程中，你需要了解的信息往往是英文提示中自己能看懂的部分，而一些根本不知道是什么意思的单词多数情况下是我们可能不需要了解的。如果对英文版的软件实在难以适应，群共享中有 Code::blocks 的汉化工具。不过我们建议同学们尽量去适应英文。

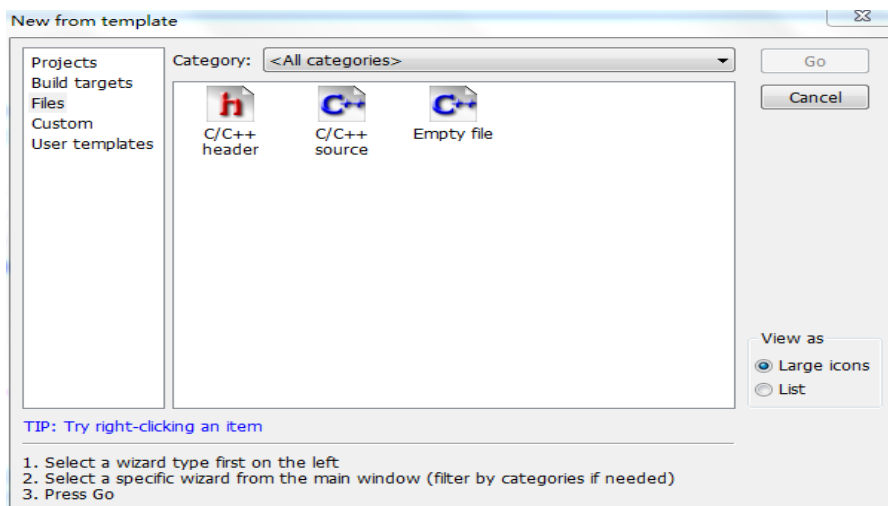
## 1.4 使用 Code::blocks 建立 C/C++源文件

尽管 Code::blocks 安装后是英文的，很多英文程序看似让人很无从下手，实际上我们只需简单的几步就可以建立 C/C++的源文件。

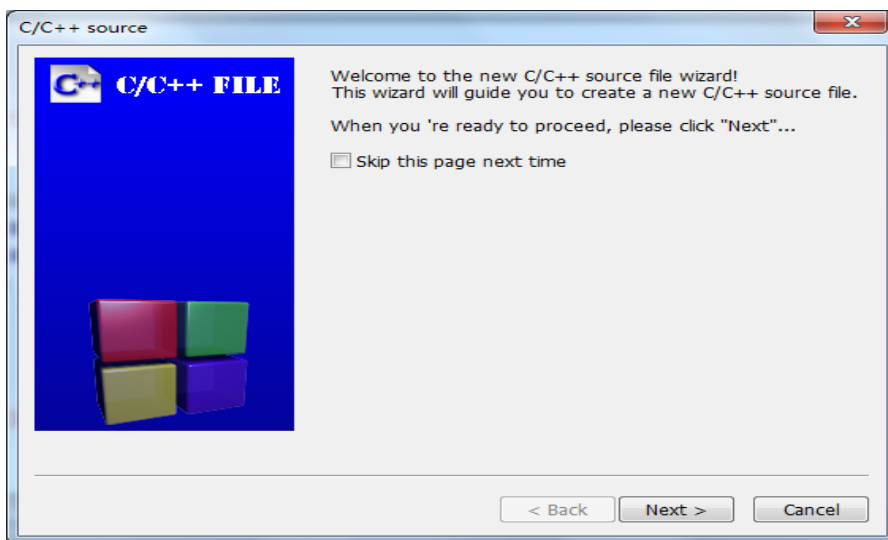
**Step1:** 运行 Code::blocks，单击菜单栏左上方的”File“，在下拉菜单中找到”New“，在”New“的子菜单中选择”File“。（如图所示）



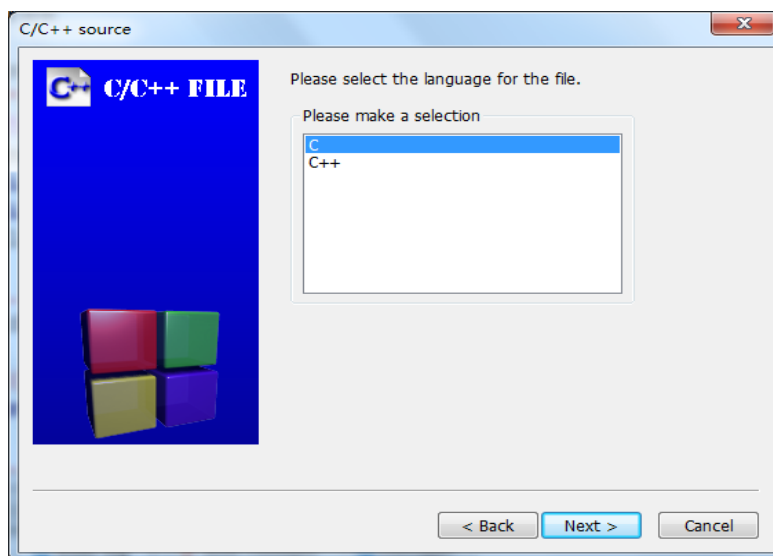
**Step2:** 在弹出的对话框中选择“C/C++ source”，并单击“Go”按钮




Step3: 勾选下图对话框中的复选框，单击“Next”

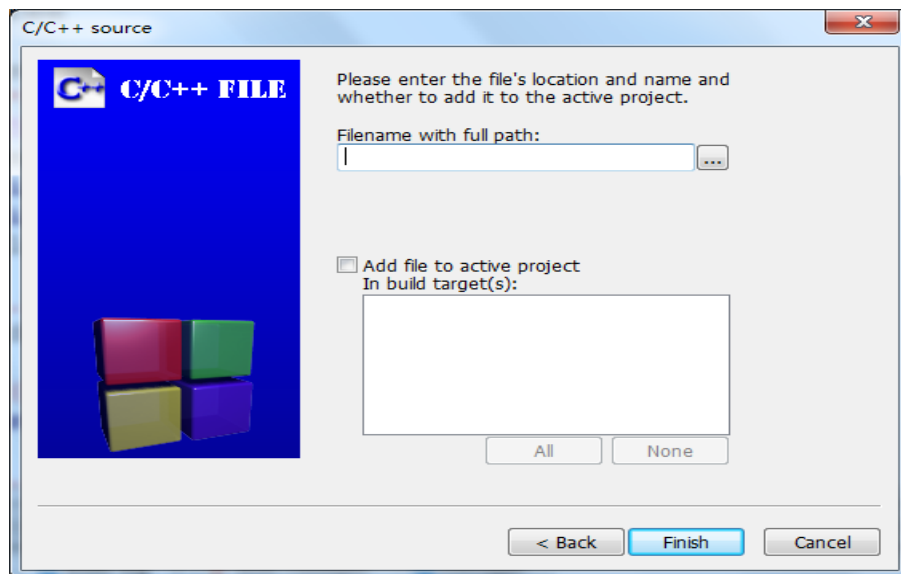


Step4: 下图对话框中，如果使用 C 语言选择 C，如果使用 C++编程则选择 C++，选择编程语言后单击“Next”

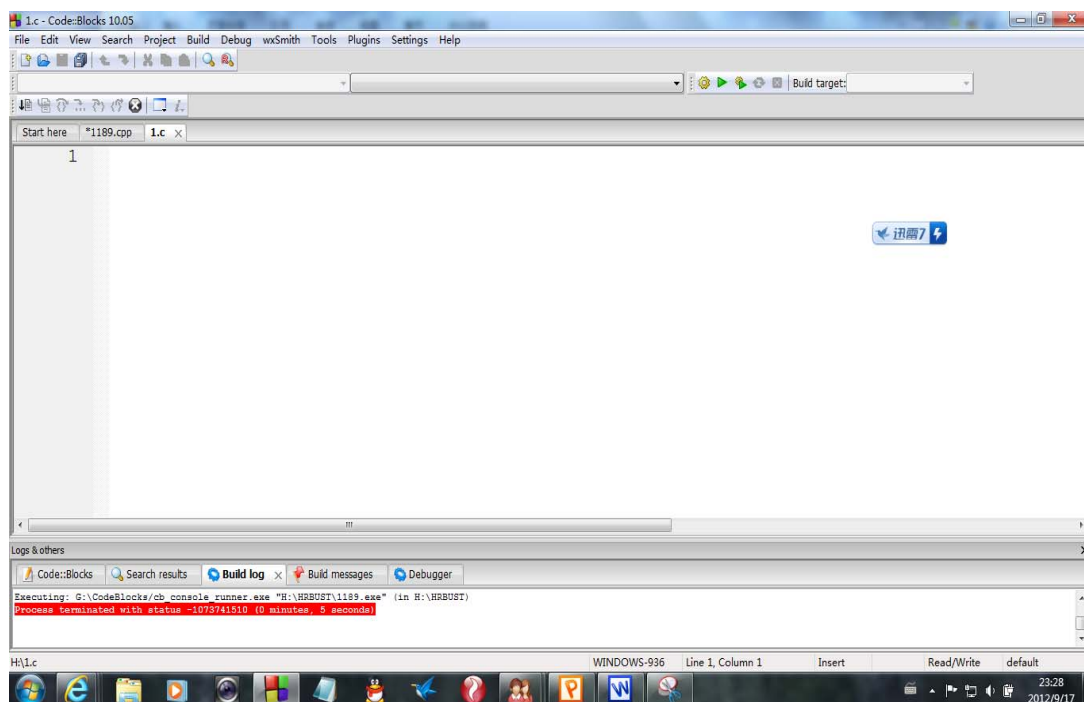




**Step5:** 下图的对话框中，选择一个保存即将建立文件的位置和文件名，单击  按钮选择文件保存路径后，单击“Finish”。




进入 Code::blocks 编译环境（如下图）

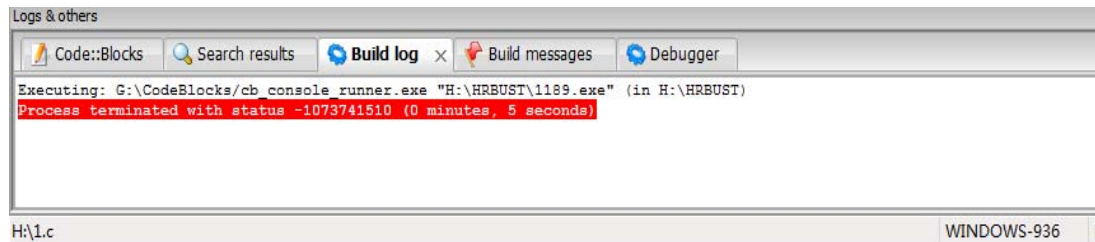


## 1.5 代码的编译

我们写完代码后是要把代码编译后，才能得到一个可执行的程序，因此写完代码后我们要编译代码，那么哪些按钮具有该功能呢？下面我们介绍一下与编译和运行有关的几个按钮（描述顺序从左至右）：

与编译有关的几个按钮：

第一个按钮是编译按钮，如有关编译的信息会在下图的区域中显示，如果编译成功该区域中会用蓝色字体提示“0 errors, 0 warnings”，遇到编译错误时该区域中会提示编译失败，并提示可能出现的错误数目和可能造成编译错误的信息，此时程序员应该优先查看一下该区域给出的编译错误的提示。



第二个按钮是执行编译后的程序，当然，如果你之前未编译过你的代码，那么单击该按钮时编译环境会提示你代码尚未编译。

第三个按钮是编译并执行按钮，如果你希望编译你的代码并立即看到程序的执行那么可以单击此按钮。

## 1.6 在 Code::blocks 中快速复制粘贴代码

使用 Code::blocks 时，往往需要复制和粘贴代码。通过单击鼠标右键弹出菜单中寻找“复制/粘贴”选项显然是非常麻烦的，不过我们可以通过键盘进行对代码快速的选择、复制和粘贴。下面来介绍一些按键组合来实现这些功能：

- 1) Ctrl+A：全选，这样就不用费力的使用鼠标将代码全部选中了。
- 2) Ctrl+C：复制选中的代码/文本
- 3) Ctrl+V：粘贴功能

当然还有其它的按键组合，这些组合也同样能实现其它的一些功能，但是在编写代码的过程中，上面三种按键组合是最常用的。

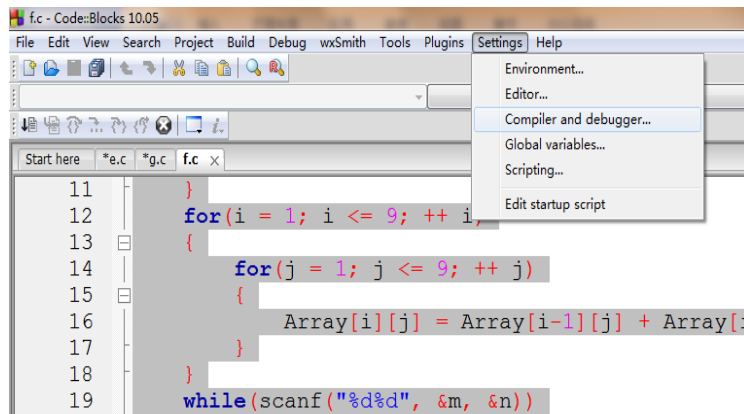
## 1.7 Code::blocks 使用问题

### 1.7.1 Code::blocks 无法编译代码

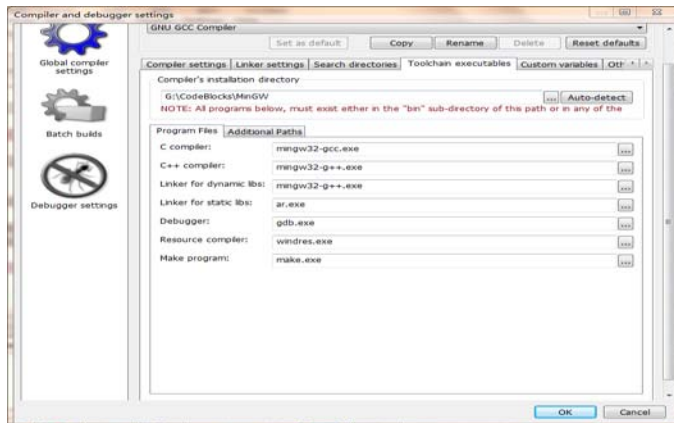
有些同学在安装 Code::blocks 后出现了无法编译代码的问题，出现这种情况很可能是在网上下载了没有编译器的 Code::blocks，这种不带编译器的安装包大小只有大约 20MB，因此不建议从网上下载这种安装包。一个带有编译器的 Code::blocks 的安装包大小大约是 70MB。我们建议从群共享里下载 Code::blocks 安装包。

如果在使用从群共享里下载的 Code::blocks 仍然无法编译代码，那么请遵从下面的步骤：

Step1: 打开 Code::blocks, 单击菜单栏中的“Setting”, 在弹出的下拉菜单中选择“Compiler and debugger”



Step2: 在弹出的对话框中找到“Toolchain executables”选项卡, 单击该选项卡, 出现下图界面:



在对话框中找到并单击“Auto-detect”按钮, 这样 Code::blocks 会自动定位编译器的位置, 定位成功后会弹出一个对话框, 单击“确定”即可。最后单击“OK”。

### 1.7.2 cannot open output file 导致无法编译的问题

假设已经建立的一个 C 语言的源文件 1000.c, 有些时候, 在编译时编译器会提示类似下面的内容:

```
ld.exe cannot open output file
```

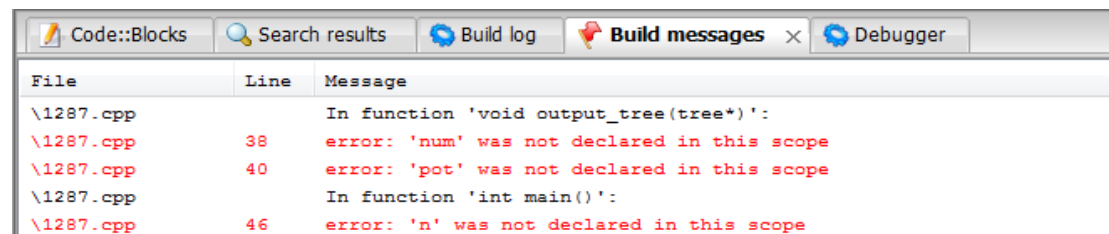
```
C:\Users\YangHeyu\Desktop\contest\contest\1000.exe Permission denied
```

出现这种情况源文件是无法正常编译的, 原因是可执行文件 1000.exe 正在后台运行。出现这种情况请打开任务管理器, 将 1000.exe 这个进程结束。这样就可以正常的编译文件了。

### 1.7.3 编译错误

编译问题是一个程序员经常能遇到的问题, 就是一个从业多年的程序员也不能保证自己一段代码敲下来没有任何的语法错误。如果遇到编译错误我们应该立即将其排除。因为出现编译错误, 代码是不能被编译为可执行文件的。这里介绍一下简单的寻找编译错误的方法。

对于一些短的代码我们可以逐行检查（这种方法一般会费些时间），我们可以查看编译环境下方的这个区域中提示的信息（如图）



File	Line	Message
\1287.cpp		In function 'void output_tree(tree*)':
\1287.cpp	38	error: 'num' was not declared in this scope
\1287.cpp	40	error: 'pot' was not declared in this scope
\1287.cpp		In function 'int main()':
\1287.cpp	46	error: 'n' was not declared in this scope

“Line”这一列提示代码中哪一行有可能出现编译错误（代码的左边标注了代码的行号），“Message”这列提示可能对对应行出现错误的原因。因此，可以从提示的那几行中分别查看提示可能出现的导致编译错误的问题是否存在（尽管提示是英文的，但是有用的提示往往是所有人都能看懂的那些单词。因此不要惧怕英文提示）。如果按照提示经过反复检查后仍然无法通过编译，那么建议从第一行开始一行行检查。因为一行出现编译错误，很可能让编译器认为其它行的代码出现编译错误。

#### 1.7.4 代码调试问题

一般来说，我们要使用编译环境提供的调试工具来对代码进行调试是需要建立一个工程的。而建立一个工程会生成一些现阶段对我们没有用处的文件。往往编译环境提供的调试功能对于我们来说使用并不是很方便。通常调试题目程序时普遍采用将关键值或运算中间过程数据输出的方法，这样做更加的直观、灵活。因此现阶段对代码的调试可以采用输出关键值或运算中间过程数据的方法。

### 1.8 本章总结

通过本章阅读本章的内容同学们一概了解集成开发环境 Code::blocks 的使用，同学们需要尽快的使用这款软件。其实 Code::blocks 是非常简单易用的，以前习惯使用其它集成开发环境的同学最好将以前书写的代码也是用 Code::blocks 这款软件重写的书写一遍，这样对熟悉 Code::blocks 是非常有帮助的。

建议使用 Code::blocks 是有原因的，使用 Code::blocks 集成开发环境和 GCC 编译器对于代码语法的规范是非常有帮助的，希望使用其它编译环境（尤其是 VC 6.0）的同学们尽快开始使用 Code::blocks。

## 第二章 ACM 在线评测系统的使用说明

对于 C/C++ 代码编写能力最好的提高方法就是去做题，同学们在电脑上练习书后的习题往往很难知道自己写的程序是否正确。同学们到在线评测系统中做题，系统会根据使用者所提交的程序用大量数据进行测试，最终反馈给使用者题目做的是否正确，这样对个人编程水平的提高是非常有帮助的。哈尔滨理工大学也为同学们提供了这样一个在线评测系统，帮助同学们提高。

我们的在线评测系统网址：<http://acm.hrbust.edu.cn>

哈尔滨理工大学在线评测系统主站：



### 2.1 注册一个在线评测系统帐号

在开始做题之前，应该先拥有一个哈尔滨理工大学在线评测系统的帐号。对于 2012 级的学生可以直接使用自己的学号来登入在线评测系统，初始的用户名与密码均为自己的学号。建议同学们使用这个帐号并实名刷题，这样方便对同学们做题情况的了解。其他同学可以单击“Register”进入注册账户界面来注册一个新账户。

### 2.2 在线评测系统的 FAQ

在正式做题之前，一定要先阅读在线评测系统的 FAQ。FAQ 中包含了一些需要使用在线评测系统时应该注意的事项。刚接触 ACM 的朋友遇到的问题往往 FAQ 上都已经给出了详细的说明，因此强烈建议同学们在做题之前一定要认真阅读 FAQ，下面我们给出了哈尔滨理工大学在线评测系统的 FAQ，供同学们参考：

哈尔滨理工大学在线评测系统提供编译器：

GCC（可编译 C 语言程序）

G++（可编译 C++ 程序）

JAVA（可编译 JAVA 程序）

提示：所有大学的在线评测系统和 ACM ICPC 一定会提供这三种编译器

编写程序应符合 ANSI C 规范：

1) ANSI 标准要求程序的主函数的类型必须为 `int`，评测系统的 GCC 和 G++ 编译器将不接受 `void` 类型的主函数。否则无法通过编译

2) 请尽量避免在这样的写法：`for( int i=0; i < n; ++i)`，建议在 for 循环外定义“i”



- 3) itoa 不是一个 ANSI 函数
- 4) 字符串处理函数不是 ANSI 函数
- 5) 当你需要使用 sqrt 函数时, 应保证其参数是 double 类型, 否则请强制转换。
- 6) 当需要使用 64 位的整数时, 64 位整数应该用 long long 表示, 其相应的格式控制符为 %lld。而无符号的 64 位整型用 unsigned long long 表示, 格式控制符为 %llu。

### 在线评测系统的评测结果:

当你提交的程序被在线评测系统评判完毕后, 你可以在“Status”页看到评判结果。常见的在线评测系统将评判结果分为如下几类:

1) **Accepted** : 表明你的程序正常运行并退出, 且输出的结果完全满足题意, 通过了题目全部的测试数据。

2) **Wrong Answer** : 你的程序正常运行并退出, 但是输出结果有错误, 无法通过全部的测试数据, 所以被判为 **Wrong Answer**。

3) **Presentation Error** : 你的程序对于全部测试数据输出的结果是正确的, 但是输出格式与题意不符。

注意: 在线评测系统可能很难准确判断这种错误, 很多情况该类错误被判为 **Wrong Answer**。

4) **Compile Error** : 你的程序没有通过编译。你可以点击文字上的链接, 查看详细的出错信息, 对照此信息, 可以找出出错原因。

5) **Judging** : 在线评测系统正在运行你的程序进行测试, 请稍候。

6) **Time Limit Exceeded** : 你的程序运行的时间超过了该题规定的最大时间, 你的程序被在线评测系统强行终止。使用 C++ 的 cin 和 cout 控制输入输出可能会超时, 建议使用 C 语言中的输入输出函数。

注意: TLE 并不能说明你的程序的运行结果是对还是错, 只能说明你的程序用了太多的时间。

7) **Memory Limit Exceeded** : 你的程序运行时使用的内存, 超过了该题规定的最大限制, 或者你的程序申请内存失败, 你的程序将被 Online Judge 强行终止。

注意: MLE 并不能说明你的程序的运行结果是对还是错, 只能说明你的程序用了或者申请了太多的内存。

提示: 如果程序的运行时间或内存占用大小接近限定的上限, 那么建议查找或思考更快捷的算法。代码的长短与程序运行效率没有关系。

8) **Restricted Function** : 你的程序运行时使用我们不允许使用的调用, 将会得到此错误, 诸如文件操作等相关函数。**请特别注意: system("pause"); 也会导致此错误。**

9) **Runtime Error** : 你的程序在运行时出现了错误, 整型数据除 0、数组越界或者指针使用不当均可能导致该问题。

10) **Internal Judge Error**: 系统发生了错误。由于异常因素导致系统没有正常运作。我们尽力保证系统的稳定运行, 但如您遇此情况, 请及时联系管理员。

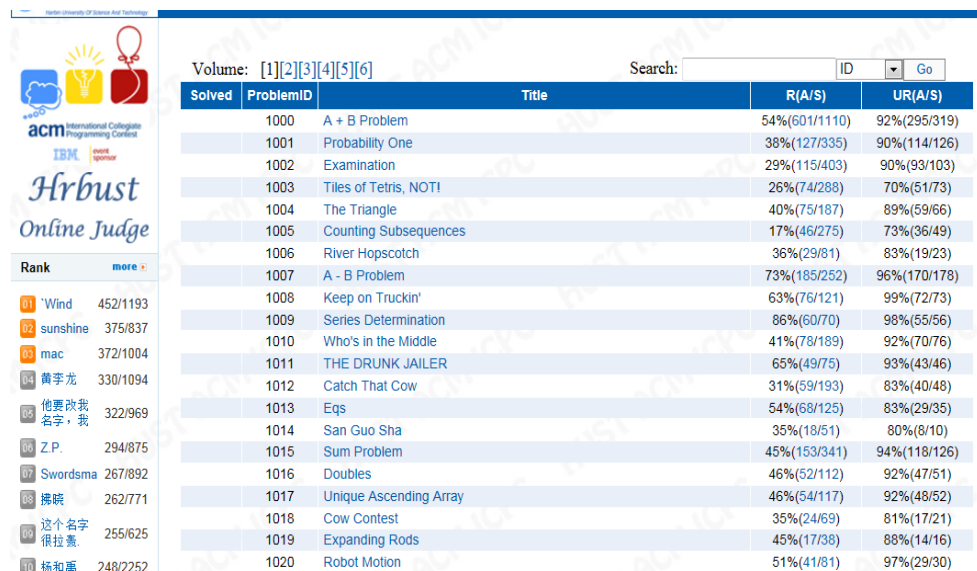
11) **waiting** : 你的代码正在等待评测系统的处理, 但是如果长时间处于该状态, 请联系管理员。

**如果题目包含多组测试数据, 应该在何时输出结果?**

在线评测系统中，你的程序的输入和输出是相互独立的，因此，每当处理完一组测试数据，就应当按题目要求进行相应的输出操作。而不必将所有结果储存起来一起输出。

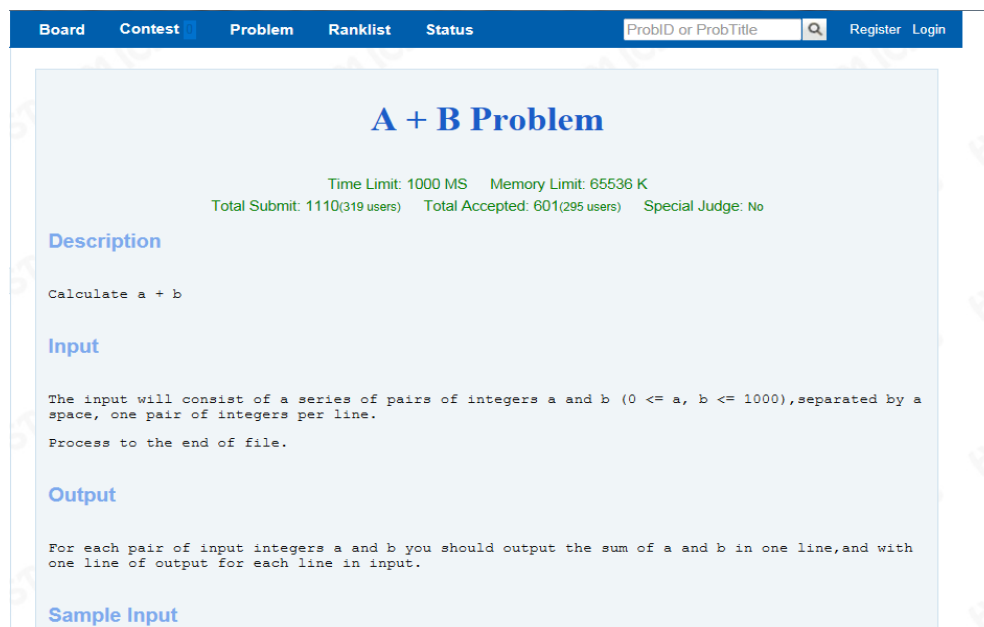
## 2.3 在线提交问题

1、单击“Problem”，进入题目列表网页（如下图）



Solved	ProblemID	Title	R(A/S)	UR(A/S)
	1000	A + B Problem	54%(601/1110)	92%(295/319)
	1001	Probability One	38%(127/335)	90%(114/126)
	1002	Examination	29%(115/403)	90%(93/103)
	1003	Tiles of Tetris, NOT!	26%(74/288)	70%(51/73)
	1004	The Triangle	40%(75/187)	89%(59/66)
	1005	Counting Subsequences	17%(46/275)	73%(36/49)
	1006	River Hopscotch	36%(29/81)	83%(19/23)
	1007	A - B Problem	73%(185/252)	96%(170/178)
	1008	Keep on Truckin'	63%(76/121)	99%(72/73)
	1009	Series Determination	86%(60/70)	98%(55/56)
	1010	Who's in the Middle	41%(78/189)	92%(70/76)
	1011	THE DRUNK JAILER	65%(49/75)	93%(43/46)
	1012	Catch That Cow	31%(59/193)	83%(40/48)
	1013	Eqs	54%(68/125)	83%(29/35)
	1014	San Guo Sha	35%(18/51)	80%(8/10)
	1015	Sum Problem	45%(153/341)	94%(118/126)
	1016	Doubles	46%(52/112)	92%(47/51)
	1017	Unique Ascending Array	46%(54/117)	92%(48/52)
	1018	Cow Contest	35%(24/69)	81%(17/21)
	1019	Expanding Rods	45%(17/38)	88%(14/16)
	1020	Robot Motion	51%(41/81)	97%(29/30)

2、我们以题库中的 1000 题为例



**A + B Problem**

Time Limit: 1000 MS    Memory Limit: 65536 K  
 Total Submit: 1110(319 users)    Total Accepted: 601(295 users)    Special Judge: No

**Description**

Calculate  $a + b$

**Input**

The input will consist of a series of pairs of integers  $a$  and  $b$  ( $0 \leq a, b \leq 1000$ ), separated by a space, one pair of integers per line.  
 Process to the end of file.

**Output**

For each pair of input integers  $a$  and  $b$  you should output the sum of  $a$  and  $b$  in one line, and with one line of output for each line in input.

**Sample Input**

在线评测系统中题目有中文题目也有英文题目，初期同学们可能不适应英文题目（可借助有道词典翻译），但是随着时间的推移应该逐渐习惯英文题。一些著名的在线评测系统上的题目都是英文的，而且 ACM 比赛的题目官方语言也是英文。我们可以从一些相对简单的题目做起。例如这道 A+B 问题，这道题的描述下方展示了一些要求多组数据输入的题目输入函数书写方式：


```
#include<stdio.h>
```

```
int main(void)
{
    int A, B;
    while (scanf("%d%d", &A, &B) != EOF)
    {
        printf("%d\n", A + B);
    }
    return 0;
}
```

这种输入方式适用于多组输入而且题目未明确具体有几组输入数据的题目，这种输入方式要记住，如果不写“!=EOF”，那么就会导致评测系统认为你提交的代码是“Wrong Answer”。

### 3、提交代码

我们在编译环境中将代码调试好之后，单击“Submit”按钮，进入提交代码页面。建议先将代码调试好并粘贴过去，不建议在线输入代码。另外就是选择语言了，（如右图）语言选择使用 C 语言的同学建议选择 GCC，而使用 C++ 编程的同学则需要选择 G++。单击“Submit”代码就会被提交，评测系统会直接跳转到“Status”，在这个页面会显示提交代码的评测结果。



Language: GCC ▼

### 4、Statistic 选项

我们在“Submit”按钮的右边可以看到“Statistic”，单击“Statistic”会进入新的页面，页面上显示了这道题的提交状况和 AC 用户的代码的运行时间、内存占用和代码长度信息。如果这道题已经 AC 了，那么可以尝试的优化自己的代码，让代码比以前更加优秀。

### 5、Discuss 选项

“Discuss”位于“Statistic”的右边，当做题时遇到不解的地方或者对题目、数据等有疑问时可以在这道题的“Discuss”发布相关信息。

## 2.4 在线评测系统的其它功能

### 1、Board 功能

用户登入后，可以在 Board 中发布一些有关在线评测系统的疑问，这样我们能最快的对评测系统现阶段存在的问题进行及时的修正，另外也可以发布一些有助于 ACM 学习的信息。

### 2、Ranklist 功能

Ranklist 是对在在线评测系统中用户的一个排名，AC 的题目越多，排名就越靠前。当 AC 的题目数目相同时，提交代码次数较少的用户排位会相对靠前。在刷题过程之中，看着自己的排名在 Ranklist 中逐步上升，这也是一个 ACMer 在做题过程中的一大乐趣。Ranklist 的存在在某种程度来说充分的调动了同学们的兴趣，让同学们有做题的欲望。

### 3、contest 功能

在线评测系统的 contest 功能可以举办一场限时的比赛，当管理员新添加一场比赛，会在 contest 中显示出来。注意，比赛过程中，用户无权查看已经提交过的代码。因此建议每次比赛的代码一定要在自己的电脑中有备份。另外比赛过程中，如果对某一道题有疑问可以在 Message 上发布，管理员会尽快解决可能潜在的问题。



## 2.5 本章总结

本章介绍了在哈尔滨理工大学在线评测系统的使用，对于同学们来说不论以后是否会坚持 ACM 这条路在线评测系统的使用都应该熟练。以后的考试可能会利用在线评测系统来完成，其实已经有很多高校的计算机专业的考试使用在线评测系统来完成，所以使用评测系统刷题是必须掌握的。评测系统的使用并不难（肯定比现在使用的手机操作系统简单多了），在使用之前认真阅读本章的内容应该就可以很快上手了。其实编者在第一次做题时连 FAQ 都没看过，第一次看到 FAQ 应该是今年上半年校赛的时候，所以只要看看说明文档，多做题在线评测系统的各种功能基本上就能搞清楚了。

阅读本章之后，希望同学们能积极的到在线评测系统做题，题目的难易程度和题号没有关系。基础不大牢固的同学可以在题库中先找简单题做，这样可以更快的熟悉评测系统，编者就是之前一直在评测系统中刷水题的。第一次使用在线评测系统的同学们不妨先做一下题库中的 1000、1007、1015 和 1080 这几道题，这些题目非常适合第一次接触在线评测系统的同学们做，这 AC 些题目可以让同学们更加快速的入门。当然对于已经有一定基础的同学们就不要只顾着刷水题了，应该有计划地练习目前学过的不同类型题目，这样会更快的提高个人能力。

## 第三章 代码书写规范与标准

### 3.1 代码书写规范的重要性

代码的书写规范是非常重要的，然而代码的书写规范的重要性往往被大多数同学忽略，这里我们先简单的论述一下代码书写规范的重要性。

目前来说，由于受到知识掌握的限制，刚刚接触 C/C++ 的朋友们通常只能通过一些简单而且代码量很小的程序进行练习。由于代码较短，涉及到的变量及函数并不多，此时往往显示不出来代码规范书写的重要性。但是，随着所学知识的深入，编写的程序可能会有数百行甚至上千行，涉及到的变量和函数会多达数十个。如果此时仍然不注重代码的书写规范，那么写出的代码可读性很差，不利于错误的查找，也同样不利于团队开发的项目。如果不良的代码书写风格不尽早纠正的话，会对程序员今后的工作造成极大的负面影响。例如一个数百行的程序，如果代码书写不规范，在阅读程序时很难判断出这个程序的功能，也很可能无法判断程序到底是哪里出现了错误，如果是正在给某个公司进行项目制作，那么这样的代码会对公司和客户造成极大的损失，因此代码的书写规范问题是不容忽视的，从现在开始就应该注意这个问题。这样对今后的学习可工作才能起到一个积极的作用。

我们那哈尔滨理工大学在线评测系统中的 1000 题为例，题目的要求是多组数据输入求解 A+B 问题，首先我们展示一段书写风格不规范的一段代码：

```
#include<stdio.h>
int main(){
int a,b,c;
while(scanf("%d%d",&a,&b)!=EOF){
c=a+b;printf("%d\n",c);}}
```

上面的程序不过才短短的几行而已，却让人感到格式不清晰，阅读稍有困难。我们难以想象如果按照这样的书写格式写出的几十行甚至成百上千行的程序会是什么样。可以肯定的是这样的程序几乎无法再阅读下去。接下来我们参考一下相对规范的 C/C++ 代码书写方式：

```
#include <stdio.h>

int main(void)
{
    int a, b, c;

    while (scanf("%d%d", &a, &b) != EOF)
    {
        c = a + b;
        printf("%d\n", c);
    }

    return 0;
}
```

这段相对规范的代码读起来感觉就比较清晰，一段良好书写格式的代码给人的感觉是舒畅的，这样的代码便于阅读，如果程序发生错误时查找错误也相对容易些。要充分的认识到代码书写规范的重要性，并及早就开始规范自身的代码书写。

不幸的是国内很多高校和课本忽视了良好的代码书写规范的重要性，导致了很多学生在书写代码时格式混乱，代码难以阅读。这对今后学生们的发展不利，我们应该及早的认识到这一点，在学习 C/C++ 之初就要纠正该问题。

## 3.2 代码的缩进格式与空格

代码的书写过程中，我们需要注意代码的缩进格式与空格的问题。缩进格式与空格直接影响一段代码的书写风格，尽管 C/C++ 对代码书写格式没有做出明确的要求，但是我们在书写代码时要做到缩进格式与空格的合理安排，这样写出的代码会非常方便阅读和查找错误。

### 3.2.1 空行问题

良好的代码书写格式对于程序的阅读和改错都是非常有用的，下面我们先来谈一谈代码中的空行问题。通常普遍被程序员们接受的空行使用规则如下：

规则 1：在每个类（C++）、结构体声明和函数定义结束后要加一个空行。

规则 2：在函数体内，逻辑关系紧密的语句之间不加空行，但是其他没有紧密逻辑关系的地方要加空行。

良好的空行使用方式（样例）：

```
#include <stdio.h>
```

```
int main(void)
{
    int a, b;

    scanf("%d%d", &a, &b);

    if (a > b)
    {
        printf("max_number = %d\n", a);
    }
    else
    {
        printf("max_number = %d\n", b);
    }

    return 0;
}
```

要注意空行的使用方式，合理空行使用会增强代码的可读性，因此建议从学习 C/C++ 之初就要对代码行中的空行问题。在平时写代码的时候就要对这点加以注意。这样才能养成良好的代码书写习惯。

### 3.2.2 代码行问题

对于代码行的问题同样也需要重视，但是往往这个问题在 C/C++ 的学习中不得到重视，下面是关于代码行书写中的两个被普遍认可的规则。

规则 1：每一行代码只做一件事情，如只定义一个变量，或只写一条语句。这样的代码容易阅读，并且方便于写注释。

规则 2：if、for、while、do 等语句自占一行，执行语句不得紧跟其后。不论执行语句有多少都要加 {}。这样可以防止书写失误。

当然，每行仅仅定义一个变量可能对于很多人都会非常的不习惯。比如说为了定义一个整型变量‘i’就单独消耗一行，因此一行只定义一个变量可以慢慢的适应或是暂且当做一条建议。但是对于任意的一行代码，不要同时写下多条语句，这样在大工程中会显得代码一场混乱。因此一定要做到一行代码只书写一条语句。

对于第二条规则来说，其实作为该文档的编者，表示也不大习惯这样做。例如 if 语句后面只有一条 break 语句时很自然的就直接将 break 语句接到 if 的后面去了。尽管看起来规则 2 的规定有些不必要，但是经过仔细思考后觉得这样做确实是有道理的。如果从学习编程的开始就注意这些书写问题，看到文档中说明的规则就不会肝胆不习惯了。

### 3.2.3 代码行内的空格问题

代码行之中的空格问题也是不能被忽视的，代码行内加入适当的空格会让代码更适合阅读。如果代码行内不加入空格，如果遇到某航代码包含较多的运算符或者其他内容时会造成程序员阅读代码的不便。对于代码行中的空格问题我们必须掌握。下面是在一个文献中查到的一些代码行中的空格使用规则，同学们务必牢记，并体现在实际代码书写中。

规则 1：关键字之后一定要留空格。const、virtual（C++中的关键字）、inline、case 等关键字之后至少要留一个空格，否则无法辨析关键字。而 if、for、while 等关键字之后应留一个空格再输入左括号以突出关键字。

规则 2：函数名之后不要留空格，紧跟左括号以与关键字区别。

规则 3：左括号向后紧跟，右括号、逗号、分号向前紧跟，紧跟处不留空格。

规则 4：逗号之后要留空格，如 Function(x, y, z)。如果分号不是一行的结束符号，其后要留空格，如 for (initialization; condition; update)。

规则 5：二元运算符的前后要加空格，但是一元运算符的前后不加空格。

规则 6：下标运算符、成员运算符前后不加空格。

规则 7：对于表达式比较长的 for 语句和 if 语句，为了紧凑起见可以适当地去掉一些空格，如：for (i=0; i<10; i++)和 if ((a<=b) && (c<=d))

下面的一段求解 n 个数中最大值的代码的书写基本符合上述几个规则的要求，希望同学们参考一下：

```
#include <stdio.h>

int Max(int a, int b)
{
    return a > b ? a : b;
}

int main(void)
{
    int i;
    int n;
    int max_number;
    int array[100];

    while (scanf("%d", &n) != EOF)
    {
```

```

        for (i=0; i<n; ++i)
        {
            scanf("%d", &array[i]);
        }

        max_number = array[0];

        for (i=1; i<n; ++i)
        {
            max_number = Max(max_number, array[i]);
        }

        printf("max_number = %d\n", max_number);
    }

    return 0;
}

```

### 3.2.4 大括号的对齐问题

大括号的对其问题同样影响了代码的书写风格，在代码中大括号的书写应该格式统一、清晰，对于大括号的对齐方式通常认可的规则有限面两条：

规则 1：程序的分界符 ‘{’ 和 ‘}’ 应独占一行并且位于同一列, 同时与引用它们的语句左对齐。

规则 2：{ } 之内的代码块在 ‘{’ 右边对齐，通常都是空四个对齐。

良好的大括号对齐方式：

```

int Max(int a, int b)
{
    return a > b ? a : b;
}

```

在积累了一定的代码书写经验后，会发现其实大括号的对齐对代码的书写风格影响是非常大的。如果没有养成良好的大括号对齐习惯，那么书写的代码往往看上去一团糟，不便于错误的检查。所以说大括号的对齐是代码书写过程中非常重要的一点，从现在开始书写代码就要注意这点，这样以后才能写出书写风格较好的代码。

### 3.2.5 长行的拆分

有时在代码书写的过程中可能会出现某一行代码代码长度很长的现象，为了方便程序阅我们需要将长行进行拆分。当然，对于长行来说并不是随便拆分的，拆分长行往往需要遵从一定的规则，否则对长行的拆分会适得其反。

多数程序员认可的一点是在长行拆分时，从优先级较低的操作符处对长行进行拆分，长行拆分后应该将拆分处的操作符置于新拆分出来的行的前端（分号除外）。新拆分出的行叫象前缩进，让代码保持整齐。

### 3.2.6 代码注释的问题

代码书写过程中注释是非常必要的，尤其是在一些代码量很大的程序中，注释是非常必要的。适当的注释可以帮助程序员更快更好的阅读代码，不过注释也不能乱写。

首先代码注释应该是对程序中的代码行起到一个提示作用，而不是写一个技术文档。因此在程序中写注释时应该在关键地方，这样就可以起到对代码的提示作用。当然了，如果代码本身的意义已经很明确就不要加注释了，这样显得有些多余。

另外我们要注意注释与代码的一致性，如果修改了被注释的代码，那么相应的代码注释就应该及时进行修改，这样可以防止一些不应该出现的问题发生。注释应该表达准确，不应该具有二义性。

对于注释的放置也是有一定的要求，注释可以置于需要注释的代码的上一行或者右边，但是不应该放置于徐注释代码的下一行，这点应该要记住。如果代码中存在循环或 if 语句的嵌套那么应该在每层 if 或者循环结束后加上注释标注其结束，这样方便阅读。

### 3.3 函数及变量的命名规则

对于函数和变量的命名规则我们要特别的注意，因为实际上函数及变量的命名是非常重要的。对于一个程序来说对函数和变量的命名都应该改使用与其功能相关的英文单词来命名，不要简单地用 fun 给一个函数命名。这样对于于一个程序来说更加让易于理解，程序的可读性大大提高。这对今后开发程序来说都很有帮助，因此现在我们就应该对函数变量的命名规则重视起来，好的编程习惯要从学习编程之初就开始养成。

就目前来讲没有一个绝对统一的命名规则。现在的任何一种命名规则都有很多程序员表示不赞同，因为各种命名规则都有其明显的优点和缺点。尽管如此，并不意味着命名规则问题不需要我们注意。同学们可以参考一下“匈牙利命名法”，可以从中吸取一些“匈牙利命名法”的优点。这样通过给函数和变量较为科学的命名，代码的质量也会有显著的提升。

### 3.4 对与代码书写规范的总结

代码的标准书写确实是一个非常重要的问题。C/C++自身并没有对代码的书写格式进行要求，因此 C/C++的书写格式是非常自由的。但是如果不按照一个良好的代码书写风格书写代码，那么对于一个数千行的代码来说可能让人难以阅读，因此规范书写是非常必要的。但是现实是我们的 C/C++教育中往往忽视了这个问题，可多教材中的代码书写风格不良，而且这个问题课堂上老师也没有强调这类问题。包括在各种考试中也不会去深究每个人的代码编写问题。因此导致很多从大学校园中走出的程序员编写的代码书写风格不好，代码难以查错。因此在这里我们重点强调了代码编写规范的重要性，对于刚刚接触 C/C++的同学们更应该重视这些，因为不良的习惯一旦养成就难改掉了。为了同学们今后能更好发展，请认真阅读并总结这章的内容，接下来我们给出一个建议性的做题代码规范推荐。

```
/*  
*    注释部分写出题号和这道题的思路和考点还有需要注意的地方  
*    注释采用这样的多行注释的形式  
*/
```

**注释行结束后书写题目代码！**

下面我们以哈尔滨理工大学在线评测系统的 1000 题为例，我们参考一下良好的做题习惯：

```
/*  
* 哈尔滨理工大学在线评测系统 1000 题  
* 注意题目需要多组输入输出
```

```

* 输出 a + b 的结果后要换行

*/

#include <stdio.h>

int main(void)
{
    int a;
    int b;

    while (scanf("%d%d", &a, &b) != EOF)
    {
        printf("%d\n", a + b);
    }

    return 0;
}

```

当然了，我们实际做的习题的难度会比这道题大，所以代码之前的注释内容也不可能是这样简单的内容，因为这部分内容主要是记录一道题的思路和考点还有需要注意的地方。尽管比赛中我们没有过多的时间去写这个，但是在平时做题时我们建议这样做。因为如果明确的分析出题目的思路和需要注意的地方后再敲代码，可以很大程度的提升做题的准确率，也会避免一些敲代码时可能出现的问题。所以平时的做题中，我们强烈建议同学们做题之前对题目进行充分的分析，并将分析结构与注意点写在代码前的多行注释之中。同学们不要怕麻烦，在编程的起步阶段养成良好的习惯是非常的重要。

### 3.5 本章总结

我们强烈建议同学们书写代码的风格要参考本章中的规范。代码书写风格问题大多是因为在正常高校的 C/C++ 教育中没有被重视而导致的，但是这绝对不是我们代码书写风格混乱的一个借口，现在开始规范代码的书写完全来得及，起码比在工作时发现自己的代码都没法看下去时要好。同学们应该从现在开始规范自己的代码书写，对于以前写过的题目也要将代码进行规范，趁现在不良的代码书写习惯还没有养成尽快的规范代码书写习惯，良好的代码书写习惯会让我们从中获益的。

另外在做题的时候希望同学们能采纳本章中 3.4 的提议，做题时要在题目代码钱加上注释，注释内容是这道题的解题的思路和题目的考点（这个注释应该在敲代码之前打好，写完题目再写这个效果显然就没有那么好了）。



## 第四章 C/C++初步基础的常见错误与 ACM 例题分析

这一章中我们将结合实际 ACM 题目来分析一些常见的 C/C++ 的基础语法错误，结合实际案例的分析往往给人的印象会更深。对于 C/C++ 的基础部分同学们要用心学习，不论学习什么方面的知识，基础部分都要熟练掌握。要以为 C/C++ 基础部分是多么的简单，多数情况下最容易出现的问题就是那些基础问题。

### 4.1 标识符与主函数的问题

我们这里将以哈尔滨理工大学在线评测系统的 1000 题为例，我们先来参考一下样例代码：

```
#include <stdio.h>

int main(void)
{
    int a;
    int b;

    while (scanf("%d%d", &a, &b) != EOF)
    {
        printf("%d\n", a + b);
    }

    return 0;
}
```

这里我们主要通过这道题的代码说明一下标识符和主函数的问题。要求主函数的返回值类型必须是 int 类型，现在很多编译器不支持 void 类型的主函数，因此 void 类型的主函数在评测系统提交时将不会通过编译。而主函数要有返回值，通常主函数的返回值为 0。

对于标识符的问题，如果认真看了前面的部分应该就不会存在问题了，通常的命名法是以英文单词为基础，而且一般标识符的命名不倡导加数字，除非几个标识符之间确实有很强的逻辑关系。希望同学们给标识符命名时要按照一个合理的命名方法（标准）。

另外在线评测系统的 1000 题的样例代码也提示了如何实现多组输入输出该如何实现，因为在想评测系统中，每道题的测试数据都保存在一个文件中，因此当输入数据到文件结尾时输入就应该终止了，而 EOF 正是文件结尾的标志，因此使用输入输出函数时应该加上不等于 EOF 的判断。希望同学们能对这段简单的代码加以重视。

### 4.2 符号问题

在 C/C++ 语言中所有的标点都应该是英文标点，而不是中文标点，中文标点符号会引起编译错误。Code::blocks 中英文的标点大多是红色的，因此可以发现黑色的标点是中文标点。Code::blocks 中，标点符号是中文还是英文书写的还是比较容易辨别的。但是中文的空格也会导致编译出错，但是由于空是无形的，很难查出，所以强烈建议在书写代码时不要开启输入法。

如果因为开启输入法遇到了中文空格导致的编译错误那你就杯具了，尤其是在代码比较长的时候（ACM 中 200 行代码其实就算是很长的了），你需要逐行删除所有的空格（注意删除后为了代码的可读性应该用英文的空格恢复应有的缩进）；或者是选择将代码推到重写（代码长的时候怎么处理貌似都不大划算）。编者在新生辅导的过程中就遇到了这种情况，一个 30 行的代码足足找了 10 分钟才将中文空格的问题排除，希望各位同学要吸取教训啊。



### 4.3 变量的赋值与初始化

变量赋值与初始化的问题是一个程序员必须重视的，因为使用未赋值或初始化的变量可能会导致程序运行结果异常。对于没有初始化或者赋值的变量我们认为它的值是未知的，因为 C/C++ 没有规定如果变量未初始化或赋值时值为多少，因此你就不要指望那个编译器会给你保证变量的初始值。不论变量是全局还是局部变量都应该养成将变量初始化或赋值的习惯。记住，宁可相信所有变量定义后的值是随机的也不要特意去记忆某些编译器在定义变量给什么样的变量某一固定的值。

如果变量的定义距离第一次使用这个变量的距离过远，那么应该在定义变量的同时并初始化，防止意想不到的问题发生。例如某些变量未初始化可能会导致做题过程中，测试所有的单组数据的结果是正确的，但是在提交时确实 WA 的情况。如果是放到某个工程中，和有可能会让一个工程出现莫名其妙的问题甚至在运行过程中崩溃。尤其是未赋值的指针变量是非常的危险，因此变量只有在初始化或者赋值之后才能使用。

### 4.4 声明、定义与重定义问题

所有的变量和函数都需要先定义（声明）在使用，没有被定义的变量或函数如果被使用编译器将报错。同学们不要认为变量先定义在使用是一件非常麻烦的事，实际上这种机制防止了编程过程中一些可能隐含的错误。因此变量和函数先声明再使用的规则是非常的必要，然而对于刚刚接触 C/C++ 的朋友们往往会忽略定义问题而使用了没有定义（声明的）变量或函数。此时编译器报错并提示使用了没有声明的变量或函数，因此变量和函数应定要先定义或声明之后再使用。

声明只是表示告诉编译器我们将使用某一变量或者函数，因此声明不是定义。你可以将一个变量或者函数声明多次，但是定义只能定义一次。如果你在同一区域（例如全局或某个具体函数内）对一个变量进行多次定义，那么就会出现重定义问题。重定义属于一种 C/C++ 语法上的错误，一般在编译过后比较容易检查出来。为了避免重定义的问题，建议给函数和变量命名时参考匈牙利命名法的优点，要保证每个变量的名称应该与其功能是密切相关的。这样可以很大程度上避免重定义的问题。但是如果你在全局定义的函数名称与预处理中包含的某个头文件中的函数名称相同，那么将造成重定义。在全局定义函数时要避免函数名称与库函数或者其他头文件中包含的函数名冲突。

虽然 C/C++ 允许在全局和函数体内的变量名称相同（由于处于不同的区域，所以不算是重定义），但是不建议这样去做。局部变量的名称要避免与全局变量的名称冲突。这样可以防止程序员在变成的工程中产生不必要的失误。

对于变量的定义不可以随用随定义，尽管这样没有语法问题，但是这样做不利于代码的阅读和检查。因此在书写程序时要大体确定某个函数需要什么类型的变量，变量需要多少个，并在函数体中优先书写变量的定义。我们可以认为这是一种公认的变量定义规范，有很多东西编译器是不能保证的，例如良好的代码书写，因此这些能让代码书写更加清晰的书写规范一定要引起我们足够的重视，这样在以后 ACM 做题或是工作中才不会因为代码的书写问题吃亏。

### 4.5 数据范围问题

C/C++ 中每种数据类型都有自己固定的范围，如果计算或者输入的过程中超过数据的范围就会使程序运行结果出现异常，因此在变成过程中我们应该杜绝数据超过范围的问题出现。

我们应该要记住每种类型数据的数据范围（这里的数据类型一般是指在 32 位机器

上），记住了每种数据的类型之后，就应该分析在程序中数据最大可能会是多少以确定合适的数据类型来存储数据（通常多次相乘造成数据超范围的可能性比较大，尤其是对于整型数据）。这样就可以很大程度上避免了数据超范围的问题。

但是如果遇到一些程序可能涉及到树枝较大的整数，此时就应该使用 64 为整型。浮点数的计算时要采用 double 类型，因为 float 能准确表示的数据范围相对 double 要小很多。

数据范围问题应该引起足够的注意，因为有时数据超范围的问题很难检查出来。这就需要在编程之前正确的确定存储数据的类型，这个能力是一个合格的程序员必须具备的。

各种常见类型变量的取值范围和大小：

数据类型	数据范围	数据类型大小
整型 int	-2147483648 ~ 2147483647	4 字节
64 位整型 long long	-9223372036854775808 ~ 9223372036854775807	8 字节
双精度类型 double	存储的数据范围很大，能准确表示约 15 位数	8 字节
字符型 char	0 ~ 255	1 字节

## 4.6 运算符优先级问题

C/C++中提供了丰富的运算符，这让 C/C++编出来的程序能实现很多运算。但是运算符是涉及到优先级的问题，优先级问题是比较让人头痛的问题。作为文档的编者，表示自己没有记住所有运算符的优先级，确实都记住这些优先级有不小的难度。在实际编程中，往往出现一个涉及到多个不同类型的运算符时会使用括号来确定运算的优先级，这样代码看起来更加的清晰而且还不易出错。虽说我们用括号可以强制确定表达式运算的优先级顺序，但是我们应该尽量去记住各种运算符的优先级，作为一个合格的程序员这些基础能力是要具备的。

## 4.7 条件运算符

条件运算符是 C/C++中惟一的一个三目运算符，条件运算符的作用及使用方法这里就不赘述了（相见教材运算符部分）。对于一个 C/C++的程序员来说条件运算符一定要做到熟练运用，因为条件运算符的使用简便灵活，一些较长的判断代码使用条件运算符就能很快解决。

## 4.8 表达式的表示

表达式的表示问题在 C/C++中很重要，尽管我们看到编程中很多运算表达式与数学中是相同的，但是也有很多是不同的。例如我们想要用 C/C++判断是否成立“ $a < b < c$ ”，如果写成了“`if (a < b < c)`”，那么这个表达式实际上等价于“ $(a < b) < c$ ”。显然意思与我们的初衷是不符合，我们应该这样表述逻辑关系“ $a < b$  且  $b < c$ ”，写成 C/C++中的判断语句应该是“`if ((a < b) && (b < c))`”。

我们需要注意的 C/C++中的表达式是需要按照严格的逻辑关系来书写的，不能是想当然，我们要注意 C/C++中表达式与数学中表达式的差异。

还需要注意的是在书写表达式的时候尽量不要写一些过于复杂的表达式，过于复杂表达式难以阅读，使用过程中还很容易出现错误，因此如果不是必要的话要避免复杂的表达

式的书写。除此之外，建议一个表达式不要包含多种用途，这样包含多种用途的表达式通常书写相对较麻烦，程序员阅读程序时出错率也相对较高，此时建议把一个多用途的表达式拆分成多个表达式来书写。

## 4.9 除法问题

除法问题需要我们特别的注意，因为涉及到除法问题往往容易出现错误。例如两个整数的除法问题，很多程序员往往不经思考就将两个变量的类型定义为了整型。我们知道两个相同类型数据的运算结果也一定是一个相同类型的数据，整型数据也不例外。若定义整型数据  $a$   $b$ ，让  $a$  除  $b$  那么得到的结果应该是一个不大于  $a$  除  $b$  数学上的实际值的最大整数，例如整型数据 4 除 3 的结果就应该是整数 1。

那么我们应该在运算过程中将两个整型变量中的一个强制转化为 `double` 类型的数据，C/C++规定不同类型的数据进行运算，结果自动向精度较高的数据类型转换，此时就可以得到一个相对准确的浮点数表示的结果。

下面我们以哈尔滨理工大学在线评测系统中的题为例，题目要求计算整数  $a$  除整数  $b$  的值并保留两位小数。注意整数除整数的结果不一定是整数。下面参考样例代码，希望同学们在遇到除法问题时要考虑到结果很可能是小数的问题。

```
#include <stdio.h>

int main(void)
{
    double a;
    double b;

    while (scanf("%lf%lf", &a, &b) != EOF)
    {
        printf("%.2lf\n", a / b);
    }

    return 0;
}
```

上面的代码中，假定我们输入两个整数 3 和 4，程序的输出结果是 0.75。但是如果我们把  $a$  和  $b$  定义为整型变量但是运算过程中未将  $a$ 、 $b$  中的某个变量强制转化为 `double` 类型，那么输出结果就是 0。0 显然不是正确的结果，因此上面所涉及到的问题一定要注意。

还有一种除法问题可能会导致程序运行问题甚至会让程序崩溃，那就是除 0 问题。如果一个整型数据除 0 会直接导致程序崩溃，而浮点数除 0 会导致结果异常。当然通常来说我们是不会在  $a/b$  问题中将  $b$  的值输入为 0 的，因为我们知道这样做没有任何实际的意义。但是实际计算中，我们往往会让某一个数去除一个表达式的计算结果，对于一个复杂的计算过程或者表达式我们很难判断出表达式是否会在合法数据输入过程中是否会出现表达式值为 0 的情况。在很多问题的处理过程中我们最好在涉及到除某个表达式时，应该对表达式的值加以判断，在出现除 0 的情况时要进行特殊处理。这样就能解决因为除 0 造成的程序异常。

## 4.10 空语句问题

空语句的问题我们要引起重视，因为不起眼的空语句往往会导致一些让人匪夷所思的问题。空语句就是单独的一个分号，空语句在程序中也是要执行的，只是空语句多数情况下没有起到什么实际的作用罢了。

但是有的时候空语句却会让程序产生意想不到的问题，比如“while (.....) ;”。事实上循环体是这个空语句，所以如果在这种地方误写了一个分号循环就在执行空语句而不是程序员所认为的循环体。因此在循环和条件结构写完后，建议检查一下是否存在类似的情况致使循环体或 if 执行的语句变成空语句的情况。我们不建议在程序中出现空语句，这样能防止空语句对程序造成的影响。

#### 4.11 本章总结

本章中提到了很多有关 C/C++ 基础问题，我们发现刚接触 C/C++ 的同学们出现的错误往往就是这些非常基础的地方。因此不要总是以为什么东西都很简单，到头来自己总是会犯一些非常间的错误，打好基础是非常重要的，希望各位能充分的注意这些基础问题，这样才能避免基础的错误从而写出相对质量较高的代码。对于基础题目的练习可以从在线评测系统中的水题或者书后的练习题开始，不管怎么说扎实的基础是练出来的，不是空想出来的。只有勤练习，勤思考才能快速的掌握所需的基础知识。

## 第五章 ACM 中的输入输出

在计算机出现的早期，计算机主要应用于科学计算，并不像今天这样应用如此的丰富。对于科学计算而言，在保证计算公式正确的条件下准确的输入数据是保证程序正确运行的前提。尽管我们看到很多应用程序貌似不需要输入输出函数来实现某些功能，但是在科学计算和程序竞赛中输入输出是非常重要的。可以说熟练掌握输入输出是一个程序员必备的技能。

### 5.1 输入输出函数与 ACM

通常 C 和 C++ 是 ACM 竞赛中最常用的两种编程语言，因为 C/C++ 编译出的程序运行效率相对较高而且逻辑严谨，深受广大 ACMer 的喜爱。在 ACM 竞赛中，不论是使用 C 语言的选手还是使用 C++ 的选手通常会选择 C 语言的输入输出函数进行输入输出。主要原因是 C 语言的输入输出函数的速度要远远快于 C++ 中的输入输出流，而且输入输出函数的使用在某些方面上相对要灵活。尽管 C 语言的输入输出函数要程序员自身控制输入输出格式控制符的正确性，但是 ACM 比赛中代码运行的时间是非常重要的，所以不论是 C 语言选手还是 C++ 选手都选择输入输出函数。

因此 C 语言的输入输出函数是一个合格的 C/C++ 程序员必须掌握的。虽然 C 语言的格式控制种类并不少，但是花谢时间记忆这些控制符是非常值得的，熟练使用输入输出函数也同样是对自身编码能力的一种检验。

这里不是说 C++ 的输入输出流就没有用处，C++ 的输入输出流相对 C 语言中的输入输出函数要更加的智能化，可以识别输入输出数据的类型并做出相应的处理，还能够输入 C++ 中独有的 string 类型数据。对于一些难度较低，数据量较小的题目来说学习 C++ 的同学们可以视情况选择使用输入输出流来进行输入，有少数的情况下用 C++ 的方式来处理数据会更加方便。但是这种所谓的智能化付出了相当大的时间代价，付出时间的代价在 ACM 中是不划算的，有时题目输入输出的数据量很大，使用输入输出流有时连输入都没有结束程序就超时了。而对于一个细心的程序员来说，输入输出函数的书写是不会出现错误的，因此要求同学们要熟练掌握并使用 C 语言的输入输出函数。

### 5.2 格式控制符

对于 C 语言的输入输出函数，函数使用的正确性需要程序员使用正确的格式控制符来保证。这就要求程序员要记住每种数据类型所对应的格式控制符，常用数据类型的格式控制符我们必须牢记在心。这部分内容 C 语言的教科书还有群共享中的输入输出函数的课件（输入输出（授课：卢俊达））里均有详细的讲解，故这里就不再赘述了。希望同学们能对输入输出函数和其格式控制符有较为深刻的认识。

另外格式控制符的使用非常灵活，如果在格式控制符列表中进行一定的修改可以实现对很多较为复杂的输入输出的处理，但是要注意输入输出函数是严格按照格式控制符列表执行输入/输出操作的，因此如果题目没有要求不要在格式控制符列表中随意加入空格和其它的标点，否则可能会引起程序运行结果出现问题甚至程序崩溃。

我们的在线评测系统中，有一些题目就是专门练习输入输出函数和格式控制符的，同学们可以先做出这些题来巩固与提高对输入输出函数和格式控制符的掌握。

### 5.3 取址运算符与输入输出函数

“&”是取址运算符，取址运算符的作用是取得某个变量在内存中的地址。为什么通常在输入函数中要对变量使用取址运算符呢？我们定义了一个整型变量 a，如果我们想要

通过输入的方式改变整型变量 a 的值就必须知道整型变量 a 在内存的位置，因此在输入函数中我们使用了取址运算符。显然在输入数据时，我们应该给输入函数提供一个变量的地址，如果忘记写“&”的话程序在输入的过程中就会崩溃。因为显然一个非指针类型的变量存储的内容不是一个合法的地址，如果程序访问了一个非法的内存地址就会让程序产生崩溃。（标准的样例代码可参照之前给出的全部标准的样例程序）

不过有一个例外，就是在输入字符串的时候。在 C 语言中我们会用一个字符数组来存储一个字符串，对于一个数组来说，数组的名称就是一个指针常量且指向数组的首地址。而输入字符串时只需提供首地址，知道字符串的首地址使用控制符%s 就可以输入一个字符串，样例代码：

```
#include <stdio.h>

int main(void)
{
    char string[200];

    scanf("%s", string);

    printf("%s\n", string);

    return 0;
}
```

但是我们发现有些参考资料甚至教材中，类似上面样例的程序中对字符串的输入函数写成了这样的形式：

```
scanf("%s", &string);
```

对于这种写法，集训队做了一个实验。我们将字符数组 string 的首地址和“&string”输出并对照了输出结果（测试环境：windows7 64bit 操作系统，GCC 4.4.1 编译器，编译环境 Code::blocks）。

```
printf("string:%p &string:%p", string, &string); //输出地址的格式控制符是%p
```

我们发现字符数组 string 的首地址和“&string”的值是完全相同的，编译器并没有给出任何的警告。但是在接下来的另一个试验中“scanf("%s", &string);”这种输入方式得到了编译器的警告（测试环境：Linux 64bit 操作系统，GCC 4.7.0 编译器，编译环境 Code::blocks），编译器认为输入输出函数格式控制符列表中的格式控制符与要输入的数据类型不匹配。虽然这样写目前并没有发现会造成什么问题，但是可以肯定的是这样的书写方式实际上并不严谨，因此在输入字符串时一定要按照形如上面样例代码的形式进行输入（目前可以确定样例代码中的输入方式一定没有问题），不建议使用“scanf("%s", &string);”这种输入方式。

## 5.4 单个字符的读入与 getchar 函数

在 C/C++中读入单个字符我们可能会使用 getchar 函数，getchar 函数可以读取任意的一个字符，比如回车。在使用 gets 函数循环读入字符串时一般会用到 getchar 函数来捕获在循环使用 gets 函数之前输入其他数据后按下的回车，因为 gets 函数可以读入一行的字符，包括回车和空格。因为回车的输入标志着一行的输入结束，因此 gets 函数捕获那个回车并认为是一行字符输入结束。如果前面没有 getchar 函数来捕获那个回车，程序的处理结果可能会出错。getchar 函数使用的样例代码：

```
#include <stdio.h>
```

```

int main(void)
{
    char char_array[300];
    int number;

    scanf("%d", &number);
    getchar(); //使用 getchar 函数捕获输入 number 后的回车

    while (gets(char_array))
    {
        //具体函数处理
    }

    return 0;
}

```

上面的代码体现了 `getchar` 函数最常见的用法，哈尔滨理工大学在线评测系统中 1562 题是 `getchar` 函数使用比较好的一道练习题，当然做这道题之前还是再看看 5.5 中对于 `gets` 函数的描述。AC 这道题需要对 `getchar` 和 `gets` 函数较为熟练地掌握。

这里要重点说明一下，不要想当然的认为 `getchar` 函数的返回类型是字符型，其实 `getchar` 函数的返回值类型是整型。这点要特别的注意，如果把 `getchar` 函数的当成一个返回值类型为整型的函数来使用的话在以后的工作中会出很大的问题。

## 5.5 gets 函数与字符串

有些时候我们需要读入并处理一行的字符，这一行字符可能包含空格。这个时候我们不能使用输入函数进行处理，因为使用输入函数读入字符串时不能读入空格。需要注意的是 `gets` 函数可以读入回车，这点跟是使用 `gets` 函数的过程中必须注意的。`gets` 函数的用法：`gets(字符数组首地址)`，这样就可以用字符数组存储一个带空格的字符串了。

但是 `gets` 函数使用过程中安全性要靠程序员自己保证，每次读入字符串的长度不能超过字符数组的长度，否则会造成程序崩溃。因为 `gets` 函数的参数只有一个字符数组的首地址，我们通过一个数组的首地址是无法判断代码中数组究竟开了多大，因此 `gets` 函数就无法提供相应的安全保障。在使用字符数组存储字符串时首先要保证字符数组开的足够大。

C 语言中很多功能和函数使用非常灵活，但是又有一定的危险性，例如一些函数形参仅有指针参数的库函数。单凭指针是无法保证函数使用的安全，因为使用指针的过程中可能会造成内存地址的非法访问。因此很多东西都需要程序员来保证，比如数组下标的实际取值范围等。因此从现在开始养成严谨的编程习惯是非常重要的，一个功底深厚的 C/C++ 程序员是可以轻松地使用 C/C++ 各种功能的。

## 5.6 本章总结

实际上关于输入输出函数的很多内容 C 语言课本和群里的课件上已经讲得很明白了，这里所写的不过是一些需要注意的事项。新的内容无非就是 `getchar` 函数和 `gets` 函数，确实这两个函数很重要（本章的内容要认真读啊），在 ACM 中会经常用到，它们的使用一定要熟练，否则以后的比赛中会很吃亏的。

还就是要对使用 C++ 的同学们说的是 C 语言的输入输出函数是一定要会的，因为输入输出函数 C++ 中也是有的，而且完全和 C 语言中的输入输出函数一样。因为输入输出函数的速度快，使用的灵活性也不错，所以 C++ 程序员也是要掌握输入输出函数的。对于输入输出函数的练习建议同学们做一下杭州电子科技大学在线评测系统 ([acm.hdu.edu.cn](http://acm.hdu.edu.cn)) 中

的 1089 到 1096 题和第一次新生练习赛中的所有题目，做完这些题目相信同学们会对输入输出函数的使用更加熟悉了。



## 第六章 条件结构与循环结构

本章我们要对条件结构和循环结构中容易出现的问题进行分析，当然这里也会对一些书本上没有提到的知识点进行一定的分析讲解，而这部分知识对提高同学们编程质量有帮助。希望同学们能从本章中详细了解条件结构和循环结构。

### 6.1 if 条件语句的使用

if 条件语句是一个 C/C++ 程序员必须能够熟练使用的，if 语句虽然简单但是能够起到很大的作用。If 条件语句的最基本的形式：if（表达式）。如果表达式的值为真，执行 if 语句后面的一条语句或者复合句（复合语句是指有大括号括起来的一系列语句）。

导致 if 语句执行问题主要是 if 后面括号中的表达式的值恒为真或恒为假，这个问题需要注意，当 if 语句出现问题时应该优先检查表达式是否会出现值恒定的情况。如果出现这种情况应该检查表达式的书写并及时更改，我们要保证这个表达式的值可以通过某些变量控制其真假。

### 6.2 if 与 else 配对问题

else 总是需要和 if 同时出现的，这点是需要我们注意的。尤其是当出现选择结构的嵌套问题时，容易出现对于某个 else 找不到相应的 if 匹配，或者是 else 与 if 的匹配混乱。由于这个问题产生的错误实在是太多了，我们无法一一列举，那我们就找一个典型的案例，例如下面就是一段编译错误的代码：

```
#include <stdio.h>

int main(void)
{
    int a;

    scanf("%d", &a);

    if (a > 4)
    {
        printf("%d\n", a);
    }

    a = 5;

    else
    {
        printf("OK\n");
    }

    return 0;
}
```

编译器提示：“error: ‘else’ without a previous ‘if’”，编译器提示对于一个 else 找不到相应的 if 进来匹配。我们发现，if-else 结构被“a=5;”这条语句隔开了，因此导致了代码中 else 找不到与之匹配的 if。因此编译器就报错了，当我们删除“a=5;”这句时就可以通过编译。尽管 if 的应用是非常的灵活，但是程序员必须保证每个 else 都有一个 if 来对应。当然如果对应的比较混乱，没有逻辑，就有可能让程序的运行结果与预期有偏差。因此要注意 else 和 if 的匹配关系。

## 6.3 判断变量与零值或其它值的关系

虽然这个问题看起来很简单，但是实际上则不然。对于这种问题应该注意代码的书写规范，这个问题实际上并不简单。这一小节主要就是谈一谈这个问题的，由于本小节的内容属于提高能力的部分，因此建议对 C/C++ 基础不牢固的同学先充分理解教材上的内容再来看这部分的内容。

### 6.3.1 整形变量与零值的判断

判断整形数据与零值的关系相对比较简单，我们定义了一个整型变量 `number`。接下来的两个 `if` 语句是判断整型变量 `number` 与零值相对标准的形式：

```
if (number == 0)           // 判断 number 是否等于 0
if (number != 0)           // 判断 number 是否不等于 0
```

### 6.3.2 浮点型变量与零值的判断

不得不说一下浮点数的问题，有的时候使用浮点数表示一个数并不准确，双精度浮点数中存储的值往往是一个非常接近这个数的一个值，这个值不完全等价于这个数本身（在乘除法的计算结果上体现的更明显）。因此用浮点数表示一个数或者是进行比较大小的判断会造成一定的误差，因此在判断的过程中，如果计算结果在一定的误差之内都应该是允许的，因此直接拿双精度浮点数据和 0.0 或者 0 来判断是否相等并不科学。我们建议双精度浮点型数据与零值比较时应该先确定一个精度范围，在精度范围内的误差都是允许的，例如定义一个双精度浮点型变量 `number` 与零值比较：

```
const double eps = 1e-8;           //eps 为自定义允许的精度误差范围
if ((number > -eps) && (number < eps)) // number 在精度允许的范围时，就可以认为值为零
```

### 6.3.3 判断字符型变量与字符的关系

对于字符来说有一个 ASCII 表，表中记录了 256 个字符的 ASCII 码值。我们不建议让某个字符直接与另一个字符的 ASCII 码值直接比较，尽管那样做是没有问题的。建议对于一个字符变量，可以让它直接和某个字符相比。因为如果直接和 ASCII 码比较在一个代码行很多的一个程序中有可能让阅读的程序程序员误解为这是整型变量与某个数值的比较（不怕一万，就怕万一，还是保险起见）。定义一个字符型变量 `character`，判断该字符与大写字母 A 是否相等，样例代码：

```
if (character == 'A')
if (character != 'A')
```

### 6.3.4 布尔型变量与零值的比较（如果使用 C 语言这部分可以跳过）

布尔型数据的值只有两个，`true` 和 `false`。对于布尔型数据来讲零值就是 `false`，非零就是 `true`。但是用布尔型变量与零值比较时通常不会直接拿来和 `true` 或者 `false` 进行比较，跟不可能和数字 0 或 1 进行比较。若定义布尔型变量 `flag`，比较布尔型变量与零值的关系：

```
if (flag)
if (!flag)
```

通常布尔型变量与零值的比较都是上面两个 `if` 语句所示的形式，一些质量较高的 C/C++ 的学习资料上认为这样的书写风格是相对良好的书写风格。

### 6.3.5 小结

其实对于这些细节部分，编者以前并没有注意。编者以前做的很多 ACM 题目中都是用浮点数直接与数字 0 进行比较，根本没有考虑过浮点数误差问题，也没有考虑过其他类型变量比较时的良好代码风格。几天前看了一个有叫《高质量 C/C++编程的文档》才突然发现这些问题，发现自己写的代码居然有那么多毛病啊。为此编者开始对进来写过的代码逐个进行了剖析，有很多代码已经被推到重写。

这里我想说的是规范代码的书写应该从 C/C++的入门开始，这些关于 if 语句的细节要注意，规范这些细节无疑对以后的学习和工作有帮助。如果你像编者那样学习 C++(编者最开始是学 C++的)后很长时间才认识到这些东西那就免不了折腾一阵子。要是对这些认识的再晚一些，恐怕就会有不小的麻烦了，长期积累下来的习惯是很难改掉的，尤其是那些不良的编程习惯。

## 6.4 循环语句

### 6.4.1 for 循环语句

for 循环语句是最常用的循环语句，for 循环语句最大的优点就是结构清晰，但是要注意 for 语句的书写问题，for 后面的括号中三条语句是应该用分号隔开，千万不要写成逗号。

比如这样一条循环语句：“for (i=0; i<10000; i++)”，其中的“i++”建议写为“++i”。因为此时二者的作用对于整个循环来讲是相通的，但使用后者会有在理论上的小幅度的速度优势，在需要循环次数很多的程序中，使用后者理论上会取得少许的时间优势。对于之后可能涉及到 C++中的迭代器，如果 i 是一个迭代器的话，那么使用“++i”就会取得较为明显的时间上的优势。

### 6.4.2 while 循环语句

while 循环语句的使用也是比较广泛的，while 语句直接判断 while 后面括号中的表达式是否成立，若成立则执行 while 后面的一条语句或者复合句。很难说 while 和 for 两种循环结构那个更优秀，只能说一个高水平的程序员会因程序代码需要而做到在合理地方分配两种循环的使用。因此 while 和 for 循环结构同等的重要，对哪一种结构不熟悉都是不行的。

## 6.5 循环语句的效率问题

本小节的内容仅仅是一个理论上的分析，实践上不一定会达到预期的效果（论述内容仅供参考阅读）。这里编者要说的是要真正学好 C/C++光是靠看几本 C++的书是远远不够的，想成为一个 C/C++的大牛需要积累丰富的知识，比如和计算机硬件有关的知识。建议同学们以后学习完有关硬件的知识之后可以回头看看一些对于 C/C++更深层次论述的书籍。这些书会告诉同学们怎样去书写更高质更量高效率的代码。

### 6.5.1 多层循环嵌套与循环效率问题

对于循环来说，处理器对循环的执行实际上是有优化的，这样就使得循环结构能够更加快捷的执行。在多层循环的嵌套中，CPU 需要在循环之间来回切换，频繁的切换会导致循环的处理效率不佳。因此遇到多层循环的嵌套时，如果可能（在不影响程序运行结果的前提下）尽量将循环次数多的那层循环放在内层。这样从某种程度上来讲对循环的执行速

度的提升会略有帮助。

### 6.5.2 单层循环的效率问题

如果在一个单层循环中，如果循环语句中出现很多条件分支语句，那么有可能会对 CPU 对循环结构的优化造成一定影响，因此尽可能不在循环中使用大量的条件分支语句。

6.5 中所有的内容只是提供一个参考，具体原因目前不需要去深究。当各位只是储备量达到一定的水平时，这些东西就不难理解了。

### 6.6 break 与 continue 语句

对于这两种语句的用法相信各位都不陌生了，break 的作用是跳出循环或者 switch 语句，continue 的作用是跳过某一次循环并开始下一次循环。但是对于这两种语句要注意的是它们的作用范围，对于循环语句来说，break 和 continue 的作用范围都仅限于直接包含该语句那层循环。因此显然不要指望使用 break 语句直接跳出多层循环，这样是不可行的。

### 6.7 分析死循环问题

死循环问题是一个比较让人头痛的问题，因为程序陷入死循环中我们只能强制终止程序，所以死循环是我们不愿意看到的现象，这里我们来讨论一下造成死循环的原因和如何避免死循环。

不论是 for 循环结构还是 while 循环结构都需要写明判断循环是否继续的表达式，如果表达式为真循环继续执行，否则将退出循环。如果因为某种原因导致了判断表达式的值恒为真（或者没有判别表达式），而且循环体中有没有可以跳出循环的语句，那么程序将陷入死循环。还有一种情况就是表达式不是常量表达式，但是因为某种原因，循环的条件总是会成立，此时程序也会陷入死循环。

要避免死循环的问题，首先要检查循环的判别表达式是否是一个常量表达式，因为非 0 的常量表达式的值恒为真，如果发现判别表达式被误写成了一个常量表达式那么就需要及时的修改。接下来我们应该检查循环过程中是否存在让非常量判别表达式恒为真的情况，若存在这种情况就要及时的修改，例如在一个 for 循环结构中“for (i=1; i<10; --i)”，i 的值永远会小于 10，程序陷入死循环。因此这个时候我们就应该对循环做出修改，让 i 存在大于等于 10 的情况，这样就能避免死循环了。

### 6.8 本章总结

对于条件分支语句和循环语句来说练习是解决问题的最好方式，编者在学习 C++ 的使用也遇到过各位同学们出现的问题，比如说冒泡排序就曾经纠结了很长时间。后来发现随着代码书写越来越多，一些这样的基础问题就很少出现问题了，因此多练习是提高能力一成不变的真理。

## 第七章 函数的使用

函数的使用在 C/C++ 中时非常重要的，如果程序的代码量很大时仍然将所有内容都写的主函数中，即使你主函数内的代码写风格的再清晰代码阅读还是非常麻烦。面对连续的长代码来说一般是不容易检查错误的，因此对于稍长一些的代码我们就应该按代码的功能写成不同的函数，这样整个代码就显得非常的清晰，也便于查找错误。

一个合格的程序员对必须掌握函数的调用和参数传递和其它相关基本知识，尽管有些部分在 ACM 竞赛中体现的不是很明显，但是在今后的工作中这些东西都是很重要的，同学们务必认真的对待函数。

## 7.1 函数的定义与声明

在 C/C++ 中如果要使用某个函数就必须先声明或定义函数（函数的声明和定义详见教材），我们建议在代码的书写中，先在主函数前面声明函数。函数的定义及函数体部分一次按照顺序写到主函数之后。先写函数声明在一些代码较长的程序中是有很优势的，函数先声明后定义可以便于代码的检出，还可以使函数在需要相互调用的时候不出现问题（尽管函数之间相互调用的情况在 ACM 中很罕见）。写函数声明绝对是一种良好的编程习惯，因此建议同学们在多函数程序中的书写方式按照这一小节中的建议的那样书写。

## 7.2 函数体的书写规范

函数体的书写应该按照一个规范，我们发现绝大多数学生书写的函数体往往是存在问题的，这里也包括编者。函数体的书写会直接影响到程序的质量，书写不过关的函数会让程序漏洞百出。虽说这不完全是学生的问题，因为在学校授课的过程中老师就没有对学生传授编程质量的问题（很多学校都是这样，包括我们熟知的各大名校）。但是勤奋的同学们总是能找到提升自己的途径，让自己的编程水平远远高于大多数同学，所以就不要给自己找借口了，现在重视编程质量还为时不晚。

对于函数的书写首先要注意的是定义函数的返回值类型要和函数实际返回的变量类型相同，这样可以防止程序中出现一些我们难以预料的问题，以后同学们可能涉及到上千万行代码的程序开发，如果因为函数返回值问题导致程序异常显然是不值得。因此函数书写时程序员一定要对函数定义的返回时类型和函数实际返回变量的类型进行严格的检查。而且每个函数都应该有自己的返回值，即使是 `void` 类型的函数我们也应该写上一个 `return` 语句，`void` 类型函数的返回只需 `return` 后面加一个分号即可。还要注意的是函数的处理过程中不应该出现没有返回值的情况，这样的函数在传递某些参数后会间接地导致函数此时没有返回值。好在这种问题一般编译器会给予警告，当编译器警告说函数可能存在没有返回值的时候应该尽快修改，以免程序在运行时出现问题。

函数都是具有参数列表的，如果这个函数调用过程中不需要传递参数，那么函数定义、声明时参数列表应该用 `void` 填补。这样在以后的工作中，可以防止其他人阅读代码时误认为函数定义时忘记写参数列表。在 C 语言中，使用 `void` 填补参数列表表示函数不接受任何的参数传递，而空括号则表示函数对参数传递表示沉默。

编者在以前这些错误基本上都犯过，当然过去自己练时有些小问题并没有注意。今年八月份开始准备给新生写一份常见错误分析时这些在自己身上曾经多次出现的小毛病才重新拿出来仔细的推敲。之后又参考了很多资料，总算是把函数书写这部分的问题纠正了大半。不管怎么说确实是费了一番周折，因此同学们在函数的书写方面现在就应该按照上面的这些要求去做，要知道走前辈们走的弯路是一件很痛苦的事情。

## 7.3 主函数的问题

主函数问题是一件比较让人头疼的问题，尤其是在给新手们指导的过程中，仅仅是一个主函数的类型问题就足够让我们这些教练疯掉。最根本的原因竟然是很多的书上都用 `void` 类型的主函数（其中不乏一些大学 C 语言的教程），不过不巧的是现在新版本的编译器大多不支持 `void` 类型的主函数。以前的计算机还不像现在这样普及的时候，计算机主要用途当然就是计算，而当时不同计算机之间代码移植的问题不像现在那么好解决，`void` 类型的主函数更方便代码的移植。《C++ Primer Plus》中认为 `void main` 在较早的使用可以

轻松地适用于不同的系统，但是 `void main` 从没有成为一个 C 语言强制性的标准。

随着时间的推移，现在有很多 C 语言编译器不支持 `void` 类型的主函数了，这些编译器要求主函数的返回值类型必须是 `int` 类型，函数的返回值将返回给系统。因此现在 C 语言中主函数类型要使用 `int` 类型，函数的返回值应当是整数 0。

在 C++ 中主函数的类型是 `int` 类型（记得有这个规定），而函数的返回值为 0，这点是毋庸置疑的。但是一些早期的 C++ 编译器（以 visual C++ 6.0 为代表）并不合乎标注，仍然允许 `void` 类型的主函数。我们建议即使是编译器允许使用 `void` 类型主函数也一定要用 `int` 类型主函数。如果说目前很多有关 C++ 项目开发的书籍是以 VC 6.0 为基础，而书中就是用 `void` 类型主函数，对于此事我是想说如果哪家公司还是在用 VC 6.0 那一定是那个公司经理脑子进水泥了。如果真的有同学以后想搞项目开发、MFC 之类的尽量避免使用 VC 6.0，VC 2005 及以上版本还是可以考虑的，市面学习 VC 2005 及以上版本的高质量书籍很多的，而且 2005 及以上版本的 VC 还是比较相似的。

## 7.4 函数的调用

作为一个 C/C++ 程序员函数的调用是必须掌握的，函数的调用过程中不仅可以利用函数内部的实现来完成某些功能，也可以使用变量来承接函数的返回值以进行其他操作。函数的调用是非常灵活的，同学们可以自己体会一下。但是函数调用时应该注意下面的两个问题：

第一：函数调用时传递参数的个数与类型应该与函数定义中函数的参数列表严格的对应，如果函数定义时参数列表用 `void` 填补，那么函数调用时函数后面的括号中应该什么都不写。这点要注意，例如之前我们定义了一个函数，函数声明：“`int Function(void)`”，调用时应该写成：“`Function()`”。这点要特别的注意，如果在函数调用时写成：“`Function(void)`”将出现编译错误。

第二：主函数可以调用任何的函数包括主函数自身，其他函数可以调用除主函数之外的任何函数，这点我们要特别的注意。

第三：函数的递归调用会消耗内存，因此在使用函数递归调用时要注意

## 7.5 函数的形参问题

注：对于这部分内容，如果你有一定的指针基础，会进行指针的基本操作，那么阅读效果更好。强烈建议看这一节内容之前先复习一下教材中有关指针的部分。

函数的参数传递时问题需要引起我们的注意，当程序把数据传递给函数的形式参数时，函数会使用不同于原变量所在的内存空间中把传递来的参数拷贝一个副本。对于变量而言，把它传递给一个函数，函数实际上操作的是在另一个内存区域拷贝的副本，因此不论函数怎么怎么改变形参的值也无法改变原来变量的值，改变的仅仅是原变量的一个复制品。

如果函数的参数是一个指针那么就可以实现对这个指针指向的变量进行更改，但是传递参数后，在调用的函数中的这个指针实际上也是一个拷贝的副本（指针也是变量）。我们可以通过这个指针改变指针变量所指的地址中的变量值，但是在函数中我们无法修改这个指针所指的地址（即指针变量中存储的地址），道理和前面的是一样的。指针在函数参数传递的过程中也是一个比较容易被误解的，有很多程序员误认为函数在传递指针参数时是直接对指针本身进行操作，这个误区也就给以后可能出现的很多错误“埋下了伏笔”。

正是因为函数的参数传递时函数是在操作传递来形参的一个拷贝的副本，这样在函数



的调用过程中会出现一些问题。因为很多程序员都认为函数形参的操作实际上是对传递的参数的本身进行操作，正是因为这些程序员在基础概念方面混淆不清导致了编写出的程序之中问题层出不穷，有些时候真的是不能赖 C/C++ 不安全，更多的时候是程序员自身的问题导致了程序安全性和质量不高。我们要认清函数传递参数的实质，这样就会在实战当中避免很多问题。

编者上个月其实就在这里吃过亏，一个将近两百行的程序检查了半个多小时才发现是因为对函数参数传递机制认识不够而导致的问题。当程序调试完毕后已经花掉了很长很长的时间，但是如果实在实际工作中就没有这么便宜的事了，往往可能会因此白白浪费数个小时甚至几天（很不值得啊）。因此现在开始就应该对函数传递参数的机制了解。像这种细节性的问题你掌握的越多，相对其他的程序员你越有优势。

对于这个问题我们还是用实际的程序来演示一下，通过下面的程序相信各位会对此有很深刻的印象。

测试环境：

操作系统：windows 7 64bit

编译器：GCC 4.4.1

编译环境：Code::blocks 10.05

演示程序：

```
#include <stdio.h>
```

```
void Function(int p);
```

运行结果：

```
in main, add_number = 0022FF18
in Function, add_number = 0022FF00
```

```
int main(void)
```

```
{
```

```
    int number;
```

```
    int *pointer = &number;
```

```
    printf("in main, add_number = %p\n", pointer);
```

```
    Function(number);
```

```
    return 0;
```

```
}
```

```
void Function(int p)
```

```
{
```

```
    int *q = &p;
```

```
    printf("in Function, add_number = %p\n", q);
```

```
    return ;
```

```
}
```

我们明显的看到，定义在主函数中的变量 `number` 的地址，与这个变量传递给函数 `Function` 的那个形参的地址是不同的。因此把结论推广一下，通过函数更改任何类型的形参（包括指针类型）的值都无法更改原变量的值，原变量在与形参不同的一个内存地址中，所以一定要记住变量的副本和其本身是不等价的。

## 7.6 本章总结

不可否认函数是非常重要的部分，一个程序是离不开各种的函数调用的。因此我们要会用函数，会写函数，更要会写高质量的函数。只要函数的质量得到保证，整个程序的质量才可能保证。本章中介绍了一些有关函数的问题，当然了其实关于一个语言的每一部分都可以总结出很厚的一本资料，更多的内容就需要同学们在课余时间对这些东西进一步的了解。不过文档中提到有关函数的问题尽量去掌握，掌握这些内容对个人的编程能力就会有一定的提高。



## 第八章 数组与 C/C++编程

数组对于高级程序语言是非常重要的，编程过程中很多功能的实现都要依赖数组，包括很多常见的数据结构和一些算法。我们应该对数组的使用熟练地掌握，要做到熟练地使用一维数组和二维数组。多维数组的使用一般不多，因为多维数组的使用还受到很多因素的限制，比如在 C/C++中数组不能够开太大。假设有一个四维整型数组的每个维度都是100，那么这个数组的大小就是  $100^4 \times \text{sizeof}(\text{int})$ ，这个数值是庞大的。

### 8.1 数组的性质

C/C++中数组是非常重要的，对于数组我们应该先认清数组都有哪些性质，这样在对数组的学习过程有很大帮助。

性质 1：数组中存储的所有元素的类型必须相同

性质 2：数组在逻辑上是紧密相连的，在内存中的排列也是呈线性的

性质 3：一个长度为  $n$  的数组的合法下标范围是  $0 \sim n - 1$

### 8.2 数组的定义及初始化问题

这里我们对于数组的定义问题和初始化的问题简单的分析一下，首先是有关数组定义的问题，建议在数组的定义过程中要指定数组每个维度的大小（从 ACM 竞赛的角度出发）。这样在做题的过程中更有利于对错误的检查。

对于数组的初始化过程建议同学多看看书，在数组的初始化方面书上的内容已经是讲的很到位了，同学们要认真阅读。

### 8.3 数组的下标问题

C/C++中编译不会对数组的下标进行检查，一个数组下标使用是否合法完全是靠程序员自己来保证。如果程序员对数组下标检查不严格导致数组下标越界，那么就可能会导致程序崩溃。因为如果数组下标越界，那么程序在运行时就可能会访问不属于系统给程序分配的内存。如果这段内存被其他程序占用的话程序就崩溃了。

可以确定的是在程序运行的过程中谁都不希望看到程序的崩溃，因此程序员自己检查数组下标是一个必要的过程。那么我们就需要从程序的执行过程中查起，这个过程需要一定的耐心，但是非常值得。

### 8.4 数组大小的讨论

一个程序中数组是不可以开的无限大的，如果在函数中定义数组（指的是静态数组），我们一般不能定义一个大小 1MB 以上的数组，否则程序会崩溃。但是在 ACM 竞赛中可能需要开辟很大的数组，这时我们可以在全局开辟数组，全局开辟数组大概可以开辟相对较大的数组。当然如果开的非常大当然就不可以了，事实上一般也没有什么情况需要体积非常庞大的数组。

如果数组开的不够大就会造成数组下标越界的情况，下标越界就可能造成程序崩溃的问题。通常比赛中我们一般会选择现在全局开辟足够大的数组，一般题目中数组长度顶多也就几十万，不会太大的。

对于数组的问题在本章先讨论这么多，不是说数组可以讨论的地方太少，是因为数组

和指针与字符串的结合时相当的紧密，因此许多和数组相关的内容需要和指针与字符串一起来讨论，这样的效果应该比较好的。

## 第九章 字符串及字符串的处理

字符串及其处理一直是 C/C++ 中非常重要的部分，因此对字符串的处理方面我们要给予重视。往往在一个字符串的处理过程是对一个程序员的能力的考验。

### 9.1 字符串与数组

在 C 语言中是没有字符串这种类型的，C 语言中只有字符类型。因此我们存储一个字符串时使用一个字符数组来存储字符串，这里我们就来讨论一下字符串与数组。

C 语言中虽然字符串使用一个字符数组来存储，但是对于整个数组而言并不都是字符串的一部分，因为字符串有自己的结束标志 ‘\0’。在处理字符串时遇到 ‘\0’ 时就代表字符串结束，正因为如此，使用字符数组存储的字符串才和单纯存储一堆字符的字符数组有了区别。有兴趣的同学可以尝试用输出字符串的方式来输出一个普通的字符数组，看看会有什么情况发生并分析一下为什么会发生这种情况。还有就是在使用字符数组来存储字符串时，一定要保证字符数组开的足够大，否则会发生数组越界甚至导致程序崩溃。

接下来我们需要说明一下字符数组的初始化与字符串的问题，一般来说这里是比较容易出错的一个地方。C/C++ 规定可以在一个字符数组定义的同时将其用一个字符串初始化，但是不允许在字符数组定义之后用一个字符串给字符数组赋值，否则编译器将会报错，编译器认为复制符号的左值与右值的数据类型不匹配。因此如果希望用字符数组存储某个特定的字符串应该在字符数组定义的同时将其初始化。

虽然 C++ 中有字符串的类型，但是建议学习 C++ 的同学们要学会使用字符数组来存储并处理字符串的方式，这样对自身的能力提高非常有帮助。

### 9.2 字符串的处理

对于字符串的处理，C 语言有专门的库函数来实现对字符串的一些基本的处理。这些函数的用法相信同学们对照课本很快就会使用了，但是建议同学们要学会自己写对字符串处理的过程，这样可以提高自己的编码水平。

对于几种常见的字符串处理，同学们要学会自己去写处理函数，而且处理函数的代码质量一定要有保证，低质量的代码对程序来说有害无益。我们先来说一下计算字符串长度的方法，计算字符串长度是要利用判断字符串的结束条件，这个函数相信同学们一定能独立的完成，那么接下来就是看函数的代码质量如何了。

之后就是字符串连接函数还有字符串拷贝函数，其实从实现原理上来说这两种操作的实现并不难，只要在细节上足够的注意，就可以写出相对质量不错的代码。建议同学们有时间要写一下这两个函数的实现过程。此外就应该是字符串比较函数了，对于字符串比较大小的方式在书上有详细的说明，现在只需要我们来自己实现以下而已。通过自己书写字符串处理函数来达到对字符串的处理目的可以锻炼代码的书写能力，同学们有时间一定要亲自写一写这些函数，写完后可以和集训队的同学们进行交流。看看自己写的代码在哪些地方还存在着不足之处，并及时的修正错误。

## 第十章 有关指针的讨论

可以说指针是 C/C++ 的精髓所在，能够熟练使用指针是一名优秀的 C/C++ 程序员必须做到的。尽管指针的使用上有一定的难度，而且还有一定的危险性。大多数指针使用过程中出现的各种问题是由于程序员自身的基础、概念不牢固，代码编写能力较低的因素导致的。很多指针的问题是可以提高程序员自身能力而避免的。尽管人们都说指针是 C/C++ 中最难的地方，但是如果因为这个就去特意回避指针，那么问题就更大。如果以后去搞与 C 或者 C++ 有关的项目，指针你是躲不掉的。不如我们现在就开始对指针进行一定的了解，这样可以避免在一些简单的指针使用中出现错误，既然指针躲不掉就要勇敢的面对指针。

### 10.1 指针的初始化问题

对于指针变量在定义的同时需要初始化，我们可以将指针变量初始化为一个合法的内存地址，也可以初始化为 NULL。没有初始化的指针的指向不能确定，可以认为是一个随机值。因此在使用这种指针是很容易出现访问非法地址的而导致程序崩溃情况，而且指针变量不在定义时初始化会给代码的查错带来很大的困难。因此对指针变量要格外的注意其初始化问题，因为如果指针中的存储的地址出现了问题就会直接导致程序的崩溃。

### 10.2 指针与动态内存申请

指针的可以用于动态内存的申请，我们通常在用一个指针变量 p 申请动态内存后当内存不需要再使用了就需要将这段内存释放掉。如果忘记将内存释放，随着程序的运行程序可能会不断地向系统索要内存导致程序使用的内存异常的多，这就是内存泄露。如果某个程序出现内存泄露会拖慢系统的速度，同时也会导致其他的程序无法从系统中得到内存。尽管现在的系统对内存泄露有一定的处理，不会让某个程序因为内存泄露将系统内存完全消耗，但是内存泄露的现象我们应该是要杜绝的。最基本的就是在申请动态内存后要记住释放。

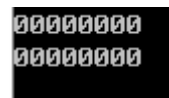
有一种情况不大容易被察觉出来，下面我们代码来演示一下这种不易被察觉的情况，系统我们仍然选择了常用的 windows 7 操作系统，编译器是 GCC 4.4.1，代码书写的环境为 Code::blocks 10.05。

演示代码：

```
#include <stdio.h>
```

```
void get_memory(int *p);
```

运行结果：



```
int main(void)
```

```
{
```

```
    int *pointer = NULL;
```

```
    printf("%p\n", pointer);    // 输出指针 pointer 所指向的地址
```

```
    get_memory(pointer);
```

```
    printf("%p\n", pointer);    // 输出调用函数后指针 pointer 所指向的地址
```

```
    return 0;
```

```
}
```

```
void get_memory(int *p)
```

```
{
```

```
    p = (int*) malloc(400);    // 用指针 p 申请一处动态内存
```

```
    return ;  
}
```

我们发现在调用了申请内存函数后，指针 `pointer` 的指向并没有改变。都知道，正常使用动态申请内存时，系统会在内存中找一块大小合适的内存空间分配给动态申请的内存，而这段内存会由一个指针变量来承接。我们姑且先认为 `get_memory` 函数中的形式参数 `p` 成功的申请了一段内存，但是由于 `p` 在函数调用时相当于 `pointer` 的一个拷贝副本，因此对于 `p` 的操作不会影响到指针 `pointer`。但是当所谓的申请内存函数调用之后，那个 `pointer` 的副本就消失的无影无踪了。不幸的是到这里问题还没有完，因为在 `get_memory` 函数中从系统中申请了一段内存，当函数 `get_memory` 调用结束后这段内存显然是无法释放的，因此还造成了内存泄露。因此使用上面 `get_memory` 函数的申请动态内存的方法是不可取的，希望同学们就这个样例进行一下完整的分析，彻底分析造成这种现象的具体原因。只有通过对事物认真的分析才能有更加深刻的认识。虽说将 `get_memory` 函数进行一定的修改就可以达到预期的效果，不过显然这样做显得多此一举，不如在主函数里直接用 `pointer` 指针承接一段动态内存。

实际应用中，申请动态内存可能出现不成功的情况（比如系统中可用内存确实少的可怜），此时承接申请动态内存的指针会返回 `NULL`，因此在写项目时要注意对这个指针进行判断。如果其指向是 `NULL` 那么就需要特殊处理。

在 ACM 比赛中一般不会使用动态内存，动态内存的申请需要耗费一定的时间，而且由于将动态内存释放时程序可能不会将内存立即还给系统。在需要多次测试数据的 ACM 竞赛中使用在每组数据测试时都开辟动态内存就有可能造成内存超出限制，当然如果在所有输入数据之前开辟一个动态数组那么还不如直接开辟静态内存方便。尽管如此，对于动态内存的申请还是要学会的，毕竟以后的工作绝不是让你在那里去刷题。

### 10.3 数组与指针

数组和指针的关系是非常紧密的，因此我们将与数组有关的一部分内容在这里与指针一起讨论。其实数组的名字就是一个指针常量，因此数组可以办到的事情用指针都能实现。但是要注意数组名字是指针常量，常量的值我们是无法改变的，因此尝试给数组名字进行自加运算或者自减运算是不可行的。对于数组和指针的一般只需了解到二维数组与指针间的关系就可以了，因为实际上超过二维多维数组的使用不多。

其实数组和指针也有一些其他方面的差异，这些差异如果同学们有兴趣的话可以查阅相关的资料了解一下数组与指针的具体关系。有关数组很指针的问题实际上有很多，并且数组和指针之间的关系是很微妙的，所以这里无法一一叙述，因此就需要同学们自己努力了，有时间可以多翻阅相关的资料。

### 10.4 字符串与指针

实际上指针与字符串的关系也是非常紧密的，比如在 C 语言中输入字符串时我们只需知道存储字符串的字符数组的首地址就可以实现输入一个字符串；字符串的处理函数实现都是使用指针来传递参数，可见字符串与指针的关系还是比紧密的。

使用指针的过程是需要慎重的，比如我们可以通过定义一个指向字符串的首地址的指针来实现对字符串的修改。但是这个指针的功能是非常强大的，甚至有能力强到修改一个字符串常量，因此我们使用指针操作字符串的时候要注意。使用另一个指针指向字符串并对字符串进行操作可能会带来一些不安全的因素，指针的使用过程中必须被严格的检查其使用的安全性和正确性。

## 10.5 八到十章的总结

其实这三章的内容联系是比较紧密的，而且涉及的东西非常的多，而这里所写的东西实在是太有限了，而且确实编者在这些方面写不出太多东西，所以只写了点相对基本的东西。其实这部分内容往往也是一本教材中最难写的部分，因为 C/C++ 的精髓部分在于指针，而指针、数组和字符串的关系还是非常紧密。内容的深度很难把握，所以这部分也会让编写相关资料的人感到头疼。

但是对于指针为代表的这一些列问题我们是躲不掉，因为这些东西是一个工程中必须具备的。所以对这些问题勤思考勤练习是最明智的选择，只要对指针和数组的问题有了一定的了解就不会感到难了，尽管基础内容掌握后到熟练应用还有一定的距离，但可以肯定的是只要肯下功夫指针和数组绝对不是什么大问题。我们总是能听到那些不愿付出或者急于求成的 C/C++ 程序员抱怨说指针不安全、太难驾驭，其实造成这种现象的原因就是程序员自身在这方面的基本功不扎实，编写代码能力不强造成的。从现在开始勇敢的面对指针，打好基础并通过后续的学习中的努力，这样才能真正的随心所欲的使用指针。

## 第十一章 学习路线

本章的内容完全是属于建议性质的，因为一套方案并不是对于每个人来说都合适，但编者还是希望同学们能认真的思考一下本章的内容。对于所有的事情都应该从最简单的最基础的地方开始做，搞 ACM 也是不例外。最开始接触 ACM 应该去刷水题，前面我们提到过做水题不但可以巩固和提高 C/C++ 的编码能力还可以对 ACM 及在线评测系统更快的掌握。对于多数人刚刚接触 ACM 一段时间（这段时间可能比较长）都是处于刷水题的时间，编者认为大多数情况下刷 150 多道水题应该对于 ACM 在线评测系统和基础的 C/C++ 掌握应该是差不多了（练习语言基础课后习题也不能马虎，最好有时间实现一些库函数，比如一些数学计算函数和字符串处理函数）。

当 C/C++ 的基础有了一定的积累，那么就可以接触一下数据结构了，数据结构是一个程序员必须会的，不论以后从事什么方面的编程，数据结构是必修的。编者在网上认识了一个研究生大哥（很文艺的一位同志），他对编者说不论以后搞什么东西（指的和计算机编程有关，也包括程序竞赛），内功一定要修炼好，数据结构正好就是这位大哥所说的内功部分（也包括算法），只要内功深厚就不怕什么了，只要是程序开发都是需要这些内功的支持。所以修炼内功时一定要认真，绝不可以偷懒。一个优秀的程序或者算法是需要一个优秀的数据结构来支持的。因此经过水题的“洗礼”之后就应该努力学习数据结构了，在这同时应该适当的看一下有关 C/C++ 比较深入的内容作为辅助。当我们基本数据结构学习完之后（要求要能用代码较为熟练地实现数据结构）就可以进行有关算法的学习了。编者是在没有什么数据结构的基础下学习的算法，结果学的很吃力，发现很多题目必须掌握树或者图这种数据结构才能做，所以又回头补习数据结构。这个过程绕了很大的弯路，所以同学们对待数据结构的态度一定要认真，如果仔细研究一下你会发现数据结构其实很有趣的。

确实在算法的学习过程中需要数据结构的基础作为支撑，对于算法的学习集训队会对新队员进行统一的培训，在一段时间内应该是属于某种算法的专题。这段时间内同学们应该针对所讲的算法进行重点的练习，一般负责讲课的学长们会布置一些题目供练习。一般来说严格的按照集训队布置的进度来练习是没有问题的（每个专题的时间都足够同学们去练习的，只要肯付出不偷懒就没有问题），你的水平会有很大的提高。算法不但是对 ACM 是有用的，而且算法的思想会影响到一个程序员处理问题的方式，对程序员的好处是很大的。

另外假期是提高个人能力最好的时期，因为这个时期没有作业和考试的困扰，有足够多的时间来学习。这段时间很适合看书和练习，所以假期的时间是绝对不能够荒废的，如果荒废了一个假期，等到开学回来的时候就可能被别人远远地甩在后面了。因此在放假之前可以向集训队的老队员请教，帮助制定假期中的学习计划，之后需要做的就是严格按照计划执行。如果有空余的时间可以查阅有关学过的知识更深入的资料，看看世界上各位大牛对这些知识的见解，看过之后可以丰富自己的知识。

还有就是当遇到做不出来的题目不要先去想我的代码哪里出了错误，而是去想我的思路是不是有问题；不要去向大牛们要代码，而是要去了解这道题的思路和思路为什么是正确的，长期阅读别人的代码而对主要思想置之不理往往会僵化思维的；不要让人给你灌输知识，灌输知识的老师不是好老师，尽管表面上看起来你的进步很快，看看养殖场里的速成的肉鸡吧；同理，只提供参考代码的资料不是好资料，代码模板有一大堆，ACM 比赛会让你携带数量不限的纸质资料，如果你愿意可以搬来一屋子的资料，但是这些资料中的思想不属于你，程序员需要做的是了解解题的思路，自己去用代码实现。如果这一段中提到的东西你能深刻的记住并付出实际行动，那么你的能力将会更上一层楼。

总体来说上面的一个基本路线是适合同学们的一种基本的路线，但是具体的计划实行会由较大的差异，毕竟每个人的能力是不同的。同学们应该扎实的走好每一步，尤其是现



在的入门阶段，只有扎实每一步最终才能取得一个让自己满意的结果。一味的追求进度是没有用的，对于学习速度的把握要根据自己的实际情况来进行。在经历了不懈的努力后，希望各位能成为一个 ACM 的大牛！

具体知识点可以参考我校集训队编写的 ACM-ICPC 培训资料汇编，内容涵盖了参加 ACM-IPPC 竞赛所需的大部分知识点，建议顺序阅读，在掌握入门知识后再深入扩展学习和运用，注意培养运用这些知识的思维训练。

## 第十二章 ACM 新手常见基本问题集锦

### 12.1 评测结果与评测环境

#### 一、一份代码在交到系统后，它是如何进行评测的？

这部分内容大家不必理解的很深，因为完全理解它的评测机制还需要关于操作系统方面的相关知识，就目前大家的知识储备来说，只要了解它大致是一个怎么样的流程即可。我一直也认为，一旦你能了解它的评测机理，那么对于 AC 题目也有一定程度的帮助，很多原理性的东西在你不清楚它到底是怎么回事的时候，往往都是别人跟你说该如何如何做，你就照做。但如果你自己明白它的原理，你就可以凭借自己的思考进行创造，所谓举一反三，有些类似这样的道理。啰嗦了这么多现在开始进行正题。

评测程序：文中的评测程序相当于一个法官，评判官，来评判你的程序是对是错。

编译：这里说的编译其实包含了编译和链接两个过程，因此当我引用到“编译错误”时你应该知道它代表的是编译错误和链接错误中的一个。

标准输入/出：标准输入默认是键盘，标准输出默认是屏幕。

重定向：这里专指输入输出重定向（IO 重定向）。当指向 IO 重定向到文件时，即可将输入由键盘改成读取文件，而将输出由屏幕也改为文件。

首先要知道，对于每一道题目，在我们的 OJ 系统上都有 2 个文件与之对应，它们分别是 data.in 输入数据和 data.out 输出数据，这两个文件是评测你的程序对与错的关键。你的一份代码交到 OJ 上后，评测程序会将你的代码写到一个文件，比如 main.c 之中，接着评测程序会用编译器对这个 main.c 进行编译，此时，如果该代码能通过编译，则可生成一个可执行程序，我们起个名叫 main.exe，但假如此时编译失败了，则直接返回结果 Compile Error。对于成功编译的情况，评测程序会运行生成的 main.exe 并将它的标准输入重定向到 data.in 文件，标准输出重定向到 out.txt 文件。这样你写的程序会根据输入文件 data.in 中的内容计算出一组结果并输出到 out.txt 这个文件中。在 main.exe 运行过程中评测程序会“监视”它，看它有没有超过规定的时间限制，内存制限，调用非法函数，运行时错误，这些只要有一个发生则停止程序返回相应结果分别是 Time Limit Exceeded, Memory Limit Exceeded, Restricted Function, Runtime Error。如果以上情况均未发生则 main.exe 正常终止。于是评测程序会对 out.txt（你程序跑出来的结果）和 data.out（标准答案）就行比对，如果完全一样则返回 AC 的结果，否则看是否多了空格回车之类的来区别 PE 还是 WA。这便是整个评测流程了。

#### 二、64 位整数的格式控制符到底是%lld 还是%I64d？

这个问题要从评测环境讲起，如今的 OJ 基本都建设于两种操作系统基础上，其一为 windows 系统，另一个就是 linux 系列。在 windows 系统上我们该使用%I64d 来控制，而在 linux 系统上应该用%lld 控制。其原因是编译器的不同导致，这里不必深究。至于定义一个 64 位整数则应使用 long long 来定义，有时你可能会看到如\_\_int64 这样的定义，这只在 VC 编译器下有效，对我们正规比赛用的 GCC 编译器是不行的，因此强烈建议此处只用 long long 定义 64 位整型，格式控制依 OJ 系统而定。典型的 OJ 中，我们学校、POJ、ZOJ 都是用%lld。HDU 杭电上的要用%I64d。

#### 三、Compile Error 一般是如何产生的？

CE 这一项其实包含了两方面内容，一是编译错误，二是链接错误。大家一定区分这两个过程并不是一回事。

编译是由源代码产生目标代码的过程，即由.c/.cpp 文件生成.obj 目标文件，目标文件已经接近可执行文件了，只不过缺少一些链接库。

链接是由目标文件产生可执行文件的过程，即由.obj 生成.exe。链接将程序所需的静态库和目标文件合在一起生成一个可执行文件。

这里提到的静态库其实就是一些标准库函数的代码，你能见到的.lib 或.a 文件都是静态库文件，与之对应的还有一种叫动态库的东西，我们在 windows 下常见的.dll 文件就是动态库。它们本质上都是提供函数的实现发式。

现代的编译器比如 GCC，通常都是将编译和链接统一在一起，即 GCC 既可以编译，也可以链接。但其实编译和链接是两个过程。这里清楚是两个过程就好。

介绍了关于编译和链接的东西之后，那常见的此类错误有哪些呢？下面说说

#### 1. 变量未定义就使用，或定义后使用位置超出了定义的作用域

说明：如 b=25 但程序并未在使用前定义好 b 的类型，还应注意变量作用域。

#### 2. 括号不匹配

说明：()、[]、{} 一定要有始就得有终

#### 3. 使用非法类型

说明：如 \_\_int64 在 GCC 编译器下不识别，你却用它

#### 4. 需少分号

说明：忘了语句以一个分号结束

#### 5. 用函数未引头文件

说明：如 printf 函数在 G++ 中未包含 stdio.h 文件。GCC 较 G++ 稍宽松些此处。

#### 6. 包含非标准头文件

说明：如 conio.h 并非标准库中定义的头文件

#### 7. 使用非标准库函数

说明：如 itoa、strupr 等并不是 C 标准库中的函数，不能使用

#### 8. 有非法字符

说明：一般是你网上或幻灯片里粘贴过来的代码有非法字符引起

#### 9. 自定义函数未声明

说明：自定义函数一定要先声明再使用

### 四、Presentation Error 是由哪些字符引起的？

这个错误和 CE 错误我觉得是最不该发生的。PE 错误通常是由于空白符的多或少输出导致。所谓空白符是指如下的几个：' \r'，' \n'，' \t'，' '，它们分别叫回车、换行、水平制表、空格。也就是说你的程序多输出或者少输出这几个字符就会产生 PE 的结果，出现 PE 基本可以认为你的程序是对的，只要稍加调试这些东西就可以 AC 了。当然针对那些让你排版的题目除外，如 POJ-1093。

## 五、Time Limit Exceeded 是超时，该如何避免呢？

超时的一个主要原因是算法时间复杂度太高，此时应考虑选用合理的数据结构降低你的程序的时间复杂度，或者更换算法。我们做过的每一个题都有一个时间限制，通常是 1000MS，即 1 秒，也有 2 秒，5 秒，10 秒的题。而每一个题的数据范围题目会事先说明，比如  $n \leq 100000$ 。假如你的程序有一个 for 循环进行  $n$  次，那它运算次数的数量级为  $10^6$ ，又如你的程序有两层嵌套的 for 循环，每层都循环  $n$  次，那它运算次数至少为  $n*n$  即  $10^{12}$ 。一般的计算机来说 1 秒内的运算次数为  $10^7$  到  $10^8$  左右，那么显然循环  $n$  次的算法可以在 1 秒内跑完，而循环  $n*n$  的算法不可能在 1 秒内跑完。所以通过预估你的程序的时间复杂度，加上题目给的数据范围和时间限制，你就大概能判定你的程序是否超时了。更多的关于时间复杂度的理论，请自行参考《算法导论》。

## 六、Memory Limit Exceeded 超内存，怎么避免呢？

通常一个题目超内存的可能性不大，一般的题目对内存的要求并不是很严格，所以出现超内存的情况还是很少的。但并不是没有，出现超内存时我们需要对自己的程序的空间复杂度进行优化，此处的空间复杂度是与时间复杂度相对应的，你可以在算法导论中查看它具体的定义。避免的方法只能是跟据题目所给出的数据范围，看一看数组开辟的能不能再小一些，或者更改算法以使用更小的内存。

## 七、RunTime Error 运行时错误产生的原因都有哪些？

所谓的运行时错误，重点在“运行时”这个词上，什么叫运行时呢？通常一个程序被写好后通过编译链接成一个可执行文件后它是存放在硬盘的，此时的可执行文件仅仅是一个静态的文件，它并没有被运行。当我们要执行一个程序时通过双击启动该程序，接下来的工作是由操作系统将可执行文件利用装载器装载到内存中，操作系统同时为这个程序创建一个进程实体，这个过程实现了可执行程序由原来在硬盘中静静地呆着状态，变到在内存中要动态执行。一旦该程序加载到内存中并且操作系统为它分配 CPU 时间了，这时的程序就叫正在运行，假如程序在执行的过程中出现了问题，一般的具有保护性的操作系统都会将它直接杀死，也就是结束进程。简单的说运行时错误就是程序在运行时出现的错误。它又分为几种：

1. Floating Point Error，这是因为你的程序出现了除法运算中除数为 0 的情况
2. Segmentation Fault，段错误，引发段错误的原因主要的又可以分为 2 点：

(1) buffer overflow 缓冲区溢出，缓冲区溢出一般就是你的程序中数组开小了，产生了越界访问，比如定义 `int a[100]`；此时你引用了 `a[1000]`就很可能出现运行时错误。“缓冲区”一般指的就是程序中定义的一个数组，这片连续的内存空间用以存放一些要处理的数据。值得一提的是缓冲区溢出漏洞就连成熟的程序员都有可能不小心忽视它导致巨大的损失，所以现在写代码一定要养成深思熟虑，考虑好每一个值的范围，以防越界。之所以存在缓冲区溢出漏洞，一个主要的原因是 C 语言并不对数组下标进行界限检查。

(2) stack overflow 栈溢出，俗称的暴栈。栈这个东西在操作系统中通常用来维护一个函数的调用，C 语言中在调用函数的时候会依赖一个“栈”这种数据结构的

性质的内存。如果你细心可以看到在 CodeBlocks 的调试当中有一项叫“Call Stack”，此即“调用栈”，操作系统为每一次函数调用时创建一个“栈帧”，它记录了函数的参数，函数返回地址，以及当前进程上下文环境等内容。当函数调用层次很深时，尤其是深度调用的递归函数，极为可能引发“栈溢出”这个运行时错误。原因是每次递归一层函数时，系统都要建立一个新的栈帧给该层调用的函数，深度递归时只调用而不返回，那么就会一直利用“有限的栈空间”来创建很多的栈帧，这样总有一个时间会把这个栈给挤暴的。通常栈的大小为 8K。

综上所述，引发 RE 的原因主要有“除 0”，“数组越界访问”，“递归层次太深”三个。

## 八、可能由数组越界访问引发的超时问题。


这一点一定要引以警醒，当你的程序 TLE 的时候，先考虑这个问题，我程序的数组有没有开小？如果你的数组开小了极有可能引发 TLE 这个结果。原因如下，请先看段代码：

```
#include <stdio.h>
int main()
{
    int i, a[10];
    for (i = 0; i <= 10; i++) {
        a[i] = 10 - i;
        printf("%d\n", a[i]);
    }
    return 0;
}
```

先分析一下程序是干什么的，有没有什么问题。直观地看此程序是输出 10, 9, 8...0. 再仔细看的话，你会发现 a[i] 这个数组竟然访问到了 a[10] 这个元素，可是定义的 a[10] 对 a 数组元素进行访问的合法范围是从 a[0] 到 a[9] 啊，循环中对 a[10] 进行了赋值 0 的操作，这将有什么影响呢？试着运行一下这段程序，你会发现它的结果是死循环！

问题来了，为什么会死循环呢？我仅仅访问了一个 a[10] 就死循环了？一个初步的猜想是 a[10] 也是有一个内存空间的，转换成更加直接的指针写法是 \*(a+10)，即以数组首地址 a 为基址，偏移地址 10 个单位的那个地方。那个地方存的是什么呢？存的这个东西到底是什么可以让我给它赋成 0 之后循环就变成死循环了呢？再猜，一个想法就是 i 这个变量，即 a[10] 和 i 就是同一个东西，它们指的就是同一个内存地址。OK，按照这个猜想，我们不妨去看一下，如果 i 和 a[10] 是内一个存储空间，那它们的“内存地址”一定是相同的。这样在循环前面加上一句 printf(“Address of i is %p\nAddress of a[10] is %p\n”, &i, a + 10) 把这句话加上，再把 for 循环里的 printf 语句删掉，我们运行一下看一看是什么结果？

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int i, a[10];
7      system("title Test for Buffer Overflow");
8      printf("Address of i is %#p\n", &i);
9      printf("Address of a[10] is %#p\n", a + 10);
10     for (i = 0; i <= 10; i++) {
11         a[i] = 10 - i;
12         //printf("%d\n", a[i]);
13     }
14     return 0;
15 }
16
```



这个结果证实了我们的猜想，即 `i` 和 `a[10]` 的确是占用着同一个内存地址，它们是“一个人两个名”而已。

现在你应该能解释为何产生死循环了，因为数组的越界访问，而恰好越界的那个 `a[10]` 就是 `i` 的那个位置，我对 `a[10]=0` 的操作就相当于 `i=0` 的操作，因此 `i` 自增到 10 后又被赋成 0，重新开始了循环，每一次 `i` 到 10 都被变成 0，循环也就无休无止的进行下去。

这个例子很巧的体现出了，数组越界不一定是 RE 还有可能是 TLE。你可能会想，这也太巧了吧，实际的时候能这么巧吗？其实并不巧，不巧的原因是连续定义的变量它们会占用连续的存储空间，即你 `int a, b, c, d, e`; 这五个变量在内存中都是挨着存放的，你完全可以用 `*(&b+1)` 去访问 `a` 这个变量，用 `*(&b-2)` 去访问 `d` 这个变量。

通过这个例子你应该知道的是数组越界不一定 RE，可能 TLE。同时还应知道的是相同类型的连续定义的变量它们在内存中占用的是连续的地址空间。对于不同类型可能涉及到内存对齐问题，这里不进行讨论。

## 12.2 C 语言常见编码问题与技巧

### 一、浮点类型的 double 与 float

两者区别一个精度较高一个精度较低，`double` 类型的精度可以达到 15 到 16 位，而它所付出的代价就是需要用 8B 来存放一个浮点数，`float` 类型精度可达 6 位，它只需 4B 空间存储一个该类型的变量。通常来说我们应尽可能使用 `double`，并且尽可能不使用 `float`。其原因是虽然在一个问题面前你可能已知了它的精度是多少，但难免会出现过多过少的偏差，况且一个程序的执行不光光依靠理论依据就一定正确，一个程序的执行还受操作系统，硬件环境，运行环境等诸多方面的因素影响，因此难免可能遇到你理论上证实正确无误，不会超出 `float` 的精度，但实际上可能会出现问题。而这种问题也是你无法预料的，因此为防止因精度问题产生的重复提交，建议浮点类型都用 `double` 定义。`double` 定义变量虽然内存上占用稍大些，但一般的问题都不会卡这个的。

还有一种 long double 类型是 C 语言新标准所添加的内容，它这种类型所占用的内存大小标准并未归定，因此不同编译器实现的时候对此设定的值不同，一般 GCC 来说定的是 12 位，其它的还有 8 位和 16 位的。其格式控制符为 %Lf。

## 二、浮点类型的格式控制符

有关浮点类型的格式控制符，以 double 定义的浮点类型，用 scanf 读入变量时应使用 %lf 控制，以 printf 输出时应使用 %f 输出。其出处是 C99 中制定的方案，对 printf 中的 %f 会有一个 C 语言中的“默认参数类型提升”过程。详细可以参考 C99 手册。

## 三、数组的定义位置

你在编码中可能会遇到在局变范围定义一个较大的数组，比如 `int a[10000000]`；当我把它写到 main 函数中的时候运行可能会出问题，原因是局部变量都是定义在栈内存中的，一个程序的栈内存很有限，当数组要求开的很大时，我们通常都是将它定义为全局变量，全局变量的内容存放在内存中的全局区，与栈区不同，该区可存放较大的内容。所以对一般的大数组我们通常都是将它定义为全局变量。

这里又不得不提一点，我们搞 ACM 竞赛做题到底还是和工程上搞开发做项目有一定的区别的，在工程项目中，对于大容量数组，一般都是采用动态内存分配，而对全局变量是尽可能少用，甚至不用，全局变量的引入会影响程序各模块间的亲密度，一个良好的程序设计方案中应尽量减少全局变量的使用，提高各模块间内聚性，减少各模块的耦合。而现在了解一些，以便在实际应用中区分好竞赛和工程应用各自适合哪些。

## 四、常数的存储类型相关问题

默认情况下，一个整数常数是以 4B 来存储的，举个例子：`x = 1 << 30` 该表达式将整数 1 左移 30 位并将结果赋值给 x。对于 1 这个整数来说，它在内存中是用 4 字节来存放的，相当于是一个 int 类型，同时它还是一个 signed int 即有符号的整数 1。现在假如我想将 1 左移 50 位赋给 y，如何写？这样写：`y = 1 << 50` 可以吗？当然不行，前边已经说了 1 默认是用 signed int 一个 4B 大小的内存空间存放这个 1 的，当 1 左移 32 位时候就已经达到了它的尽头，再往后移，剩下的只能是 0 了。但是我又想让它移 50 位正确结时给 y 该怎么办呢？C 语言中用加上后缀即可，此处可以写成 `y = 1LL << 50` 这意思就是将 1 用一个 long long 的 8B 空间存储，再将它左移 50 位，由于 8B 整数能存放最大  $2^{63}-1$  的数，因此足够了。与 LL 类似的定义常数的类型的还有 U、UL、ULL 分别代表 unsigned int, unsigned long int, unsigned long long int。这个问题别看不起眼，确实在实际用的时候有过因为没加 LL 而 WA 的题



## 第十三章 Linux 使用简介

### 13.1 Linux 简介

#### 13.1.1 历史与发展

如今的操作系统主流分两大类别，一类是我们经常使用的 windows 操作系统，由以 Bill Gates 为代表的微软公司研发，最早的一个版本发行于 1985 年；另一类就是 Unix 操作系统，最早由 KenThompson、DennisRitchie 和 DouglasMcIlroy 于 1969 年在 AT&T 的贝尔实验室开发。而我们平常能听到的 Linux 操作系统其实是属于 Unix 系统中的一个分支，就连苹果的 Mac OS 也是基于 Unix 内核开发出来的。Windows 我们且不提，我们用它用的太多了，以至于在使用方面几乎对它都没有什么问题了，这里主要介绍一些简单的 Linux 方面的知识，以及其用于程序开发的使用方法。

Unix 系统的分支系统非常多，但为何 Linux 现在如此主流，一个主要的原因应该归属于它是开源软件，开源软件的好处是任何人都可以下载它的源代码，并且修改之，然后他可以再次发布出来让其它人修改。这样在全世界众多优秀的程序员共同努力之下，Linux 系统不断被完善，它的性能也不断地提高，俗话说众人拾柴火焰高，团结的力量就是大，正因如此，Linux 才能在众分支中脱颖而出，成为在服务器架设、程序员深造的优秀操作系统。

Linux 的创史人是林纳斯·本纳第克特·托瓦兹（Linus Benedict Torvalds），这也是此操作系统命名的由来，使用的是作者 Linus 之名。平常所说的 Linux 操作系统实际指的是 Linux 内核，众多的 Linux 类操作系统又分好多，诸如我们最常用的 Ubuntu，还有 RedHat, CentOS, Debian（手机的 Android 系统是基于 Linux 的）等等，这些属于 Linux 类的操作系统，它们有一个共同特点就是它们的内核都是 Linux 的内核，但是除去内核后剩下的东西，各自有各自的特点，比如 Ubuntu 更适合我们普通程序员用户，RedHat 更适合服务器架设。这里提到了内核(kernel)，与之对应的还有一样东西叫做外壳(shell)，那它们都是什么呢？这本是操作系统的知识，在此我不想用理论性很强的概念来描述，因为在没有更多的知识奠基之前，这样的叙述只会显得更加晦涩难懂。我们中国有句俗语叫换汤不换药，这个能很好地解释出 Ubuntu、RedHat 等发行版的区别。这“药”就是“内核”，众多的 Linux 发行版所使用的“药”都是一个，就是 Linux 内核，但另一方面这“汤”可就不同了，比如 Ubuntu 这碗药里我加的是甜一些的汤，而 RedHat 这碗药里我加的是咸一些的汤，因此“表现”出来的效果就是看上去它们不太一样（包括外观啊，应用方面啊），其实它们核心都是一个“药”，都是 Linux 内核，内核不变，它们的主要功效就不变。这也像我们每个不同的人，人都有五脏六腑，这些是核心，它们在每个人体内功能都是一样的，但每个人长相又不同，因此就产生了 Ubuntu、RedHat 等等分支。

在此还要提一点，由于 Linux 是基于 Unix 开发而成，而 Unix 是由 K&R 两位 C 语言发明者研发而成的，在 C 语言中是区分大小写的，由此，在 Linux 中的任何东西也都是区分大小写的，这一点与 windows 不同（windows 不区分大小写）。

#### 13.1.2 Linux 中的用户

众多的 Linux 发行版在此不可能面面俱到，因此本文就能够用着方便，且为以后比赛做准备的方针，选用 Ubuntu。在此仅声明以后任何操作均在 Ubuntu 上进行，但后文所述的主语仍是 Linux，大家只要知道这里的 Linux 都指的 Ubuntu 发行版就可以。其实大多数在 Ubuntu 上能够进行的实践操作在其它的发行版也一般也都可以。

Linux 中的用户大体上分成三类，围绕着用户展开的话题一般都是“权限”的不同。以平时常用的 windows 系统来先说说用户，我们用的 windows 在安装后都建立了一个自己的用户，通常这个用户都是具有管理员权限的，管理员的权限可以让你对本机做许多操作，与管理权限相对应的另一类是没有管理员权限，这类的用户通常只能简单地使用计算机，而不能进行安装，卸载等重要的操作。

Linux 下的用户的三类是这样分的，第一种是超级管理员，每个安装后的 Linux 系统都

有一个超级管理员用户，名字叫 `root`，第二种是普通管理员，这种用户一般是在安装过程中程序询问你创建的一个个人用户，第三种就是外来客人用户，这种是最低级用户，通常啥也做不了，只能进行有限的操作。这三种用户的权限级别是超级管理员>普通管理员>客人用户。

`Root` 这个超级管理员可以做任何你能想到的事情，比如删除系统文件啦，修改下系统文件之类的它都能做。它拥有整个系统中最高的权限。

普通的管理员是自己创建的用户，虽然它也是管理员，但相对于 `root` 来说它能做的事情就要少多了，普通管理员可以安装卸载软件，或删除一些不是很重要的文件，但对于系统重要文件来说，这种类型的用户是不能进行此操作的。在一定程度上受到了限制。

客人用户一般只是看一看电脑上有啥文件，简单地用一用计算机，基本上做不了什么事情，就连关机的权限它都没有。

`Root` 用户可以和你自己建立的用户进行切换，具体切换方法不在此叙述，将在后文件需要的地方给出。这里大家可以连系到手机 `Android` 系统，你要听说过手机 `root` 之类的字眼，现在应该很明白了，其实就是要从普通管理员用户切换到超级管理员 `root` 这个用户，以获得更高的权限来做更多的事情。

### 13.3.3 Linux 目录结构

目录结构这东西说白了就是系统有哪些目录以及它们的作用都是什么。就拿 `win7` 先说，安装后的 `win7` 一般都会有“`windows`”、“`用户`”、“`Program Files`”等，“`windows`”存放系统文件，“`Program File`”存放安装好的程序，“`用户`”存放用户配置文件。对于 `Linux` 系统来说其目录结构与 `windows` 是完全不同的。

`Linux` 系统有一个最顶层的目录称之为根目录，用一个斜线“`/`”来表示。其下分若干子目录，如 `bin`、`etc`、`usr`、`root`、`home` 等，这些子目录各自有各自的功能，在此我不想过多的介绍每一个目录都是干什么的，这样做无非是增加了入门手册的长度，使得一些暂时无必要记住的东西也掺杂进来，不便理解。在此仅介绍与入门相关的，其余的东西如果大家感兴趣请自己百度搜索“`linux 目录结构`”，肯定会有一大堆另你满意的结果。整个 `Linux` 的目录结构如同数据结构中的“树”一样，请看下图。



刚刚说了“`/`”代表根目录，那“`/root`”就代表根目录下的 `root` 目录，这个目录正如其名一样，它就是 `root` 这个超级管理员的用户目录，一些 `root` 用户的配置文件什么的，都是放在这个目录中。记住 `root` 的用户目录是 `/root`。

另外一个，假如现在有一个普通管理员用户名叫 `zeropointer`，它的用户目录就创建在 `/home`，中，在 `/home` 下会有一个 `zeropointer` 用户目录，其完整的路径就是 `/home/zeropointer`。这就是 `zeropointer` 的用户目录。

这里介绍这两个目录的作用主要是用于区分超级管理员和普通管理员用户目录的不同，在以后讲到基本命令的时候还要用到的。

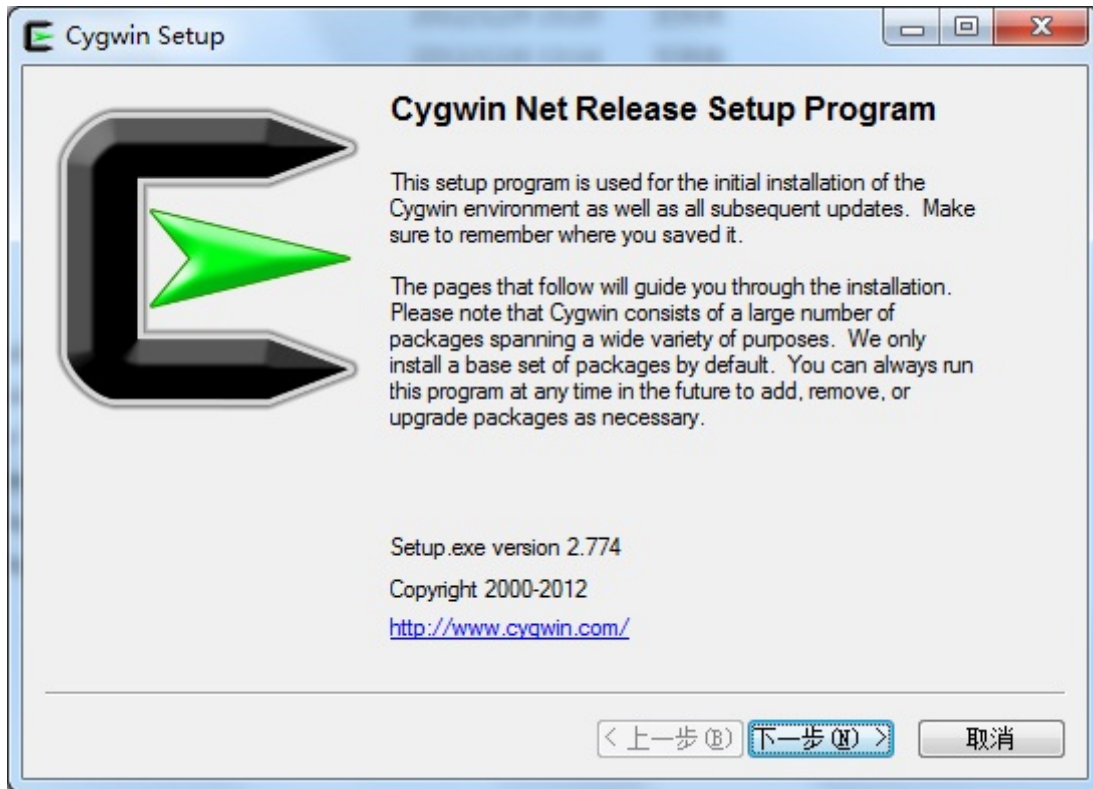
### 13.3.4 总结

基本的 Linux 知识就介绍到这里，大家应该清楚，这里进行的仅仅是入门的介绍，入门的话，我们一定会尽可能用简短的言语来叙述尽可能对入门级别有用的东西，因此，还有很多更详细的内容，不会在本文中出现，这就需要有深度求知的你自己去查相关的资料来学习，任何一本书都不可能面面俱到的讲到每一样知识，这就需要你多方面搜索资料，补充不足。对于 Linux 方面的学习，在此也介绍一本书《鸟哥的 Linux 私房菜》基础学习篇，此书对掌握 Linux 有很大帮助，如果你志愿今后从事服务器或者 Linux 相关的工作，建议可以先读此书。

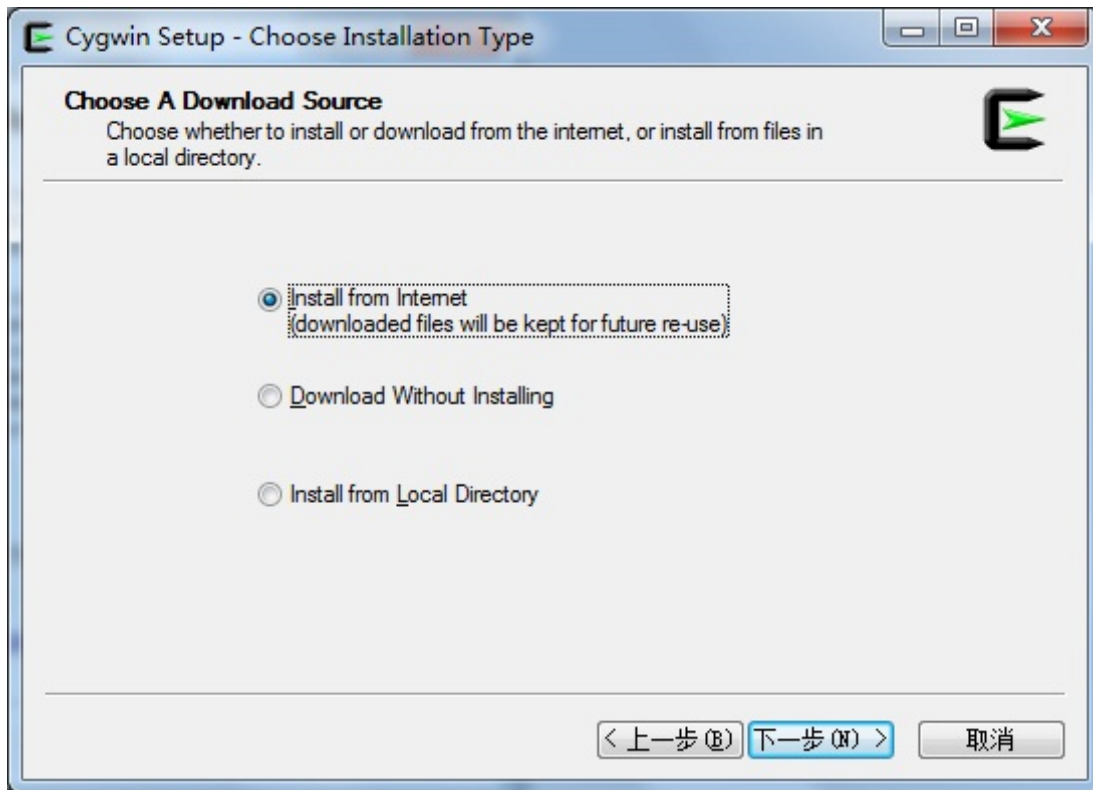
### 13.3.5 cygwin

cygwin是一个在windows平台上运行的unix模拟环境，是cygnus solutions公司开发的自由软件。官方网站是<http://www.cygwin.com/>。到这里下载一个叫做setup.exe的东西，就在主页上就可以看到，下载后的大写大约是几百KB。下载完成后运行它。按下面给出的图中所做的一步一步安装。

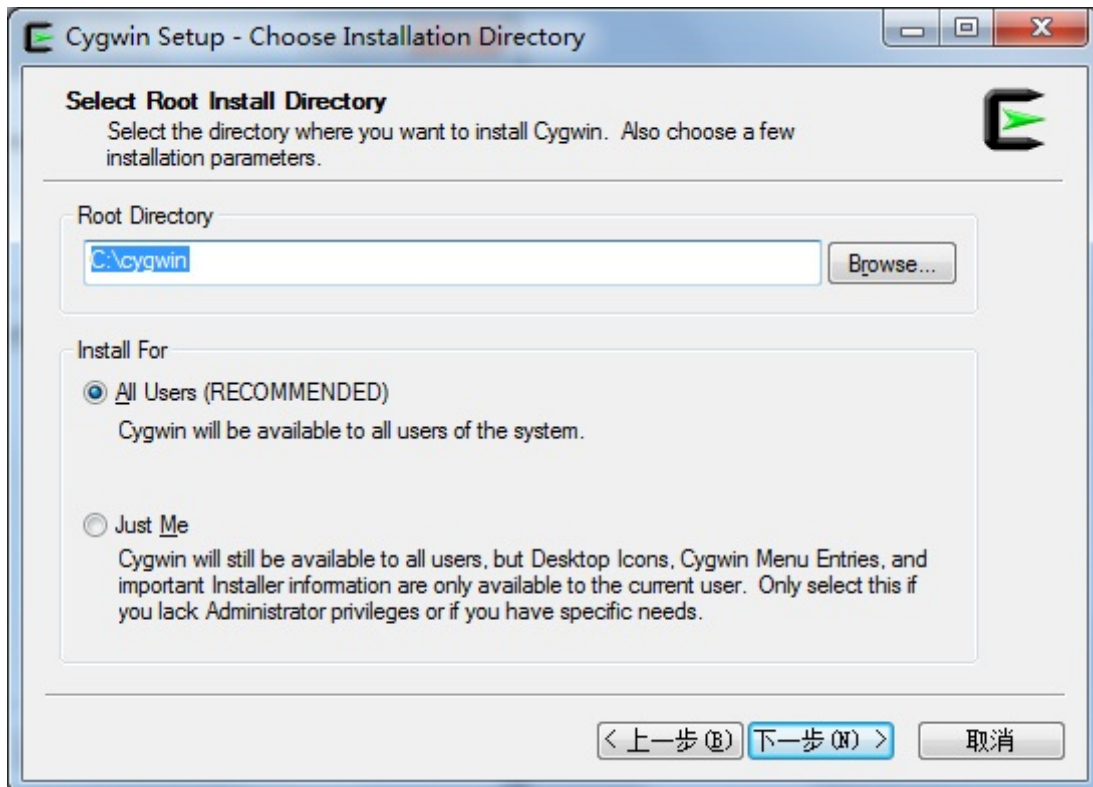
STEP 1:



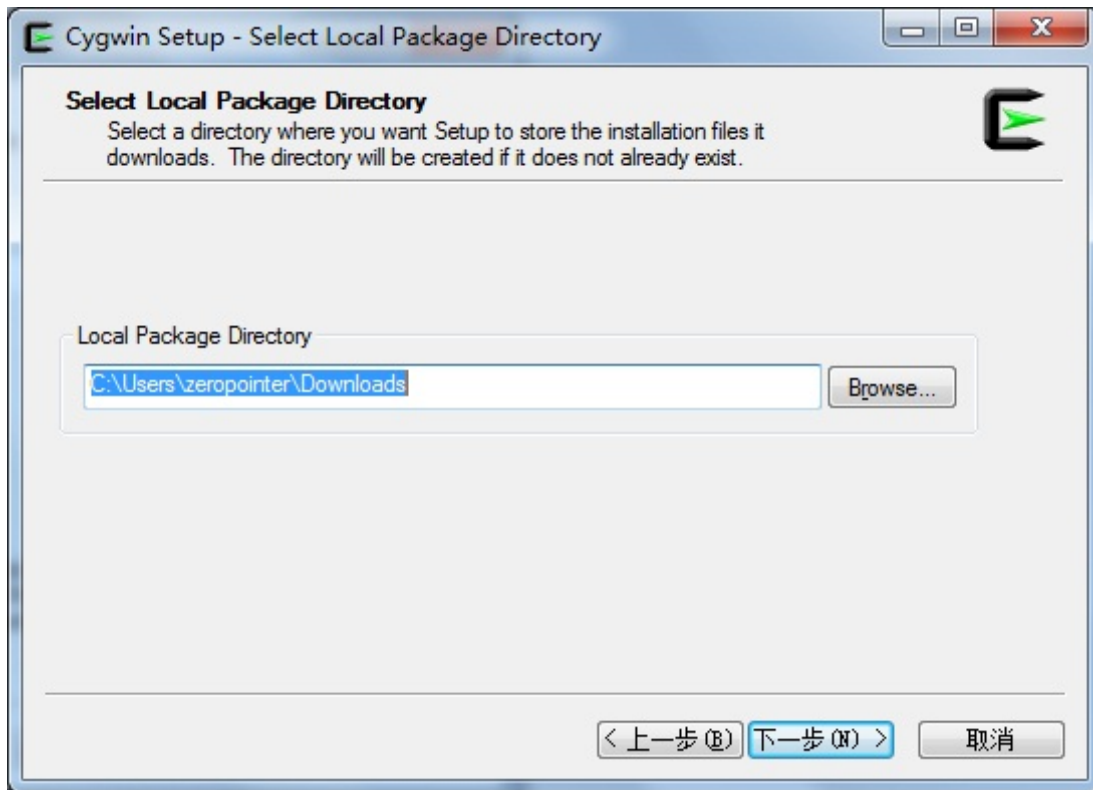
STEP 2:



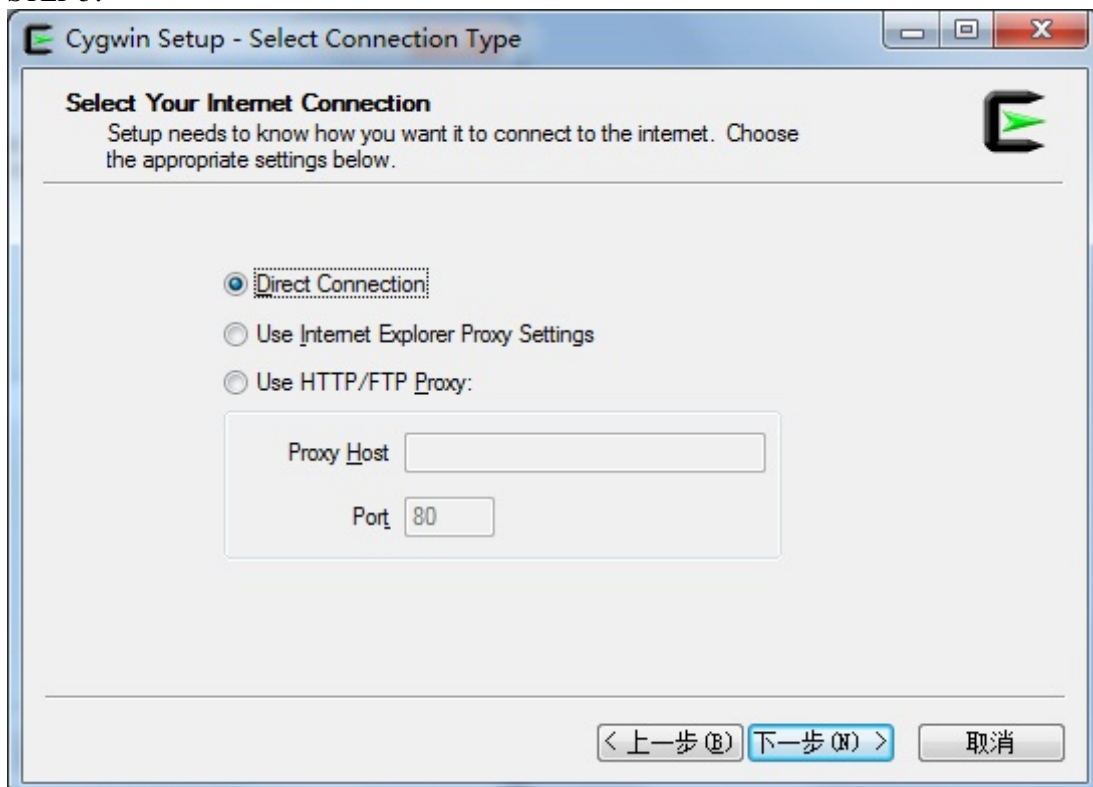
STEP 3:



STEP 4:



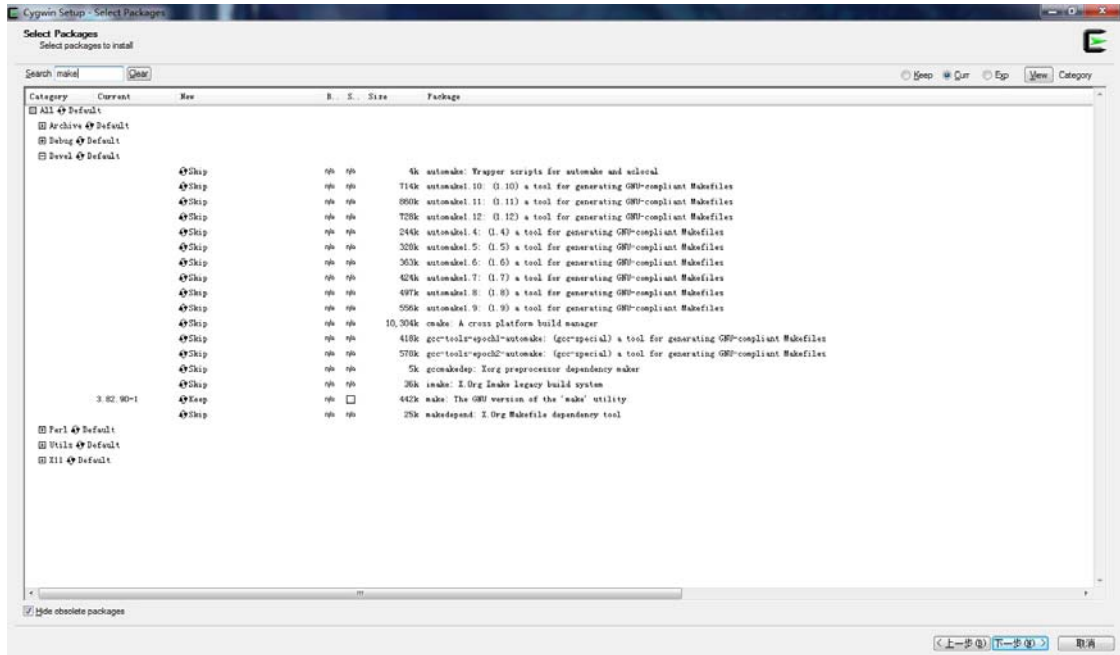
STEP 5:



STEP 6: 这里开始搜索要安装的软件，首先是 vim，按照图中所示。







## STEP 9:

最后一步点一下，它就开始从网上下载你选中的软件了，稍等片刻后就会安装完成。

## 13.2 Linux 下常用命令简介

### 13.2.1 引子

基本命令这里，针对咱们搞竞赛的需要只介绍常用的命令，Linux 下的命令很多，就算是看书看一遍你也未必能一下子就全都记住，记忆命令的使用方法最好的办法就是经常使用这个命令，常用就会熟练了。

在介绍命令之前需要先说一样东西，你看，咱们到现在说了半天的命令命令的，可是命令写在哪里呢？总不至于把那些命令都写到记事本里去吧？当然不是，Linux 下有一个叫终端的东西，它的英文名叫 **terminal**（记住它）。你打这终端这个应用程序（注意终端也是一个程序）后可以在其中输入各种命令，这就是交互。你输入命令告诉你的 Linux 干什么事情，终端接收到你的指令后就传给 Linux 的壳（**shell**）由这个壳程序来解释你输入的命令到底是什么意思，最后返回给你相应的结果。其实这个过程就是你和你的 Linux 在进行对话，不是吗？

OK，那终端在哪？如何打开它？现在这样做，在你安装好的 Ubuntu 操作系统的桌面上，什么也别做，现在按一下 **ALT** 键，在左上角的地方出现一个可以输入的框框，在其中输入 **terminal**，就是刚才让你记的那个东西，然后回车。这时终端就启动了，你可以发现它是一个背景是黑色的窗口，在其中有一个小光标在闪烁，这就是等着你给它输入命令呢！（如下图所示）



```
dragon@uvm1104: ~
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
dragon@uvm1104:~$ ssh -X oracle@192.168.0.222
oracle@192.168.0.222's password:
Welcome to Ubuntu 11.04 (GNU/Linux 2.6.38-8-server x86_64)

* Documentation:  http://www.ubuntu.com/server/doc

System information as of Thu Jul 21 14:17:55 CST 2011

System load:      0.14                Processes:         112
Usage of /home:   5.1% of 91.67GB     Users logged in:   0
Memory usage:     1%                  IP address for eth0: 192.168.0.222
Swap usage:       0%

Graph this data and manage this system at https://landscape.canonical.com/
Last login: Thu Jul 21 14:16:21 2011 from 192.168.0.14
oracle@bcserver4utest:~$ xclock
█
```

终端已经打开了，接下来就应该是输入命令了，OK 从现在开始介绍一些常用的 Linux 命令。

### 13.2.2 清单列表命令——ls

ls 命令可以列出当前所在目录下的文件列表，包括文件、文件夹、链接等。输入 ls 并回车，看一看结果如何？我在本地运行后的效果如下图：

```
[16:53:12 zeropointer@zeropointer-PC:~]:) $ ls
acm/  test.c
[16:53:15 zeropointer@zeropointer-PC:~]:) $
```

你看，我现在的所在的目录下有两个文件，其中 acm 是一个文件夹，test.c 是一个文件。它都给列出来了。如何区分文件和文件夹呢？一般来说通过 ls 命令显示出来的结果，如果它的后面有一个“/”字符，那就说明它是一个文件夹，而没有“/”字符的就是一个普通的文件，当然，你也可以看到这两样东西的颜色都是不一样的。一般情况下你也可以使用颜色不同来区分文件和文件夹。

Ls 的功能其实正如你在 windows 下进入某个目录后所看到的结果一样，它会把这个目录下所拥有的子目录和文件等列出来给你看。最后，ls 是 list 的意思。

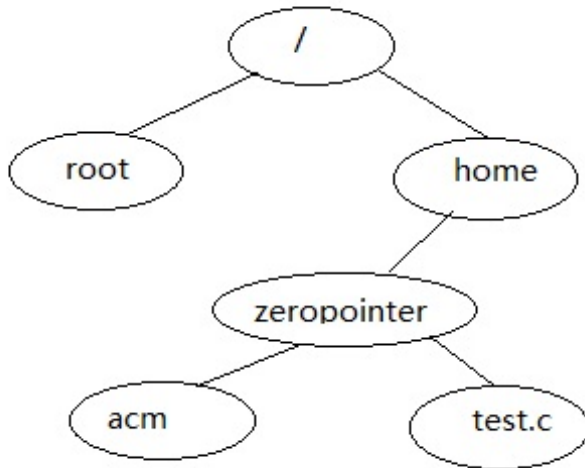
### 13.2.3 查看当前所在目录命令——pwd

刚刚的 ls 列出来当前目录下所拥有的文件，那我如何得知现在是在哪个目录下呢？使用 pwd 命令即可达到目的，在终端中输入 pwd 并回车，看一看效果如何。

```
[16:53:12 zeropointer@zeropointer-PC:~]:) $ ls
acm/  test.c
[16:53:15 zeropointer@zeropointer-PC:~]:) $ pwd
/home/zeropointer
[16:57:36 zeropointer@zeropointer-PC:~]:) $
```

通过 `pwd` 命令反馈给我的结果是 `/home/zeropointer`，说明，我现在（或者说我的 Linux 终端当前所在）的目录就是 `/home/zeropointer`。

等等，看着眼熟不？记不记得在上一节中我让大家记的根目录“/”和根目录下的一个子目录 `home`，`home` 文件夹中存放的都是各用户文件，也就是说每一个用户都会在 `home` 目录下建立一个与自己的用户名一样的文件，而这个用户名的文件夹里存的就是该用户的一些配置文件。就拿我这个例子来说吧，`zeropointer` 是我的用户名，我的用户在 `home` 文件下有一个 `zeropointer` 子目录，它的完整路径是 `/home/zeropointer`。而我现在用的终端所在的目录就是 `/home/zeropointer`。刚刚 `ls` 所见到的结果是有一个 `acm` 文件夹和一个 `test.c` 文件。OK 整理一下，现在的整个目录结构应该如下图一样：



注意一个问题，`pwd` 命令列出来的当前目录是从根目录开始的，这个你可以去试进入到一个其它的目录，无论你当前目录是啥一个，只要 `pwd` 命令，结果就是从根目录开始的。那么如何进入其它目录呢？这就小涉及到一个新命令，请看下文。最后提一点，`pwd` 是什么意思呢？`print work directory` 缩写罢了。

#### 13.2.4 切换目录命令——`cd`

且先不提 `cd` 是怎么用的，我们先想想何谓切换目录？简单地来说，切换目录就是从一目录，换到另一个目录（哦，对了，目录就是文件夹的意思，希望大家在看前文时不会不懂）。在 `windows` 当中我们已经很习惯于用鼠标，双击两下文件夹的图标，这样就从一个父级的目录进入到它的一个子级的目录，看，`windows` 下的切换目录就如此简单，只需哒哒两下就搞定了。但在 `Linux` 如何用命令切换目录呢？就是这 `cd` 命令（`change directory`）。现在假设我在当前目录下有一个 `acm` 文件夹，我要进去这里，只要输入 `cd acm` 并回车就 OK 啦。自己试一下吧。下图是我运行的结果。

```
[19:37:15 zeropointer@zeropointer-PC:~]:) $ ls
acm/  test.c
[19:37:17 zeropointer@zeropointer-PC:~]:) $ pwd
/home/zeropointer
[19:37:18 zeropointer@zeropointer-PC:~]:) $ cd acm
[19:37:22 zeropointer@zeropointer-PC:~/acm]:) $ pwd
/home/zeropointer/acm
[19:37:23 zeropointer@zeropointer-PC:~/acm]:) $
```

看到了吗？在我进行 `cd` 切换到 `acm` 目录之后，再 `pwd` 查看当前目录就变成 `/home/zeropointer/acm` 了。

现在可以总结下 `cd` 命令的用法了，`cd` 命令可以实现目录间切换，方法就是 `cd` 后跟一个空格，然后是要切换到“路径”的名称。这个路径可以是一个绝对路径，也可以是一

个相对路径。

等等，好像晕了，什么路径，还有绝对路径和相对路径都是什么？OK，以下内容开始科普，科普的意思是下边的知识不只适用于 linux 系统，windows 同样适用。

首先大家已经建过目录组成的一棵“树”形的结构了，树的根结点是最顶层的目录，由它分出一系列子目录。树中有父结点和子结点之分，同样在目录结构中也具有这些称谓。就拿/home/zeropointer 来说吧，home 是根目录“/”的子目录，zeropointer 是 home 的子目录，反过来，home 是 zeropointer 的父目录，根目录“/”是 home 的父目录，就是这样的关系。

在操作系统中有两个特殊的符号代表自己和它的父目录，它们就是“.”和“..”（那就是实心的句号）。一个点的代表这个目录本身，二个点的代表这个目录的父目录。有了这两种特殊的记号，我们就可以从一个子目录进入它的父目录了，方法就是 cd ..回车。如下图：

```
[19:37:22 zeropointer@zeropointer-PC:~/acm]:) $ pwd
/home/zeropointer/acm
[19:37:23 zeropointer@zeropointer-PC:~/acm]:) $ cd ..
[20:14:07 zeropointer@zeropointer-PC:~]:) $ pwd
/home/zeropointer
[20:14:08 zeropointer@zeropointer-PC:~]:) $
```

这样，你就可以随意地在任何目录之间进行切换了。注意根目录由于处理树的最顶层，因此根目录没有父目录，对根目录进行 cd ..命令也不会有任何作用。

上边说的是父目录和子目录的关系，接下来该谈谈什么是绝对路径和相对路径了。

路径这个词直观来看就是从从一个地方到达另一个地方所走过的道路，如上图中的/home/zeropointer/acm 就是一个路径，它代表了从根目录到 acm 这个目录的完整路途，从中你能看出它经过了 home 和 zeropointer 目录才从根走到 acm 这里。这就是一条路径。

绝对路径，它是从根目录开始到达某一个目录的路径，任何绝对路径一定是从根目录开始的。比如/home，/home/zeropointer，/home/zeropointer/acm 这些都是绝对路径。

相对路径，即然有相对这字眼了，那这个路径一定是相对于某一个目录来说的（这很像物理中的相对速度，如果 2 物体以同速度同方向运动，那一个物体相对于另一个物体来说速度就为 0），比如，zeropointer 相对 home 目录来说，它的路径就应该是./zeropointer，注意前边那个小点，别忘了，它代表本级目录，在这里这个点指的就是 home。再比如 acm 相对于 home 目录来说，路径是./zeropointer/acm。再比如 zeropointer 相对 acm 来说，它的路径是“../”，两个小点代表父级目录，因为 zeropointer 正是 acm 的父目录。home 相对 acm 来说就是“../..”了，一个../代表 zeropointer 这个目录，对它再来一个../就是 home 了。

有了绝对路径和相对路径的说法后，在使用 cd 时就更加灵活了，cd 后可接绝对路径也可以接相对路径。举个例子：假如我想进入根目录，则输入“cd /”即可。相对路径的使用方法比如：“cd ../”就是进入当前目录的上一层目录里。

### 13.2.5 创建文件夹命令——mkdir

当我们需要创建一个文件夹的时候，只要用 mkdir 加上所要创建的文件名即可。如：mkdir mydir 并回车。一个以 mydir 命名的文件夹就成功建立在当前目录下了。它取的是 make directory 之意。

```
[17:12:08 zeropointer@zeropointer-PC:~/test]:) $ ls
acm.c
[17:12:12 zeropointer@zeropointer-PC:~/test]:) $ mkdir mydir
[17:12:15 zeropointer@zeropointer-PC:~/test]:) $ ls
acm.c  mydir/
[17:12:17 zeropointer@zeropointer-PC:~/test]:) $
```



### 13.2.6 删除文件夹命令——rm/rmdir

创建好的文件夹，如果想删除怎么办呢？这里要分两种情况，就是当文件夹为空和不空的两种情况。当文件不空时，即其下存有其它的文件或文件夹时，是不可以用 `rmdir` 命令删除的，这种情况你，只有手动进入该文件夹将其内部的文件和文件夹都删除后，然后再用 `rmdir` 将这个空文件夹删除。而如果一个文件夹本身就是空的，这种情况就可以直接使用 `rmdir` 命令将其删除，方式是 `rmdir` 加上要删除的空目录的名称。比如：`rmdir mydir` 并回车，则命令生效将 `mydir` 这个空的目录删除。

```
[17:12:17 zeropointer@zeropointer-PC:~/test]:) $ ls
acm.c  mydir/
[17:13:04 zeropointer@zeropointer-PC:~/test]:) $ rmdir mydir
[17:13:11 zeropointer@zeropointer-PC:~/test]:) $ ls
acm.c
[17:13:12 zeropointer@zeropointer-PC:~/test]:) $
```

上图为文件夹 `mydir` 为空时的情况

```
[17:13:45 zeropointer@zeropointer-PC:~/test]:) $ ls
acm.c  mydir/
[17:13:46 zeropointer@zeropointer-PC:~/test]:) $ rmdir mydir
rmdir: 删除 "mydir" 失败: Directory not empty
[17:13:49 zeropointer@zeropointer-PC:~/test]:( $
```

上图为文件夹 `mydir` 为非空时的情况

另种命令是 `rm`。`rm` 的功能更多一些，它可以删除文件或目录。如果用 `rm` 命令删除一个文件，则 `rm` 加文件名即可，比如当前目录有一个 `acm.c` 文件，我要删除它，只要输入 `rm acm.c` 并回车就可以了。`rm` 更方便的地方在于它可以直接删除一个非空的文件夹。命令格式是 `rm -rf mydir`。参数 `-rf` 代表递归地删除目录并强制删除。

```
[17:13:49 zeropointer@zeropointer-PC:~/test]:( $ ls
acm.c  mydir/
[17:14:54 zeropointer@zeropointer-PC:~/test]:) $ rm acm.c
rm: 是否删除普通空文件 "acm.c"? y
[17:15:03 zeropointer@zeropointer-PC:~/test]:) $ ls
mydir/
[17:15:05 zeropointer@zeropointer-PC:~/test]:) $ rm -rf mydir
[17:15:09 zeropointer@zeropointer-PC:~/test]:) $ ls
[17:15:10 zeropointer@zeropointer-PC:~/test]:) $
```

### 13.2.7 移动和复制——mv&cp

移动命令 `mv` 可以将一个文件或文件夹移动位置，类似于“剪切”的功能。命令格式如下：`mv 源路径 目的路径`。这里的路径即可以采用绝对路径，同时也可以采用相对路径，具体采用哪种形式，取决于你自己，看哪种方便就使用哪一种，不同的情况下不同要灵活运用。

现在有一个问题，假如我想给一个文件或者是文件夹进行重命名怎么办？`linux` 下没有提供直接进行重命名的命令，其它完全可以通过这个 `mv` 命令来实现，想想怎么办？`mv` 的功能是移动一个源地址的文件到目的地址，并且名称是可以改变的，假如现在我将源路径和目标路径都写成当前目录，而将要移动的文件名称换成其它的，就可实现重命名功能了。比如当前目录下有一个文件名叫 `acm.c`，我现在要想把它重命名为 `test.c` 如何做呢？只要输入命令“`mv acm.c test.c`”就可以了。下图为例。

```
[17:15:44 zeropointer@zeropointer-PC:~/test]:) $ ls
acm.c  mydir/
[17:15:45 zeropointer@zeropointer-PC:~/test]:) $ mv acm.c test.c
[17:15:54 zeropointer@zeropointer-PC:~/test]:) $ ls
mydir/  test.c
[17:15:55 zeropointer@zeropointer-PC:~/test]:) $
```

cp 命令实现了复制文件，它的参数也是两个就是源文件路径和目标文件路径，几乎是和 mv 使用方法相同了。比如现在要将当前目录下的 acm.c 文件，复制到当前目录下的 mydir 内，并取名 test.c。则只要输入 cp acm.c mydir/test.c 即可。注意无论是 acm.c 还是 mydir 都是相对路径，其完整写法应该是 cp ./acm.c ./mydir/test.c。因为一个点代表本层目录，而这是可以省略的，所以就出现了不写 ./ 的这种写法。

```
[17:16:39 zeropointer@zeropointer-PC:~/test]:) $ ls
acm.c  mydir/
[17:16:40 zeropointer@zeropointer-PC:~/test]:) $ cp acm.c mydir/test.c
[17:16:47 zeropointer@zeropointer-PC:~/test]:) $ ls
acm.c  mydir/
[17:16:48 zeropointer@zeropointer-PC:~/test]:) $ ls mydir
test.c
[17:16:55 zeropointer@zeropointer-PC:~/test]:) $
```

## 13.3 编写程序

### 13.3.1 引子

上一部分介绍了一些 Linux 下使用的基本命令，大家可以看到介绍到的几个命令都是对文件进行操作的。这些足够应付现在要在终端下写程序了。在终端下写程序要分为如下的几个步骤：编辑、编译、运行。

首先通过文本编辑器编写代码，这一步叫做编辑，通常我们在 Linux 下编辑代码可选用 vim 当然还有一些其它功能比较强大的如 emacs，在此不过多介绍，仅介绍 vim 的使用方法。

接下来将你写好的代码进行编译，大家一定要将编译与编辑进行区分，编辑是你写代码，而编译是由编译器翻译你的代码使它变成二进制文件好让 CPU 可以执行它。在 Linux 下的编译器 C 语言可以选用 GCC 而 C++ 可以选用 G++ 其它的语言各自有它的编译器，这里不再讨论。这一步涉及到 gcc 和 g++ 的使用方法，以及 makefile 的编写。

最后一步，假如你的代码没有任何语法错误，就可以生成一个可执行程序，运行它来验证你的程序是否正确即可。

所以你看，整个流程不再是像你在 windows 下写程序那样，打开 codeblocks 然后写代码，点一个按钮就能完成的。在 Linux 下你需要自己写编译指令之类的。那即然 windows 下那么方便为什么还要用这个 Linux 呢，怪麻烦的。其实正正相反，在 Linux 下任何事情都可以通过你写的程序变得简单起来，在 Linux 下你想做的任何事情都可以自己编写脚本让它自动执行，就拿编译来说，你可以将编译指令写入 makefile 中，这以后只要用 make 命令它会自动读取 makefile 文件内容进行一系统动作。这种方面自然不是我一言两语就能说清的，需要你在使用过程中慢慢体会。

### 13.3.2 编辑器 vim 的使用

首先是 vim 的安装，打开终端后输入 sudo apt-get install vim 并回车，然后它会提示你输入你的账号密码，这里你输要进行输入，但是，你输入的任何字符不会在屏幕上显示出来，所以，你看到当你按了好几个键后依然没反应时，不必在意，只管正常输的你密码就好。Linux 下这种输入密码时不显示字符的方式让你的密码变得能更安全一些，防止不法人员看到你的密码有多少位而进行破解。回车后它会自动搜索 vim 源并下载，中间可能输

要你输入几次 `y`，即确认，你只管输入 `y` 即可，片刻后 `vim` 就安装成功了。

安装成功后在终端输入 `vim` 并回车就可以打开 `vim` 的主程序了，当你第一眼看到它时应该没有什么喜感，它就和终端差不多，也是黑框框上边有几行字。你必须熟练使用才能挖掘到它的神通之处。

在介绍如何使用前先说明一下，`vim` 在使用时分两种模式，即命令模式和编辑模式，所谓的编辑模式的 `vim` 其实就是和记事本，`word` 这样的东西差不多，你可以在其中进行打字，包括回车、退格这些键都保持有它该有的功能。另一方面对于命令模式，这就类似于终端一样，`vim` 允许你输入它可以看懂的命令来对你写好的代码进行修改等。从编辑模式到命令模式的切换只要按一下 `ESC` 键就可以了，也就是说任何时候，只要是你在编辑你的代码，你一按 `ESC` 就退回到命令模式，只有在命令模式才能够输入命令进行代码的修改和命令下的编辑。其实这两个“编辑”容易混淆，不妨把编辑模式的编辑想成“录入”或“输入”，即录入模式或输入模式，这样好一些。

接下来开始简单地使用 `vim`，进入 `vim` 主程序后初始处理命令模式，此时你可以键入 `vim` 可以识别的命令，比如按一个 `i` 键，则进入输入模式，这里你可以在其中输入任何字符，都会直接显示在 `vim` 主窗口中。此时若想回到命令模式则按一下 `ESC`，OK 这就退回到命令模式了。其实命令是什么呢？刚才输入的 `i` 就是一个命令，该命令告诉 `vim`，用户想要在光标当前位置插入一个字符，这样 `vim` 就知道将光标所在位置不动，并进行输入模式，这时用户就可以输入字符了。

现在想想，要是写完了代码想要保存文件怎么办？这其实也是依告命令来实现的，在命令模式下输入 `:w filename` 并回车，`vim` 就会以 `filename` 做为文件名，将它保存到你打开 `vim` 时的当前目录，就是说假如你的当前目录 `pwd` 的结果是 `/home/zeropointer/acm`，在此处你 `vim` 了一下，然后再 `:w acm.c` 这样的结果就是在 `/home/zeropointer/acm` 目录下建立了一个 `acm.c` 文件。

保存完了文件后要退出 `vim`，这其实也是一条 `vim` 命令，在命令模式下输入 `:q` 并回车，就可以退出 `vim` 并回到终端了。

现在你可以看出从进入 `vim` 到编辑到保存退出都是靠各种命令来实现的，没错，这就是 `vim` 特别之处，也正因为此，你甚至不需要动一下你的鼠标，只依靠键盘就可以完成所有的任务，当你能熟练地使用 `vim` 的时候，这效率简直要高出不知道多少倍。虽然在你不熟的时候，反而没有使用 `codeblocks` 这样的现成工具快，但你只要常用常练，一点点就能得心应手了。

我承认我很懒，在此不能将 `vim` 的每一个命令都面面俱到的给大家讲的清清楚楚，这也偏离了咱们这入门手册的主题，但又怕大家在使用过程中遇到一些操作却不知道用什么命令好，在此我将主要常用的命令以表格的形式列出如下，大家自己尝试着输入以后命令来看看能达到什么样的效果。而更多的使用 `vim` 的东西，请自己百度找资料吧。另外推荐一个学习 `vim` 的好东西，在 `windows` 下有一个 `gvim` 软件，百度就能搜到下载，安装它后，它有一个叫用于实践练习 `vim` 的助手，或者说是家教，你运行那个程序就可以入门 `vim` 了。

下表为常用的命令：

命令模式	命令解释
<code>h</code> 或 向左方向键	光标向左移动一个字符
<code>j</code> 或 向下方向键	光标向下移动一个字符
<code>k</code> 或 向上方向键	光标向上移动一个字符
<code>l</code> 或 向右方向键	光标向右移动一个字符
<code>Ctrl+f</code>	屏幕向前翻一页（常用）
<code>Ctrl+b</code>	屏幕向后翻一页（常用）
<code>Ctrl+d</code>	屏幕向前翻半页
<code>Ctrl+u</code>	屏幕向前翻半页
<code>+</code>	光标移动到非空格符的下一列
<code>-</code>	光标移动到非空格符的上一列
<code>n&lt;space&gt;</code>	按下数字后再按空格键，光标会向右移动这

	一行的 $n$ 个字符。例如 <b>20&lt;space&gt;</b> , 则光标会向右移动 <b>20</b> 个字符
<b>O (HOME)</b>	(是数字 <b>0</b> ) 动到这一行的第一个字符处 (常用)
<b>\$ (END)</b>	移动到这一行的最后一个字符处 (常用)
<b>H</b>	光标移动到这个屏幕最上方的那一行
<b>M</b>	光标移动到这个屏幕中央的那一行
<b>L</b>	光标移动到这个屏幕最下方的那一行
<b>G</b>	光标移动到文件的最后一行
<b>nG</b>	移动到这个文件的第 $n$ 行。例如 <b>20G</b> , 则会移动到这个文件的第 <b>20</b> 行 (可配合: <b>set nu</b> )
<b>n&lt;Enter&gt;</b>	光标向下移动 $n$ 行 (常用)
<b>命令模式</b>	<b>查找与替换</b>
<b>/word</b>	在光标之后查找一个名为 <b>word</b> 的字符串 (常用)
<b>?word</b>	在光标之前查找一个名为 <b>word</b> 的字符串
<b>:n1,n2s/word1/word2/g</b>	在第 $n1$ 与 $n2$ 行之间查找 <b>word1</b> 这个字符串, 并将该字符串替换为 <b>word2</b> (常用)
<b>:1,\$s/ word1/word2/g</b>	在第一行与最后一行之间查找 <b>word1</b> 这个字符串, 并将该字符串替换为 <b>word2</b> (常用)
<b>:1,\$s/ word1/word2/gc</b>	在第一行与最后一行之间查找 <b>word1</b> 这个字符串, 并将该字符串替换为 <b>word2</b> , 且在替换前显示提示符让用户确认 (conform) (常用)
<b>一般模式</b>	<b>删除、复制与粘贴</b>
<b>x, X</b>	<b>X</b> 为向后删除一个字符, <b>x</b> 为向前删除一个字符 (常用)
<b>Nx</b>	向后删除 $n$ 个字符
<b>Dd</b>	删除光标所在的那一整行 (常用)
<b>Ndd</b>	删除光标所在列的向下 $n$ 列, 例如, <b>20dd</b> 则删除 <b>20</b> 列 (常用)
<b>d1G</b>	删除光标所在行到第一行的所有数据
<b>dG</b>	删除光标所在列到最后一行的所有数据
<b>Yy</b>	复制光标所在行 (常用)
<b>Nyy</b>	复制光标所在列的向下 $n$ 列, 例如, <b>20yy</b> 则是复制 <b>20</b> 列 (常用)
<b>y1G</b>	复制光标所在列到第一列的所有数据
<b>yG</b>	复制光标所在列到最后一列的所有数据
<b>p, P</b>	<b>p</b> 为复制的数据粘贴在光标下一列, <b>P</b> 则为粘贴在光标上一列 (常用)
<b>J</b>	将光标所在列与下一列的数据结合成一列
<b>U</b>	恢复前一个动作 (undo)
<b>编辑模式</b>	
<b>i, I</b>	插入: 在当前光标所在处插入输入的文字, 已存在
<b>a, A</b>	添加: 由当前光标所在处的下一个字符开始输入, 已存在的字符会向后退 (常用)
<b>o, O</b>	插入新的一行: 从光标所在行的下一行行首开始输入字符 (常用)



r, R	替换： <b>r</b> 会替换光标所指的那一个字符； <b>R</b> 会一直替换光标所指的文本，直到按下 <b>Esc</b> 为止（常用）
Esc	退出编辑模式，回到一般模式（常用）
命令行模式	
:w	将编辑的数据写入硬盘文件中（常用）
:w!	若文件属性为只读，强制写入该文件
:q	退出 <b>vi</b> （常用），快捷方式为 <b>SHIFT+ZZ</b>
:q!	若曾修改过文件，又不想保存，使用 <b>!</b> 为强制退出不保存文件，快捷方式为 <b>SHIFT+ZQ</b>
:wq	保存后退出，若为 <b>:wq!</b> ，则为强制保存后退出（常用）
:w[filename]	将编辑数据保存为另一个文件（类似另存新文档）
:r[filename]	在编辑的数据中，读入另一个文件的数据。即将 <b>filename</b> 这个文件内容加到光标所在行的后面
:set nu	显示行号，设定之后，会在每一行的前面显示该行的行号
:set nonu	与 <b>set nu</b> 相反，为取消行号
:set nohlsearch	可取消高亮，可编辑/etc/vimrc 来编辑取消所有高亮
n1,n2 w[filename]	将 <b>n1</b> 到 <b>n2</b> 的内容保存为 <b>filename</b> 这个文件

### 13.3.3 gcc/g++编译代码

大家应该也知道，一个程序的生成包括了代码的编辑——>编译——>可执行文件运行。上文提到的 **vim** 使用来进行代码编辑工作。接下来，要为我写好的代码进行编译了。说起这编译，在此处其实是两大过程，即编译+链接，源代码首先由编译器编译成.o 或.obj 的目标文件，然后由链接器接.o 或.obj 文件加入静态库.a 或.lib 从而生成可执行程序。而这里提到的 **gcc** 或 **g++** 它们都是可以完成编译+链接两个过程的。当然两个编译器同样也提供给你一些参数选项对你的代码只编译而不链接。这里我以 **gcc** 为例进行说明，**g++** 的用法和 **gcc** 在初阶段几乎没有不同之处。

针对于写好的代码 **acm.c** 进行编译只要输入如下的指令：**"gcc -Wall -o acm acm.c"** 然后回车，**gcc** 就开始对你的代码进行编译了，OK 我详细解释一下这条命令的含义，但是，等一下，这条命令在哪里输入？可别和上边说的 **vim** 的命令模式弄混了，**vim** 的命令模式仅是 **vim** 程序中的命令，你这是要运行 **gcc**，自然要在终端输入啦，你是在和系统对话哦！

首先 **gcc** 代表主程序名，说明我要运行的程序是 **gcc** 接下来由 4 部分组成，每一部分由一个空格进行分隔。

其第一部分是 **-Wall** 这是告诉 **gcc** 我要开启所有的警告，大家应该知道编译程序中会产生 **error**(错误)或 **warning**(警告)这两种，而这个 **-Wall** 就是告诉 **gcc** 不管啥样的警告都给我列出来，因为有些不是很重要的警告在默认情况下是不会被列出来的，这取决于你对代码要求的严格程度，但我们写程序重要追求一个完善，你肯定不希望由于一个 **warning** 导致你的程序产生大 **BUG**。

第二部分是 **-o acm**。好吧我承认我说的有问题了，上边说是 4 部分，而且是由空格分隔的，这里突然又出来个空格连接的 2 部分，那不是总共就分三部分了吗。的确，应该就是三部分，这个 **-o acm** 是一个整体，**-o** 是告诉 **gcc** 编译好后生成的可执行文件我要指定一个文件名，这个指定的命令就是由 **-o** 传给 **gcc** 的，而 **-o** 的后面紧跟的就是要输出的文件名，如果你不加这个 **-o acm** 默认是以 **xxx.out** 命名的。

第三部分 `acm.c` 这个就是告诉 `gcc` 我要编译的代码是 `acm.c`，你就给我编译它吧。

OK 整个三部分连起来的意思就是，`gcc` 你给我编译一下 `acm.c` 这个文件，我要求你提示出代码中所有的警告，并且以 `acm` 作为生成的可执行文件的文件名。

嗯，这样就解释完这条指令了，以后只要你要编译代码就这样写。

```
[20:56:57 zeropointer@zeropointer-PC:~/test]:) $ ls
acm.c mydir/ tests.c
[20:56:58 zeropointer@zeropointer-PC:~/test]:) $ gcc -Wall -o acm acm.c
[20:57:05 zeropointer@zeropointer-PC:~/test]:) $ ls
acm.c acm.exe* mydir/ tests.c
[20:57:07 zeropointer@zeropointer-PC:~/test]:) $
```

你可能要吐槽了，让我写完代码输这么一长条的命令编译。我还不如去用 `codeblocks` 呢！好吧我也承认它太长了，以至于是个程序员就会闲它太麻烦了，如果每次写完代码都要输这么长的命令编译，恐怕 `Linux` 程序员都要累吐血了。不过程序员毕竟是程序员，能写出来编译器我就不能写一个东西来解析这编译命令吗，以后把这些编译指令写到文件里，每次去解释这个文件不就可以了么？程序员毕竟还是聪明的。于是 `make` 程序就由此产生了。

### 13.3.4 使用 `make` 进行编译

说是 `make` 进行编译，但其实还是 `gcc` 进行编译，毕竟编译器只有 `gcc` 是啊，而 `make` 只是一个程序用来解释一系统 `gcc` 指令的，你将编译指令写入到一个文件当中，这个文件名叫 `makefile` 或者 `Makefile`，然后在终端输入 `make` 后，`make` 程序自动找寻当前目录中的 `makefile` 或 `Makefile` 然后它读取文件内的内容来控制 `gcc` 进行如何如何的工作。这样你只要将指令 `gcc -Wall -o acm acm.c` 这东西写到那个文件里再回到终端 `make` 就可以了。方便很多。

但是 `make` 文件有自己的语法规则，你也不能随便乱写，所以下文开始介绍如何编写一个简单的 `makefile`。

`makefile` 的规则如下：

```
target : prerequisites
command
...
```

`target` 也就是一个目标文件，可以是目标文件即 `.o` 或 `.obj`，也可以是执行文件 `.exe`。还可以是一个标签（`Label`），对于标签这种特性，本文说明，请自行查阅相关资料。

`prerequisites` 是要生成那个 `target` 所需要的文件或者说是依赖的文件。

`command` 也就是 `make` 需要执行的命令。

我们按照上述的语法规则进行一个简单的 `makefile` 编写就是这样：

```
acm : acm.c
      gcc -Wall -o acm acm.c
```

解释一下，第一行 `acm` 代表我要生成的可执行文件名，即目标名称叫 `acm`，后边紧跟一个冒号，然后就是生成的目标所依赖的文件，你看，咱们是用 `acm.c` 产生的 `acm` 目标，自然它所依赖的文件就是 `acm.c` 啦。第二行，这里写命令，即编译指令。就是那一长串咱们懒得去写的指令。效果图如下。

```
[21:14:53 zeropointer@zeropointer-PC:~/test]:) $ ls
acm.c makefile mydir/ tests.c
[21:14:54 zeropointer@zeropointer-PC:~/test]:) $ cat makefile
acm : acm.c
      gcc -Wall -o acm acm.c
[21:14:58 zeropointer@zeropointer-PC:~/test]:) $ make
gcc -Wall -o acm acm.c
[21:15:01 zeropointer@zeropointer-PC:~/test]:) $ ls
acm.c acm.exe* makefile mydir/ tests.c
[21:15:02 zeropointer@zeropointer-PC:~/test]:) $
```

一个简单的 makefile 就这样写好了，当然这只是简单的、初级的、入门的 makefile 写法，更多的 make 语法规则你需要另找相关书籍来学习。你想想一个庞大的工程可能会有上百个源文件，上千个头文件，把它们组织好了，也就是说制定合理的编译顺序，把它们构建成为一个大工程，这需要熟练地 makefile 写法基础，而 windows 程序员从来不需要关心这东西，因为在 windows 下一切编译顺序问题都由 IDE 比如 VC 或是 Code::Blocks 来为我们完成了，我们并不能看到它内部是如何做的，做为一名 Linux 程序员你应该有本事写出来 makefile。

最后在这里推荐学习资料 <http://www.cnblogs.com/liyanwei/archive/2010/04/29/1723931.html>

### 13.3.5 运行

运行这里其实并没有过多好说的，但是不说肯定是不行。比如你 make 出来的一个程序怎么执行呢？就上文写到的生成的 acm 可执行程序，我直接在终端输入 acm 并回车可以吗？你可以试一试，我这里的结果如下：

```
[21:15:01 zeropointer@zeropointer-PC:~/test]:) $ ls
acm.c acm.exe* makefile mydir/ tests.c
[21:15:02 zeropointer@zeropointer-PC:~/test]:) $ acm.exe
-bash: acm.exe: command not found
[21:33:45 zeropointer@zeropointer-PC:~/test]:( $
```

看到了吗，它说命令没有找到。也就是说直接输入可执行程序的文件名的话，终端在默认情况下把它当作为一条命令来解释，但是却没有这个命令，因此就会产生如上图所示的错误。那想想我要运行这个程序该怎么写呢？记不记得“.”这个东西代表本级目录，或者说当前目录，acm.exe 处理当前所在的目录，我要运行它，加上“.”是不是就可以了呢？试一下吧，事实证明，这样做是可以的。在终端输入“./acm.exe”并回车，发现程序可以运行了。效果图如下。

```
[21:36:19 zeropointer@zeropointer-PC:~/test]:) $ ls
acm.c acm.exe* makefile mydir/ tests.c
[21:36:19 zeropointer@zeropointer-PC:~/test]:) $ ./acm.exe
hello,world!
[21:36:22 zeropointer@zeropointer-PC:~/test]:) $
```

OK，我们写的程序成功地在 Linux 下通过终端运行起来了。任务总算是完成了。

这里要解释一下为何直接输入 acm.exe 不可以呢？原因是这样的，默认情况下在终端提示符后输入的东西都是以命令方式呈现的，也就是说你输入的所有合法的东西都是一条 shell 命令，那这些命令是什么呢？比如 cd,ls,cp,mv,mkdir,vim 这些命令？其实这些东西你在 /bin 目录下都可以找到对应的可执行程序，也就是说这些东西原本都是一个可执行文件，你在终端输入这些命令的时候相当于运行了这些程序，运行后程序依它实现的功能显示给你不同的结果。但是有些你却在/bin 下找不到，这些找不到的属于 shell 内嵌命令。

### 13.3.6 总结

至此你已经可以在 Linux 下通过终端进行代码的编写、编译并去执行它。我们通过 vim 编辑代码，通过 gcc 对代码进行编译，然后在终端运行，显示出结果来，这一切在 windows 下原本可以直接用 IDE 一下搞定的东西似乎在 Linux 下被分成这么几个步骤，似乎变得麻烦了。其实你暂时看到的都是由于陌生引起的，俗话说熟能生巧，当你能熟练地使用这一切的时候，把它们变得得心应手的时候，你会发现开发速度较 windows 来快很多，一个原因可能是你不至于在键盘和鼠标之间来回换手，你的手不会离开键盘去碰鼠标。