C++ 入门教程

好渴鹅

目录

1 前言

这是好渴鹅完全独立自主编写的一本如何入门 C++ 的指南，采用 Typst 文档系统系统构建，仅供学习，请勿用于商业用途。

其实现在市面上不管是博客啊还是实体书等等等等，都有非常多专门教学 C++ 的，但它们各有优缺点：网上的教程虽然不乏有一些很好的文章，但是只是太过于碎片化了，很多专门用来教学 C++ 的网站也没有对知识进行详细的分类与讲解：实体书呢则是知识板块分类较为详细，但是往往较为昂贵。

因此，好渴鹅就编写了这本《C++ 入门教程》电子书，希望能够给予想要零基础学习 C++ 的人们一点点帮助。俗话说得好："前人栽树，后人乘凉。"或许我种的树没有那么茂盛、那么郁郁葱葱，但是我相信一定会有更多的"前人"为后人栽下希望的树苗。

同时，好渴鹅在编写文档的时候难免也会出现一些错误或漏洞。如果你找到了漏洞并且想用你那善良的心修复的话，请发邮件到 19173155158@163.com，好渴鹅将会在七个工作日内给予回复。

个人网站：Haokee-Wiki

——好渴鹅

2 配置环境

打开你的电脑，如何写代码并运行呢？不要着急，你可能还没有安装 C++ 环境。本章节主要讲解了如何在自己的电脑上安装不同的代码编辑器和编译器或集成开发环境，读者可以自行选择喜欢的软件安装。

2.1 Visual Studio (Windows)

Visual Studio 是微软推出的一个强大的集成开发环境，可用于构建适用于多种平台和语言的应用程序。以下是官网的介绍：

> Visual Studio 是一款功能强大的开发人员工具，可用于在一个位置完成整个开发周期。 它是一种全面的集成开发环境 (IDE)，可用于编写、编辑、调试和生成代码，然后部署应用。 除了代码编辑和调试之外，Visual Studio 还包括编译器、代码完成工具、源代码管理、扩展和许多其他功能，以改进软件开发过程的每个阶段。
>
> 凭借 Visual Studio 中的各种功能和语言支持，你可以从编写第一个"Hello World"程序进化到开发和部署应用。 例如，生成、调试和测试 .NET 和 C++ 应用，在 Web 设计器视图中编辑 ASP.NET 页面，使用 .NET 开发跨平台移动和桌面应用，或在 C# 中生成响应式 Web UI。

总的来说，就是非常厉害的一个集成开发环境，不管你是初学者还是 C++ 老手，都可以愉快地使用 Visual Studio 来进行开发。让我们安装吧！

打开 官网，下载 Community 版本（现在应该是 Visual Studio 2022）。下载完之后，双击下载完的 exe，它就会自动帮你安装安装器。过了一会，就可以来到安装界面，有很多选项可以勾选。我们勾选"使用 C++ 的桌面开发"，（如果你的 C 盘不够用的话点开安装位置，把安装目录最前面的 C 改成 D 或其它盘），然后点击安装就行了。

过了大约两年半，软件大概就安好了。如何这时它告诉你要重启的话你就重启，重启完之后在开始菜单里面找到 Visual Studio (2022) 打开，跟着引导走，就可以来到项目界面。点击创建新项目，勾选

“C++ Windows 控制台应用程序”，填好项目名和目录（其它的尽量不要乱改），点击创建就可以创建新项目了。

创建完之后，会来到编辑界面，此时就可以开始写代码了。

2.2 MinGW (Windows)

MinGW 是什么？MinGW 其实是一个在 Windows 上运行的编译器的名称，用来把 C++ 代码编译成机器可以直接运行的程序。

我们知道，计算机的处理方式是二进制，那么我们的机器码也是二进制的，只有机器码才可以让计算机做出事情。但是，机器码它反人类啊！你想一想，光是让计算机执行一个加法就需要好多代码，因为计算机只能对位进行具体的操作，因此我们肯定是不可能直接写机器码的。

因此，后人开发了汇编语言，这种语言使用了很多英文符号，把英文符号转化为对应的二进制机器码就可以让计算机运行。但是汇编语言及其复杂，只是写起来简单一些，但是代码仍然很长。

这时，高级语言就出现了，如：C/C++、Python、Java、C#.....这种语言的代码构成就很简单了，而运行的方式就是需要一个翻译器，把代码翻译成机器码。一般有编译式和解释式，编译式就是直接编译成机器码，然后输出文件，最后点击文件就可以直接运行了；解释式就是相当于运行时逐句翻译，可以更好地面对运行时的错误，但是牺牲了速度。（**补充：**Java 等 JVM 语言是编译式和解释式的结合）

那为什么我们刚刚安装 Visual Studio 的时候没有安装编译器呢？因为 Visual Studio 是一个自带编译器的集成开发环境，因此不需要自己手动安装编译器。但是如果你想要安装 VS Code 或 CLion（较新版本自带编译器，但是功能不全）等不自带编译器的编辑器或 IDE 的话，那么就需要自己安装编译器了。

叭叭了一大堆，让我们开始安装吧！

首先转到 MinGW-w64 - sourceforge，然后一直往下翻看到 MinGW-W64 Online Installer，也就是在线安装器，但是**不要点**，下载下面的 x86_64-posix-seh，然后解压就行了。

MinGW-W64 Online Installer

- MinGW-W64-install.exe

MinGW-W64 GCC-8.1.0

- x86_64-posix-sjlj
- x86_64-posix-seh
- x86_64-win32-sjlj
- x86_64-win32-seh
- i686-posix-sjlj
- i686-posix-dwarf
- i686-win32-sjlj
- i686-win32-dwarf

接下来，假如你刚刚解压到了 `C:\mingw64` 目录，没有解压到这里也没关系，把接下来的这个目录都替换成自己的目录就行了。

然后打开文件资源管理器，点击此电脑，按 `Alt + Enter` 或者右键点属性，就会来到设置页面。往下滑，找到高级系统设置，点击环境变量，找到系统变量中的 Path 变量，双击点进去，然后将 `C:\mingw64\bin` 这个目录添加进去。

点击确定，一直推到文件资源管理器之后按 `Win + R` 打开运行窗口，然后输入 cmd 回车，接着在弹出来的黑框框内输入 `g++ -v`，如果它给你了一大段版本信息，就说明你的编译器已经配置好了。

2.3 GNU (Linux)

对于 Linux 用户，可以直接在命令行下运行该命令安装 GNU 编译器，命令仍为 G++。

```
$ sudo apt install g++
```

然后使用 `g++ -v` 就可以检测是否安装完毕了。十分简单。

2.4 XCode (MacOS)

XCode 是苹果官方的集成开发环境，用于开发 Objective-C 和 Swift（包括普通的 C++）。但是它自带了 G++ 编译器因此非常方便，只需要安装几个 G。

首先，打开终端输入 `g++ -v` 看一看有没有报错，如果有的话再安装，没有的话就不用重新安装一遍了。

然后打开 MacOS 自带的 App Store，搜索 XCode，点击获取，过一会就安装完成了。再重新打开终端，就可以发现命令没有报错了。

2.5 CLion (Windows/Linux/MacOS)

如果你想要更好的跨平台体验的话，那么 JetBrains 家的 CLion 就是一个比较好的选择。安装完编译器之后，我们找到 JetBrains 的 官网安装地址，选择自己的系统进行安装就可以了。

安装完之后，它可能会告诉你没有许可证，你可以点击试用三十天，那么你就可以免费使用三十天了。如果你用的机房电脑有系统盘还原的话，那么可以每次安装都免费试用三十天。

正式进入软件，点击“New Project”就可以来到创建项目的页面。填上项目的名称和目录就可以了，其他的选项尽量不要乱动。创建完之后就可以来到编辑页面了。开始打代码吧！

3 基础语法

3.1 引入

3.1.1 向世界问好

作为一个刚装完环境的小白，你想不想立马编出一个可爱的小程序，并让它向世界问好呢？打开 IDE，输入以下的代码：

```cpp
#include <iostream>

using namespace std;

int main() {
  cout << "Hello world!" << endl;
  return 0;
}
```

然后运行，如果没有错误的话，程序将会弹出一个黑框框窗口，然后在该窗口里面打印 `Hello world!` 这一串文本。

如果 IDE 出现了报错没能启动程序，那么大概率是你有某些地方写错了，请一个一个字母对照看出现了那些问题（缩进不用检查），如果眼睛不好的建议直接复制。**注意 C++ 的每一行有实际含义的代码都是需要打上分号的。**

如果黑框框是一闪而过的话，那么大概率是你的 IDE 的配置有问题，不过在 `return 0;` 的前面加上以下这一行就行了。

```
system("pause");
```

好的，现在我们来一行一行看程序是如何打印出 `Hello world!` 的。

第一行：`#` 后面跟着的是预处理命令，表示编译器会在编译的时候就预处理的命令。`#include` 表示我们需要包含指定的头文件，也就是导入编译器的开发人员预先定义的文件，这样子我们就不用什么都自己写了。`iostream` 就是 C++ 里面推荐用来进行输入输出的头文件，使用 `< >` 包裹表示这是系统头文件而非自己定义的。

第二行：这是一行空行，为了让代码更美观、阅读起来更好看。

第三行：`using` 表示使用，`namespace` 表示命名空间，`std` 是命名空间的名字，连起来就是使用一个名叫 `std` 的命名空间。这是为了方式标准库当中的类、变量与自己定义的重名的。但是我们暂时不需要自己自己定义，因此我们就使用这个语句，不写的话所有标准库里面的东西就都需要加上 `std::` 前缀。

第四行：这是一行空行，为了让代码更美观、阅读起来更好看。

第五行：这是定义 `main` 函数的地方，`main` 函数就是程序的入口，所有的语句都会一条一条执行。前置 `int` 表示该函数的返回值为整数（Integer）类型，若为 0 则表示程序运行正常。`()` 空括号表示我们的 `main` 函数不接受任何参数，但这并不代表 `main` 不能接受参数，因此这个空括号**不可以不写**。大括号包裹的就是函数体，里面是需要运行的语句。

第六行：这是输出 `Hello world!` 的核心代码。`cout` 就是 C++ 当中的标准输出（默认是控制台），输出的内容用 `<<` 一个一个连接，`"Hello world!"` 就是一个名叫 `Hello world!` 的字符串。为什么是字符串呢？就比如 `cout`，没有用引号包起来就表示这是运行的语句，用引号抱起来就是可以输出的字符串了。后面跟着输出 `endl` 就表示换行并立即刷新缓冲区，刷新缓冲区是为了让输出更快地显现在控制台上，但是运行效率会变慢，因此可以考虑使用带有 `\n` 的字符串，也就是换行符。不过我们的程序并没有对效率有很高的要求，因此可以不删。

第七行：这是用来退出主函数也就是程序的，`return 0` 就是返回 0 的意思，也对应了前面所说的内容。

第八行：与前面的主函数的大括号成对。

是不是恍然大悟呢？没有听懂也没关系，可以自行上网了解。也可以将引号内的字符换成其他的，达到输出其它内容的效果。

3.1.2 代码习惯

养成好的代码习惯跟固定的代码风格是非常重要的。不过，代码风格没有美丑之分，适合自己的才是最好的，但是一定要养成。现在好渴鹅来简单讲解一下好的代码习惯。

首先，**要有良好的缩进**。就像刚才的代码里的主函数一样，大括号括起来的函数体都是锁进了哪个空格的，这样子才能分的清层次。除非你的眼力真的很好。一般都是空 4 个空格或者 2 个空格，但是**尽**

量不要用 Tab。因为 Tab（制表符）的长度是不固定的，虽然大多数编辑器都是等同于 4 个的长度，但是有些会是 8 个空格，因此为了代码的美观，请尽量不要使用 tab。

然后，在比较拥挤的地方，尽量要**空开一个空格**，至少要看上去美观一些，但是不要空太多空格。空空格的好处也是读起来更美观。当然，如果你真的喜欢把代码挤在一起的话，那当我没说。

古语有云：“写代码是一门艺术。”我们都是艺术家，但是“画”并不是给自己看的，而是给别人看的，让别人领会你的“画”的美丽。因此养成好的代码习惯比学习算法更加重要。

3.1.3 A+B 问题

到现在，我们只能在控制台上打印一些简单的东西，那能不能让用户在控制台上输入某些东西呢？其实非常简单，我们可以使用 `cin` 来输入，而输入的内容就需要用 `>>` 来连接，与 `cout` 恰恰相反，是不是非常形象？

但是在输入之前，我们遇到了一个问题：我们如何存储输入的信息呢？非常简单，我们可以使用变量来存储信息。例如，在主函数里面 `return 0;` 前面加入如下代码可以创建一个能够表达 -2^{31}~$2^{31}-1$ 的整数，而它的名称叫 `x`。

```
int x;
```

现在，尝试输出这个 x，看一看值为多少。（`//` 后面的内容是注释，是给人看的，编译器不会管。多行注释可以使用 `/*...*/` 表示。）

```
cout << x;  // 也可以加 endl 或 '\n' 换行
```

多运行几遍，可以看到基本上都不是 0，而且有很大概率是不一样的。这是因为在主函数里面定义的变量不会默认赋初始值。解决这个问题很简单，可以把定义变量的代码放在主函数的外面或者像这样在定义的时候就直接赋 0。

```
int x = 0;
```

当然，你也可以单独写一行 `x = 0` 来赋值。既然是 A+B 问题，那么我们就需要定义两个变量了，a 和 b，用逗号隔开。不过不用进行初始化，因为我们会输入它们。

```
int a, b;
```

然后我们输入他们，并输出它们的和（使用 `+` 运算符），最后用 `\n` 换行。

```
cin >> a >> b;
cout << a + b << '\n';
```

最后运行该程序，就可以发现黑框框没有了输出，这是因为程序在等你的输入。现在，我们在黑框框内输入 `1 1`（将两个 1 用空格隔开），然后回车——程序立马输出了 2。可以看到，我们的程序成功地算出了 $1+1=2$。当然，你也可以输入其它的整数，但是记住不能超过 -2^{31}~$2^{31}-1$，不然则会因为溢出而运算错误。

值得注意的是，C++ 中的输入不用在意使用的是换行隔开还是空格隔开，只要你输入了足够数量的数，那么就会结束输入。因此你也可以使用换行输入的模式输入两个 1，但是记得输入完要回车。

3.2 变量与运算符

3.2.1 数据类型

在前面的代码当中，我们已经学会了如何定义 `int` 类型的变量。但是，数据类型远远不止 `int` 那些，还有更多其他的数据类型。

数据类型名称	含义	字节数量
`int`	存储 -2^{31}~$2^{31}-1$ 范围内的整数	4 字节
`short`	存储 $-65,536$~$65,535$ 范围内的整数，操作方法与 `int` 一样	2 字节
`bool`	布尔类型，值为 true (0) 或 false (1)	1 字节
`float`	单精度浮点类型，位数较少	4 字节
`double`	双精度浮点类型，位数较多	8 字节
`char`	字符型	1 字节

可以看到，表示的数据类型非常多。让我们来实验一下：

```
int a = 1;
bool b = true;
float c = 1.145141919810;
double d = 1.1145141919810;
char e = '!';
cout << a << ' ' << b << ' ' << c << ' ' << d << ' ' << e;
```

（此处输出的 `' '` 就是表示一个空格，用于隔开输出信息）输出的内容如下：

```
1 1 1.14514 1.14514 !
```

非常的有趣，我们可以发现几个特征：

- `char` 类型变量虽然在定义的时候赋值为的是 `true`，但是输出的时候却转到了 `int` 的格式——也就是 1。
- `float` 型变量与 `double` 型变量好像似乎没有什么差别，都输出了 6 位有效数字，而且都是一样的。可是 `double` 类型不应该表示的 `float` 类型更加精确吗？

对于第二个问题，其实答案非常简单。只是因为我们的 `cout` 输出默认不会保留那么多的有效位数，因此输出的时候它们看起来是一模一样的。后面好渴鹅会讲如何保留多少多少位小数。

但是，如果是进行除法运算，那么 `float` 与 `double` 之间的差别可就大了。我们都来计算 $\frac{1}{700}$ 的值，`float` 保留二十位小数部分输出的值为 $0.001,428,571,413,271,129,13$，而 `double` 输出的值为 $0.001,428,571,428,571,428,57$，可以显而易见地看出差别。

那么如何保留制定小数位数呢？非常简单，首先需要像引入 `iostream` 头文件一样地引入 `iomanip` 头文件：（但是引入 `iostream` 的语句不要删）

```
#include <iomanip>
```

然后在输出的时候在变量前面输出 `fixed` 和 `setprecision(要保留的位数)`。例如，如果你想输出 x 并保留 20 位，那么大概可以这样写：

```
cout << fixed << setprecision(20) << x;
```

需要注意的是，如果要同时保留多个数的小数位数，那么需要在每一个输出变量的前面都加上如此繁琐的语句。那么，有没有更简单的保留小数位数的方法呢？可以参考后面的 C 风格输入输出。

3.2.2 类型修饰符

以下，是所有的类型修饰符：

类型修饰符名称	含义
`signed`	表示有符号的类型
`unsigned`	表示无符号的类型
`long`	表示额外加长类型的长度

注意： `signed` 和 `unsigned` 可以单独使用，也就是说 `signed int = signed`，`unsigned` 同理。`int` 默认就是 `signed` 的。`long` 可以用来修饰 `int` 或 `double` 类型，修饰 `int` 类型可以单独使用，`int` 可以加两个，`double` 可以加一个。

`long int` 类型是 4 或 8 个字节的，取决于编译器和操作系统。`long long int` 是强制 8 个字节的，表示范围为 -2^{63}~$2^{63}-1$。

3.2.3 常量

在 C++ 当中，常量是一种特殊的变量，采用 `const` 前缀定义。常量必须是在定义的时候就必须初始化，并且初始化的值不能是变量表达式。(不同于 Java 中的 `final`，Java 中的 `final` 是只能赋一次值)

那你可能会问，常量有什么用，直接用变量表示常量不久行了吗？话虽然是这么说，但是编译器一般都会对常量进行特殊优化。比如取模操作，可以把模数定义为常量，这样子运算速度就会快很多。

```
int x = 1, y = 2;
const int a = x + y;   // 不行，不能赋值为变量
const int b;           // 不行，必须初始化
const int c = 100;
c = 1000;              // 不行，不能做运算
```

3.2.4 变量的运算

变量有很多种运算符，以下是表格：

运算符	含义
+	对两个操作数进行加法或正号
−	对两个操作数进行减法或负号
*	对两个操作数进行乘法
/	对两个操作数进行除法

运算符	含义
%	对两个操作数取模
+= 及其他	运算并赋值运算符
++	前后缀自增运算符
--	前后缀自减运算符
==	等于运算符
!=	不等于运算符
&&	逻辑与运算
||	逻辑或运算
!	逻辑非运算
&	位与运算
|	位或运算
~	位取反
^	异或运算
≪	左移运算符
≫	右移运算符
::	域运算符
.	成员访问运算符
->	成员访问运算符（指针）

现在我们来一个一个学习使用。

3.2.4.1 算术运算符

其实就是加减乘除和取模（整数除法得来的余数），可以参考下面的代码：

```
#include <iostream>

using namespace std;

int a = 114, b = 514;

int main() {
  cout << a + b << '\n'
       << a - b << '\n'
       << a * b << '\n'
       << a / b << '\n'
       << a % b << '\n';
```

```
    return 0;
}
```

输出结果：

```
628
-400
58596
0
114
```

3.2.4.2 += 等运算符

这些运算符没有具体的名称，但是效果十分简单：

```
a += b
a = a + b // 等同于
```

其它运算符也是一样的道理。注意这些运算符是**向右结合**的！例子：

```
int a = 1, b = 1;
a += b += 1;
```

此时，$a = 3, b = 2$ 而非 $a = 2, b = 3$。可以看出首先进行运算的是 `b += 1`，然后进行了 `a += b` 的运算。C++ 当中也有很多运算符的优先级十分恶心，参见下面的介绍。

3.2.4.3 自增减运算符

自增减是初学者最容易弄错的运算符之一了。别看 `a++` 和 `++a` 之间相差不多，但是实际使用中却又很大不同。

```
int a = 1, b = 1;
cout << a++ << ' ' << ++b;
```

可以看到，输出是 1 和 2，说明 `a++` 是先返回 a 本身，然后在把 a 改成 $a + 1$；而 `++a` 则是先把 a 改成 $a + 1$，然后在返回 a 本身。但是**注意**，`a++` 的实际实现是先拷贝一个跟 a 一样的数，然后再把 a 改成 $a + 1$，最后返回拷贝过去的数。因此，实际上返回的数是不能够更改的。例如：

```
a++++     // 不行
++++a     // 可以
(++a)++   // 可以
++a++     // 未定义的行为
```

自减运算符同理。

3.2.4.4 等于运算符

等于运算符（==）是初学者最容易弄错的运算符之二了。为什么呢？因为计算机中的单等于号（=）和数学中的等于号（=）虽然外形相似，但是实际上却有很大的差别。

在计算机当中，= 表示赋值并返回复制后的值。例如 `cout << (a = b)` 表示将 a 的值赋为 b，并返回 a 赋值后的值来输出。而判断两个变量是否等于的运算符是双等于号 = =。比如，若 `a = 1, b = 1`，那么 `a == b` 的返回值就为真（true）。

需要注意的是，在本文当中，为了是观看体验更佳，使用了 JetBrains Mono 字体的连字功能，因此在代码块当中 == 实际上会显示成 ═。其实只需要注意一下用途就行了。

不等于运算符就比较形象了，写作 !=。就比如刚刚的例子，`a != b` 的返回值就为假（`false`），因为 a 和 b 实际上是等于的。

需要注意的是，本文当中 != 在代码块里面会显示成 ≠，注意一下即可。

补充：一般情况下，`(a ≠ b)` 等价于 `!(a = b)`。

3.2.4.5 逻辑运算

- **逻辑与运算：**如果两边都为真，那么返回真，否则返回假；
- **逻辑或运算：**如果两边有任意一个为真，那么返回真，否则返回假。
- **逻辑非运算：**如果操作数为真则返回假，假则返回真。

一般用与 if 表达式当中，可以参照后面的内容。

3.2.4.6 位运算

位运算就稍微牵扯到了计算机基本原理方面了。在 C++ 当中，我们的所有变量都是由二进制所构成的。为了让用户更加地接近于底层，C++ 提供了很多直接对位进行操作的运算。

- **位与运算：**把两个数的位都给对齐，逐个进行操作。对于每一个位，如果两个参数都为 1，那么答案的对应位也为 1，否则为 0；
- **位或运算：**把两个数的位都给对齐，逐个进行操作。对于每一个位，如果有大于等于 1 个参数对应位为 1，那么答案的对应位也为 1，否则为 0；
- **位取反运算：**对操作数的每一个位都取反，0 变成 1，1 变成 0；
- **异或运算：**把两个数的位都给对齐，逐个进行操作。对于每一个位，如果两个参数不一样，那么答案的对应位就为 1，否则为 0；
- **左移运算：**把所有位往左移指定位数，高位舍去；
- **右移运算：**把所有位往右移指定位数，低位舍去；

需要注意的是位运算的优先级非常低，大部分时候都需要加上括号提高优先级。关于位运算的更多奇技淫巧请参照“更多技巧”小节。

3.2.4.7 其他运算符

使用比较广泛的是 `sizeof` 运算符，可以装入一个任意类型的参数，参数为单个变量时可以不加括号。返回的就是参数的字节长度。值得注意的是，`int` 等本身就存在的数据类型或自己定义的数据类型也可以用 `sizeof` 来求长度。

例如：`sizeof(int) = 4`，定义 `int a` 后 `sizeof(a) = 4`。塞入数组可以查看整个数组所有元素的字节长度之和。

其他运算符现在还根本用不到，到后面的时候会零零散散地讲解。

3.3 输入与输出

3.3.1 C++ 风格

C++ 使用 cin 进行标准输入，cout 进行标准输出，clog 进行日志输出，cerr 进行错误输出。存在与 `iostream` 头文件，std 命名空间里面。

那你可能会问了，输入只有一个 cin，为啥输出不仅有 cout 还有 clog 和 cerr 呢？其实解释起来非常简单。

你知道什么是输出缓冲区吗？如果是对每一个输出都直接打印到控制台上面的话，那么效率就会极其低下。因此，C++ 的创始人想了个办法：我搞一个数组存输出的内容，等数组满了在一起打印到控制台上面不久行了吗？

因此就有了不同的输出对象。cout 就是普通的输出，一般会在输入的时候或者缓冲区满了的时候进行刷新；clog 是专门用来输出日志的，因为一般程序的日志都会很多，因此使用 cout 会比较慢，而 clog 只会在缓冲区满了的时候自动刷新，因此需要手动刷新，效率也最高；cerr 是用来输出错误的，没有缓冲区的设计让他可以在程序异常的时候更好地及时输出错误信息。

刷新缓冲区的办法就是输出 flush。注意 endl 不仅会换行还会刷新缓冲区，这也就解释了为什么前面说使用 endl 换行效率极其低下。

3.3.2 C 风格

在古老的 C 语言中，输入和输出是使用 scanf 和 printf 函数来完成的。它们的格式如下：

```
scanf/printf(format: char *, ...)
```

format 就是将要格式化的字符串，后面任意多的参数就是要格式化进字符串里面的参数。格式化字符串当中有非常多的控制符需要记住：

控　制符	作用
%d	用来格式化一个整数
%c	用来格式化一个字符
%f	用来格式化一个单精度浮点数
%lf	用来格式化一个双精度浮点数
%s	用来格式化一个 C 风格字符串
%x	格式化整数为十六进制
%o	格式化整数位八进制
%ld	格式化一个长整型
%lld	格式化一个 long long 型
%Ix	格式化 x 位的整数

还有很多很多，在此不再描述。具体用法可以参照下面的：

```
scanf("%d", &x);   // 读入 x，注意取地址符
printf("%d", x);   // 输出 x
```

然后输入或输出其他不同类型的变量参照上面的格式表就行了，是不是非常简单。虽然很快就能学会，但是在普通时候，C 风格的输入输出比 C++ 的要繁琐许多，在此介绍两个 C 风格输入输出可以较好地操作的使用场景。

需要注意的是，C++ 的缓冲区默认会与 C 进行绑定，这样子就可以一会用 C++ 一会用 C 进行混合使用了，但是速度也会大打折扣。取消绑定的方法为 `ios::sync_with_stdio(0)`（或填 `false`），但是取消之后就不可以进行混合输入或混合输出了。

还有另外一个优化则是 `cin.tie(0)` 和 `cout.tie(0)`（或传入空指针）。正常情况下 C++ 的 cin 和 cout 是互相绑定的，也就代表着输出完之后如果有输入就会立马刷新输入的缓冲区好让用户接受输入信息，但是这样子也会降低输入/输出效率。增加上上面几行后，你就可以发现，有些代码的输出就会留到最下面。

这时，C++ 的输入输出的效率就比 C 要高了。

P.S：取地址符是 100% 需要的。那你可能会问，同样是输入，为什么 C++ 不需要取地址符呢？因为 C++ 对 C 进行了改良，加入了“引用”，正好可以弥补指针滥用的危险（特别是初学者）。而 C 还没有引用，因此就必须传入地址而非值（后面会讲）。

3.3.2.1 大量文本格式化

例如需要输出 `(a + b) = c` 这类的题目输出，就可以看出明显差别：

```cpp
cout << '(' << a << " + " << b << ") = " << c << '\n';  // C++
printf("(%d + %d) = %d\n", a, b, c);                    // C
```

可以看到，在这个使用场景下，C 风格输入输出不仅比 C++ 风格输入输出的代码要短，而且可读性还很高。

3.3.2.2 保留小数位数/设置宽度

前面说过了，保留三位小数 C++ 需要 `<< fixed << setprecision(3)`，但是 C 风格输入输出就只需要把控制符 %lf 改为 %.3lf 就行了。设置宽度同理，C++ 需要 `<< setw(3)`（还要加 `iomanip` 头文件），而 C 风格输入输出就只需要把控制符改为 %3d 就行了。

3.3.3 文件输入输出

程序的输入输出不仅有标准的输入输出操作，还有对文件的输入输出操作。在 C++ 当中，有三种文件操作是最为常用的。

3.3.3.1 fstream

使用该库之前，需要包含头文件 `fstream`，然后就可以定义自己的输入输出了。`ifstream` 是用来文件输入的，`ofstream` 是用来文件输出的。比如 `ifstream fin("text.txt")` 可以打开一个名叫 `text.txt` 的文件并读入。输入输出的方式就跟 `cin` 和 `cout` 一样啦。

注意如果定义了自己的文件流的话，那么就不要再傻乎乎地使用 `cin.tie(0)` 啦，要用 `自己定义的文件流.tie(0)`。

还有，如果在程序运行的中途不需要继续读写文件了，那么记得调用 `.close()` 函数，释放文件的各种缓存，也方便其他程序调用。

以下是一个小的例子：

```cpp
ifstream fin("test.in");
ofstream fout("test.out");
int x;
fin >> x;     // 读入 test.in 里面的数
fout << x;    // 写入到 test.out 里面
```

```
fin.close();   // 释放缓存
fout.close();
```

3.3.3.2 freopen

这个也是算法竞赛当中比较推荐的一种，原理就是直接修改标准输入/输出的文件指针，从而达到重定向文件输入输出的效果。

格式：

```
freopen(文件名: char *, 读写方式: char *, 文件指针: FILE *)
```

文件名就是想要定向到的文件，读写方式为 “r” 或 “w”，文件指针就是 `stdin` 或 `stdout`，这也是我们之前一直说的标准输入输出。

```
freopen("test.in", "r", stdin);
freopen("test.out", "w", stdout);
int x;
cin >> x;
cout << x;
fclose(stdin);
fclose(stdout);
```

3.3.3.3 fopen

这个是用得相对比较少的 C 风格传统文件读写，但是比较复杂，需要使用 `fscanf` 和 `fprintf`，读者可以自上网了解。

```
FILE *in = fopen("test.in");
FILE *out = fopen("test.out");
int x;
fscanf(in, "%d", &x);
fsprintf(out, "%d", x);
fclose(in);
fclose(out);
```

3.4 分支结构

在之前的程序中，我们的语句都是一条一条顺序执行的。但是，在本小节就要打破这个规律了。假如要你编写下面一个程序：如果 $x = 100$，输出 1，否则输出 0。是不是在之前都没有学过这种语法？那么快来学习吧！

3.4.1 if 语句

`if` 语句是最简单的分支结构。分支结构就是根据不同的条件去做不同的事。格式为 `if (条件) { }`，大括号里面就跟主函数一样塞一些语句去运行。对于上面的例子，我们可以这样子做：

```
#include <iostream>

using namespace std;

int main() {
  int x;
  cin >> x;
  if (x == 100) {
    cout << 1;
  }
  if (x =/= 100) {
    cout << 0;
```

```
  }
  return 0;
}
```

但是，我们可以发现，虽然我们可以判断 $x = 100$ 了，但是判断 $x = 100$ 不为真还得再写一个 $x \neq 0$。万一判断条件很复杂怎么办？这是，就需要请出我们的 `if-else` 语句了。它的格式如下：

```
if (条件) {
  语句;
} else {
  语句;
}
```

这种语句的逻辑为：如果条件为真，那么执行 `if` 里面的内容，否则执行 `else` 里面的内容。那么，刚才的程序就可以简写为以下形式：

```
if (x == 100) {
  cout << 1;
} else {
  cout << 0;
}
```

那么，如果有多个 `if` 只能满足其中一个，并且需要判断是不是都不满足呢？那么就可以使用我们的 `if-else` 语句（`else` 可以不需要）。

比如，现在需要你编写一个判断 x 的函数，如果 x 为整数，输出 1；如果 x 为负数，输出 -1；否则输出 0。我们当然可以使用三个 `if` 进行判断，但是这样子可读性就会很糟糕。可以这样写：

```
if (x > 0) {
  cout << 1;
} else if (x < 0) {
  cout << -1;
} else {
  cout << 0;
}
```

当然，你也可以将 `else` 改成 `else if (x == 0)`，但是 $x > 0$ 和 $x < 0$ 就排除了 $x \neq 0$ 的所有剩余情况了，因此这种情况直接使用 `else` 会更好一点。

还有更复杂的情况，那么就是 `if` 的嵌套。比如现在给你两个数，一个是 `iq` 表示智商（0 或 1），一个是 `id` 表示情商（0 或 1）。然后根据生活经验判断并输出 `"2B"`, `"3B"`, `"ZB"`, `"Clever"`，那么我们就需要使用到 `if` 的嵌套了，因为嵌套的可读性要更高（不是盲目地嵌套）。

比如可以这样写，而不是写四个 `if` 排一起。

```
if (iq == 1) {
  if (id == 1) {
    cout << "Clever";
  } else {
    cout << "2B";
  }
} else {
  if (id == 1) {
    cout << "ZB";
  } else {
    cout << "3B";
```

```
  }
}
```

我们还可以使用之前的逻辑运算符。比如需要判断 $a = 100$ 和 $b = 10$ 是否同时满足条件，那么就可以使用 `a == 100 && b == 10` 来进行判断。

3.4.2 switch 语句

`switch` 语句是比较冷门的一种分支语句，但是当判断条件比较多的时候，`switch` 语句说不定比一大串的 `if-else` 效果会更好。格式：

```
switch (需要判断的数) {
  case 判断数 1:
    语句;
    break;
  case 判断数 2:
    语句;
    break;
  default:
    语句;
}
```

翻译成 `if` 语句就是下面这样子的：

```
if (需要判断的数 == 判断数 1) {
  语句;
} else if (需要判断的数 == 判断数 2) {
  语句;
} else {
  语句;
}
```

可以看到，`switch` 语句是逐条判断是否等于指定数的，并在判断成功时候运行语句吧并退出。`default` 就相当于 `if-else` 里面的 `else`。

易错点：首先，`case` 里面必须加上 `break` 语句，这个语句的作用可以在循环结构里面找到，如果不加的话那么就会发生奇奇怪怪的错误；其次，`判断数` 必须是常量，变量统统不行。而且，`switch` 语句还只能判断基础数据类型，因此非常鸡肋。

好吧，有时候还是很有用的。

3.5 循环结构

如果你遇到了一个问题，给定 n，打印 $(1, n)$ 当中的所有整数，那么你该如何求解？难道一行一行地输出？因为在之前的所有学过的知识当中，我们并没有学过循环处理的语法，根本没有充分利用到计算机超快的运算速度。接下来，好渴鹅就会详细讲解关于 C++ 循环结构的知识。

3.5.1 while 语句

`while` 在英语当中是“当”的意思，而在 C++ 里面就是当某个条件成立时不断运行语句。语法为 `while (条件)`。其中，条件是一个布尔表达式，为真或假。注意，如果条件一直为真而循环里面没有退出语句的话，那么就会一直进行循环，这种循环被称为“死循环”。

例如，以下代码就是一个“死循环”，会一直打印 `"Hello world!"`：

```
while (true) {
  cout << "Hello world!" << endl;
}
```

而对于上面的例题，我们只需要使用一个“循环变量” i 即可完成操作。代码：

```
cin >> n;
int i = 1;
while (i <= n) {
  cout << i << ' ';
  i++;
}
```

如果输入 $n = 5$ 的话，那么循环步骤为：

- $i = 1$，满足 $i \le n$，输出 i，$i \leftarrow i + 1$；
- $i = 2$，满足 $i \le n$，输出 i，$i \leftarrow i + 1$；
- $i = 3$，满足 $i \le n$，输出 i，$i \leftarrow i + 1$；
- $i = 4$，满足 $i \le n$，输出 i，$i \leftarrow i + 1$；
- $i = 5$，满足 $i \le n$，输出 i，$i \leftarrow i + 1$；
- $i = 4$，不满足 $i \le n$，退出循环。

可以看到我们成功完成了任务。

3.5.2 do-while 语句

你肯定会问，do-while 语句跟 while 语句到底是有什么区别呢？不久多了一个 do 吗？没错，区别仅仅在 do 上面，但是它们的执行方式确有很大区别。

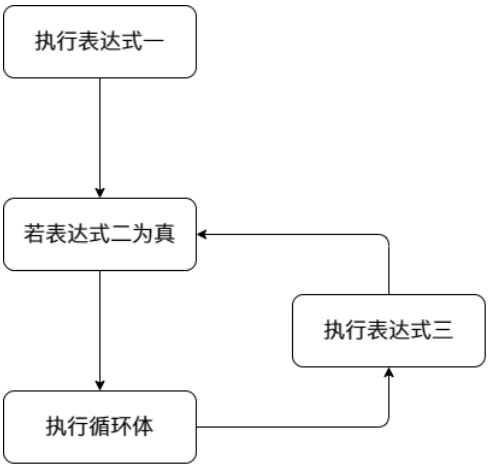
while 语句是当条件成立就一直执行，而 do-while 语句则是先执行在判断是否满足条件。举个简单的例子，如果条件一开始就不成立，那么 while 循环就不会执行，但是 do-while 循环则会至少进行一遍。比如将上面的代码改成 do-while 循环：

```
cin >> n;
int i = 1;
do {
  cout << i << ' ';
  i++;
} while (i <= n);
```

你就会发现 $n + 1$ 也输出出来的。因此，使用 do-while 循环在这种情况下必须将 $\le$ 改成 $<$。而这样也不严谨，因为当 $n < 1$ 的时候也会输出。你是不是觉得 do-while 循环一无是处呢？也不一定，在特定环境下，do-while 循环可以更好地处理问题。

3.5.3 for 语句

for 循环应该是所有循环语句里面最复杂的那个了。语法为 for（表达式一；表达式二；表达式三）。首先在循环开始之前，执行表达式一的内容，然后当表达式二成立的时候执行循环体里面的内容，每一次执行完之后就执行表达式三，然后一直循环。大概是这样子的：



就比如上面的代码，可以简化成如下的形式：

```
cin >> n;
for (int i = 1; i <= n; i++) {
  cout << i << ' ';
}
```

可以看到，变量 i 放到了第一个表达式里面进行定义，判断条件没有变，循环变量的增量则是放在了第三个表达式里面。其实，只要是循环最后面的若干条指令都可以全部放进第三个表达式里面。

注意 `for` 循环定义的循环变量只能够在循环里面使用，而 `while` 循环在循环体外面定义的循环变量则在后面的循环中也可以使用。

3.5.4 break 与 continue

如果在循环当中不需要继续循环了呢？我们只需要使用一个布尔类型的变量，在需要跳出循环的时候将其赋为真，然后在循环条件里面判断就行了。

但是如果是在循环当中想要跳出怎么办？用上面的方法必须在循环重置的时候才会跳出，循环后面的语句还会继续执行。因此我们可以使用 `break` 关键字，这会跳出最里层的循环。例如，当 i 为质数时跳出。（假设 `isPrime(x)` 可以判断 x 是否为质数）

```
int i = 1;
while (true) {
  if (isPrime(i)) {
    break;
  }
  i++;
}
```

而 `continue` 呢则是重置当前的循环（同样需要判断条件），后面的所有语句都省略。注意 `for` 循环的增量在 `continue` 之后还是需要执行的。

就比如，输入 n 个数，当输入的数为偶数时，就直接输出，否则把这个数逐位反转之后输出。代码怎么写？

首先，我们知道直接输出对比反转之后输出那肯定是直接输出简单很多，因此我们使用一个 `if` 来判断输入的数是否是偶数（偶数的基本判断条件是是 2 的倍数，也就是对 2 取模为 0），如果是的话那么就直接输出，然后 `continue`，也就是跳过循环后面的所有语句。

接下来我们再考虑如何对一个数反转输出，注意听好了。首先，我们知道，一个数对 10 取模可以得到这个数的最后一位（十进制下），这里使用了十进制的基本原理；而对一个数除以 10 可以去掉最后一位（十进制下）。因此，当这个数不等于 0 时，我们就不断执行这个操作，就可以对这个数的每一位倒序输出了。

```
cin >> n;
for (int i = 1, x; i <= n; i++) {
  cin >> x;
  if (x % 2 == 0) {
    cout << x << '\n';
    continue;
  }
  for (; x; x /= 10) {
    cout << x % 10;
  }
  cout << '\n';
}
```

补充：在最外层的 `for` 循环的第一个表达式里面，我们不仅定义了循环变量 i，还定义了用于输入的局部变量 x。内层的 `for` 循环可能对于初学者来说有些难以看懂，其实这也是 `for` 循环的常用用法，转化成 `while` 是这样的（在布尔表达式里面直接写整数等同于判断这个整数不等于 0）：

```
while (x) {
  cout << x % 10 << ' ';
  x /= 10;
}
```

3.6 数组

现有这么一道题，给定 n 个数，然后倒序输出他们。以我们之前所学的知识是完全做不出来的，因为在之前我们都是使用一个变量来表示一个值，但若是 n 很大呢？那我们就没有办法了，这时候就需要使用一种名为“数组”的类型。

3.6.1 定义与使用

定义长度为 n 的数组 a 可以采取以下的格式（在 C++ 11 之前 n 必须是常量，否则可能发生编译错误）：

```
type a[n];
```

那么如何访问呢？可以通过 `a[i]` 来访问数组当中下标为 i 的元素。为什么是下标 i 而不是第 i 个元素呢？其实是因为 C++ 当中长度为 n 的数组下标的范围是**从零开始**的，即 $[0, n-1]$，因此访问数组 a 的第一个元素得写 `a[0]`而非 `a[1]`。

数组的存储在内存当中一定是连续的，而非像普通的变量一样可能会存储在内存的不同地方。这样的设计是为了方便申请内存和加快缓存速度。

回到刚刚的题目，是不是就非常简单了。我们可以使用 `for` 循环正序输入元素，再用 `for` 循环倒序输出元素。代码：

```
int main() {
  int n;
  cin >> n;
  int a[n];
  for (int i = 1; i <= n; i++) {
    cin >> a[i];
```

```
  }
  for (int i = n; i >= 1; i--) {
    cout << a[i] << ' ';
  }
  return 0;
}
```

补充：为了更贴合人类使用，在一般的使用过程中一般都会把数组的长度设为 $n+1$，然后将 a_1 当做第一个元素。一般我们都会定义长度为常量的数组，相对来说会要更稳定一些。数组也要尽量定义在函数的外面，使用堆而非栈，可以开更大的数组。

3.6.2 多维数组

在一些特殊的情况下，例如需要输入矩阵的时候，就需要使用多维数组了。二维数组是最简单的多维数组。例如以下的例题：

给定一个 $n \times n$ $(1 \le n \le 1000)$ 的数组 a，将其的行和列反转后输出，通俗一点就是每一行变成列，每一列变成一行，原来的 $a_{i,j}$ 就变成了 $a_{j,i}$。

这时候就需要使用我们的二维数组了，定义方式和一维数组差不多，为 `type a[n][n]`。使用 `a[i][j]` 即可访问数组 a 的第 i 行第 j 列（也可以颠倒，这对于使用来说不重要）。

那么上面的代码就很简单了，输出的时候只需要第 i 行变成第 i 列，第 j 列变成第 i 行就可以了。注意输出完一行之后要换行。

```
const int kMaxN = 1001;  // 数组大小，注意我们是从一开始的，因此要加一
int a[kMaxN][kMaxN], n;  // 数组和整数定义

int main() {
  cin >> n;
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
      cin >> a[i][j];
    }
  }
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
      cout << a[j][i] << ' ';
    }
    cout << '\n';
  }
  return 0;
}
```

3.7 函数

在 C++ 当中，主函数是程序的入口，代码都会逐行运行。但是如果有一些语句重复了很多次，那么就可以使用函数来进行封装，提高代码的可读性（读起来不会骂人）和复用性（重构起来不会骂人）。

C++ 在标准库当中定义了很多的内置函数，只需要几行命令引入并调用就行了，比如算数平方根、三角函数等数学函数。我们也可以自定义函数，来实现自己想要的功能。

函数还有很多常用的叫法，比如方法、程序等等，英文为 Function。

3.7.1 标准库

假如有一道题，是这样的：输入一个实数 x $(x \geq 0)$，要求得到 $\sqrt{x}$ 并输出。$\sqrt{x}$ 为 x 的算数平方根，意思是得到一个 y 使得 $y^2 = x$。由于在实数范围内没有数的平方是负数，因此 $\sqrt{x}$ 必须大于等于 0。

那么我们就可以调用 C++ `cmath` 库当中预定义的 `sqrt` 函数（英文全程为 Square Root）来求出算术平方根。代码如下：

```cpp
#include <iostream>
#include <cmath>

using namespace std;

double x;

int main() {
  cin >> x;
  cout << sqrt(x) << '\n';
  return 0;
}
```

C++ 还有更多常用的内置函数，在后面中会对其中的一些进行使用，也可以上网搜索更多 C++ 的内置函数。

3.7.2 自定义函数

在 C++ 当中，不仅可以使用内置函数，还可以自己定义函数。语法如下，其实也跟 `main` 函数差不多：

```cpp
return-type function-name(type args, ...) {
  function-body;
  return return-value;
}
```

比如，定义一个名叫 `times2` 的函数，并传入参数 x，返回 x 的两倍就可以这样定义：

```cpp
int times2(int x) {
  return 2 * x;
}
```

然后就可以调用这个函数了，例如 `int x = times2(1)` x 就会等于 2，因为**实际参数** 1 传到 `time2` 里面会变成**形式参数** x，然后我们返回了 $2 \times x$，因此在外面的变量 x 就接受了这个值，即 2。

函数除了返回普通的类型，还可以返回特殊的类型 `void`，这个类型称之为**无类型**，这代表着函数只是一个过程，没有任何返回值，因此返回语句可以不写。如果需要在特定地方结束的话，可以使用 `return;` 或 `return void();` 语句退出。

一般纯过程的函数都会改变外部变量，不然这个函数是没有任何意义的。例如，以下函数就是一个纯过程的函数：

```cpp
void seyHello() {
  cout << "Hello world!" << endl;
}
```

3.7.3 函数递归

调用本身函数的函数就叫做递归函数，递归函数可以让代码变得更加简洁，但是可能会带来较大的内存负担并拖累程序的运行速度。

比如，你需要定义一个函数 $f(x)$，使得可以计算 $x!$ 并返回。($x!$ 为 x 的阶乘，等同于 $1 \times 2 \times ... \times x$ 或 $\prod_{i=1}^{n} i$) 怎么办？是不是可以使用我们之前学过的循环结构！

```
int f(int x) {
  int fac = 1;
  for (int i = 1; i ⩽ x; i++) {
    fac *= i;
  }
  return i;
}
```

但是这样子代码根本就不简洁。考虑数学上的递归，是不是有 $x! = (x-1)! \times x$？但是这样子会地址递归下去，因此我们需要设立递归边界。因为 $1! = 1$，因此我们就把递归边界设为 $x = 1$。代码：

```
int f(int x) {
  if (x == 1) {
    return 1;
  }
  return f(x - 1) * x;
}
```

让我们来模拟以下调用 $f(3)$ 递归的操作：

- **第一个函数**传入了参数 $x = 3$。由于 $x \neq 1$，因此返回 $f(x-1) \times x$，调用第二个函数。
- **第二个函数**传入了参数 $x = 2$。由于 $x \neq 1$，因此返回 $f(x-1) \times x$，调用第三个函数。
- **第三个函数**传入了参数 $x = 1$。由于 $x = 1$，因此返回 1，回到第二个函数。
- **第二个函数** $x = 2$，$f(x-1)$ 已经计算完了等于 1，因此返回 $1 \times 2 = 2$，回到第一个函数。
- **第一个函数** $x = 3$，$f(x-1)$ 已经计算完了等于 2，因此返回 $2 \times 3 = 6$，递归完成。

是不是非常烧脑？其实我们只需要掌握 $x! = (x-1)! \times x$ 就行了，这个公式是这样推导出来的：

$$\begin{aligned} x! &= 1 \times 2 \times ... \times x \\ &= [1 \times 2 \times ...(x-1)] \times x \\ &= (x-1)! \times x \end{aligned}$$

在递归时只需要将其转为 $f(x) = f(x-1) \times x$，并设好边界条件（不然会一直计算 $f(x-1)$）就行了。是不是恍然大悟？

3.7.4 数组传参

需要特别注意的是，C++ 的数组是在所有数据类型当中最特殊的那一个，其他数据类型当做形式参数的时候都会值拷贝，唯独数组是直接引用传递。这就代表着在函数内部改变数组的值在外面也会起到作用。这样看来，数组是不是就跟后面所学的指针一样，只是一个指向数组头部的指针呢？也不是，因为在 `sizeof` 的时候，数组返回了整个数组的大小，而指针只返回了指针本身存储在内存当中的大小。

因此，使用数组传参的时候一定要多加小心，防止出现非预期的错误。

3.8 指针

3.8.1 指针的认识

什么是指针？其实，在 C++ 当中的变量，全都是存储在内存里面的，而内存都会有一个“地址”用于访问值。C++ 编译器就处理了变量对应的地址，使得你使用变量就可以访问这个变量的地址上面的值。

而指针就是专门指向一个地址进行访问的，可以使一些复杂的数据结构变得更简单、灵活，并且指针还可以申请堆上面的内存。

在 C 语言当中是没有引用的，而函数传参都是直接转递值，这就代表着在函数里面更改参数的值在外面不会起作用。这种情况下，就需要使用指针来直接操作地址了。

指针本身只是一个指向数据的数据类型，本身也是变量，所以还有指向指针的指针——双重指针，这都是比较复杂的内容了。虽然它很难，在实际操作中极易造成内存溢出，但是只要你认真学习，一定能够熟练掌握。

3.8.2 使用指针

定义指针的方法特别简单。在过去，我们定义一个 `int` 类型的变量是不是直接 `int a;` 就行了呢？指针的定义也不复杂，指向一个 `int` 类型的地址的指针可以使用 `int *p;` 来定义。

注意星号只对后面的一个变量起作用，这就代表着 `int *p, a;` 这种定义方法只有 p 是指针，a 就只是普通的变量。因此，在定义的时候，尽量将星号写在靠近变量名的那边而非类型的那边。

想要让你的指针指向一个指定的变量的地址，我们需要用到取地址符 `&`。诶，这好像有点似曾相识？没错，就是位与运算，但是当只有一个操作数的时候就变成了取地址符。

比如以下代码：

```
int a = 1;
int *p = &a;
```

这段代码首先定义了一个 `int` 类型的普通变量 a，并将其初始化为 1。然后定义了一个指向 `int` 的指针 p，并把它指向的地址初始化为 a 的地址。

想要通过指针指向的地址访问这个地址上面的元素，需要使用解引用符 `*`。你没有看错，就是乘法的那个星号，但是跟取地址符一样，若只有一个操作数的话就变成了解引用符。

比如，上面的代码增添上下面这一句：

```
cout << *p << '\n';
```

就能发现程序输出的值为 1，因为指针 p 指向的地址是变量 a 的地址，而 a 的值此时为 1，因此就会输出 1。需要注意的是，由于指针是直接操作内存，因此更改解引用后的指针的值就等同于直接修改变量：

```
int a = 1, *p = &a;
*p = 2;
cout << a << '\n';
```

可以看到，程序输出了 2。因为 p 指向的是 a 的地址，而我们将这个地址上面的数修改成了 2，因此输出 a 的时候值就会也变成 2。

当你只想要使用指针，而不是再定义一个毫无意义的变量来供指针指向的话，可以使用 `new` 关键字来申请一个空间。比如：

```
int *p = new int;
```

这样子 p 就可以指向一个新申请的空间，而这个空间是“匿名”的。

3.8.3 连续空间

在刚才的尝试当中，我们的指针都是只指向了一个元素的，那么，如何让指针指向多个元素呢？其实这是不可能的，但是我们可以指向一段连续元素的头，这样子就可以达到指向多个元素的效果。

申请一段连续空间的语句是 new，例如以下代码：

```
int *a = new int[5];
```

这段代码就是定义了一个指向 int 的指针 a，并申请了长度为 5 的连续 int 空间给它，这样子它就能称为一个“数组”了，同样可以使用中括号访问元素。

```
a[1] = 1;
cout << a[1] << '\n';
```

需要注意的是，new 一般都是申请的堆空间，因此程序在运行时不会自动清理，需要手动销毁。销毁单个指针指向的元素为 delete，销毁指向的连续一段空间为 delete[]。例如，销毁 a 指向的连续一段的空间的代码为：

```
delete[] a;
```

3.8.4 函数指针

3.8.5 引用

3.9 更多技巧

3.9.1 位运算拓展

这里是关于位运算的更多奇技淫巧。

位运算一般都是有三种作用，例如：

- 题目当中明确说明了需要使用位运算。
- 用来表示集合，但是如果位数较多需要使用 STL bitset。
- 用高效的指令集取代某些低效的算法，例如求 2^n。

值得一提的是，大部分 Noip 系列的比赛都会在编译时默认开启最大速度优化，也就是 O2。这时用 >> 1 除以二反而没有任何速度提升，因为编译器已经将 / 2 优化成 >> 1 了。

3.9.1.1 计算 2^n

因为在二进制当中是逢二进一的，因此 1 后面跟 n 个 0 就等同于 2^n。这点我们人类使用的十进制也是有所体现的，在十进制当中 1 后面跟 n 个 0 就等同于 10^n。

因此，我们对 1 左移 n 位，使得后面空出 n 个 0，也可以达到 2^n 的效果。如果需要快速计算 $n \times 2^m$，那么不用计算完 2^m 再去乘 n，实际上直接将 n 左移 m 位就行了。

```
int pow2(int n) {
  return 1 << n;
}

int pow2(int n, int m) {
  return n << m;
}
```

如果是要计算 $\frac{n}{2^m}$ 的话，那么直接将 n 右移 m 位就行了。但是需要注意的是，这个的结果是向下取整而非向 0 取整的，在使用过程中需要注意。

3.9.1.2 判断奇偶性

假如给你一个 n，要你判断 n 是否是偶数，你会怎么做？是不是对 2 取模，然后判断是不是 0 啊？实际上也可以对 1 按位取与。因为 1 的前面都是 0，因此结果的前面也全都是零；如果 n 的最后一位为 1 的话，那么结果就是 1，否则就是 0。这也利用了二进制的基本性质。

```
bool isEven(int x) {
  return x & 1 == 0;
}

bool isOdd(int x) {
  return x & 1 == 1;
}
```

3.9.1.3 表示集合

位运算让我们能够用整数来表示集合。例如，在 C++ 当中 `int` 类型通常都是 32 位的，这让我们能够用它来表示范围仅有 1~32 的集合。虽然看起来位数并不多，但是奈何不住它快啊！常见的集合表示方法都是用一个二叉树来，既耗内存又没有速度优势，而我们使用整形来表示集合可以很快地进行集合相关的操作，必要时还可以当下标使用。

需要注意的是，整数的位数是从右往左的，最后一位是第 0 位；而我们这里所说的位数虽然也是从右往左，但是最后一位是第 1 位。

比如想要让 x 的第 i 位变成 1，那么只需要执行 `x |= 1 << (i - 1)` 就行了。这句话的原理就是，1 占了最后一位，我们左移 $i-1$ 位就可以让这个 1 变到第 i 位，然后让 x 或上去，因为其它位或 0 都没变，而这一位或 1 就一定能变成 1。

同理，想要让第 i 位变成 0 只需要执行 `x &= 0xffffffff - (1 << (i - 1))`。注意 `0xffffffff` 是一个十六进制数，而这个数字在 2 进制里面正好为 32 个 0（感兴趣的可以学习一下十六进制转二进制），我们减去 `1 << (i - 1)` 是为了让第 i 位变成 0，这样子就变成了除了第 i 位全都是 1 的了。而任何数和 1 与都是不变的，任何数和 0 与都会直接变成 0。因此，该操作就能成功地将第 i 位改成 0。

如果是要取交集的话，那么位运算就已经自带位与运算了；如果是要取并集，就可以使用位或运算。总之，用整数来表示集合还是有很多好处的，但是需要有更好的水平才行，

4 算法基础

4.1 引入

4.1.1 复杂度

时间复杂度和空间复杂度是衡量一个算法在运行效率和空间效率上的重要标准。

同一个算法在不同的计算机、操作系统上面运行的速度会有一定差别，而实际测量算法执行速度非常麻烦，因此通常我们都不会考虑一个算法具体执行了多久的时间，而是算法运行大约需要的基本操作次数。基本操作次数通常包括，加减乘除、赋值与访问等。

不说太多没用的，我们直接来说时间复杂度。例如，有一个算法是有两层循环，每一层循环都是 1~n，那么这个算法的时间复杂度就为 $O(n^2)$，用大 O 表示。循环内部可能还会有一些运算，但是如果全部都是常数级的运算的话，那么就可以忽略它们。

在更多时候，算法可能会不止进行一次。在这种情况下每一次进行算法的平均复杂度就叫做均摊复杂度。

空间复杂度既是对算法所用空间的描述。例如，一个算法使用了长度为 n 的数组，那么这个算法的空间复杂度就是 $O(n)$ 的；若使用了 $n \times m$ 的二维数组，那么这个算法的空间复杂度就是 $O(n \times m)$ 的。

4.1.2 最大数问题

现在我们就尝试来设计一个最最简单的算法吧。给定 n $(n \geq 2)$ 和长度为 n 的数组 a，要求设计一个函数 $\max(a, n)$，返回 a_1~a_n 当中的最小值。

让我们来思考一下。首先，max 函数是具有结合律的。我们可以把它看做一个二元运算符 $\oplus$，$a \oplus b \oplus c$ 一定等于 $a \oplus (b \oplus c)$。因此，$\max\limits_{i=1}^{n} a_i = \max\left(\max\limits_{i=1}^{n-1} a_i, a_n\right)$，也就是 a_1~a_n 的最大值等于 a_1~a_{n-1} 的最大值和 a_n。而 a_1~a_{n-1} 的最大值又等于 a_1~a_{n-2} 当中的最大值和 a_{n-1} 的最大值。因此我们只需要从头望后扫一遍，每次看一看新来的那个数是否比之前的数大，如果是的话就把之前的数改为这个数就行了。

由于只需要 1~n 的循环，因此该算法的时间复杂度是 $O(n)$ 的。由于 a 数组是通过传参传来的，因此该算法的空间复杂度是 $O(1)$ 的，即不使用额外非常数空间。

```
int max(int a[], int n) {
  int mx = a[1];
  for (int i = 1; i <= n; i++) {
    if (a[i] > mx) {
      mx = a[i];
    }
  }
  return mx;
}
```

当然，该算法也可以使用递归实现，因为 $\max\limits_{i=1}^{n} a_i = \max\left(\max\limits_{i=1}^{n-1} a_i, a_n\right)$。

```
int max(int a[], int n) {
  if (n == 1) {
    return a[1];
  }
  return max(max(a, n - 1), a[n]);
}
```

可以看到，我们使用了同名的 max 函数，但是这里并没有任何影响。因为标准库当中的 max 函数两个参数必须都是同一类型的参数，而我们定义的 max 的参数列表不一样。

4.1.3 平均数问题

写完最大数问题，趁着火苗正旺，赶紧来写一下平均数问题吧。给定你一个长度为 n 的整数数组 a，你需要实现一个函数 $\text{ave}(a, n)$，表示求 a_1~a_n 的平均数，返回 `double` 类型。定义 a_1~a_n 的平均值为 $\sum\limits_{i=1}^{n} a_i \times \frac{1}{n} = (a_1 + a_2 + ... + a_n) \times \frac{1}{n}$。

很简单，我们只需要照着式子写就行了。先求出 a 的和。然后将这个数除以 n 就行了。

```
double ave(int a[], int n) {
  int sum = 0;
  for (int i = 1; i <= n; i++) {
    sum += a[i];
```

```
    }
    return 1.0 * sum / n;
}
```

注意 `int` 类型和 `double` 类型的乘积是 `double` 类型，因此最后一行的代码（`1.0 * sum`）实际上就是将 `sum` 转化为 `double` 类型，而 `double` 类型除以 `int` 类型仍然为 `double` 类型，也就是小数除法。

4.2 枚举

4.2.1 暴力枚举

枚举是一种通过暴力枚举状态来猜测可能得解的一种解题策略。通常来说，我们可以限定枚举的范围让它的复杂度更高。虽然这是一种非常基础的算法，但是后面的很多算法都是基于枚举来实现的。

> 好渴鹅想要吃一条美味程度为 n 的鱼。现在好渴鹅一共有 10 种调料，每一种调料都可以放 1~3 克，而一条烤鱼的美味值就是所放调料的克数之和。你需要输出所有美味值为 n 的方案，具体就是输出每一种调料放多少克。

我们可以枚举每一种调料放多少克，最后如果加起来正好等于 n 了就输出。

```
for (int i = 1; i ⩽ 3; i++) {
  for (int j = 1; j ⩽ 3; j++) {
    for (int k = 1; k ⩽ 3; k++) {
      for (int l = 1; l ⩽ 3; l++) {
        for (int p = 1; p ⩽ 3; p++) {
          for (int o = 1; o ⩽ 3; o++) {
            for (int u = 1; u ⩽ 3; u++) {
              for (int m = 1; m ⩽ 3; m++) {
                for (int y = 1; y ⩽ 3; y++) {
                  for (int t = 1; t ⩽ 3; t++) {
                    if (i + j + k + l + p + o + u + m + y + t == n) {
                      cout << i << ' ' << j << ' ' << k << ' ' << l << ' ' << p << ' '
<< o << ' ' << u << ' ' << m << ' ' << y << ' ' << t << '\n';
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

这里使用了我们之前循环结构里面的循环嵌套，也就是枚举第一个的所有可能，在其之上枚举第二个的所有可能，……，直到枚举完最后一个的所有可能。这样子的方法非常直观，并且能够枚举出所有情况，完全正确。

根据之前学习的复杂度，我们能够求出这个算法的时间复杂度是 $O(3^n) = O(1)$ 的，但是如果所放克数可以是 1~k 的话，那么该算法的时间复杂度就会为 $O(k^n)$，但是这样子枚举出所有情况真的好吗？有没有可能枚举了多余的元素呢？

4.2.2 减小枚举范围

我们发现，当已放调料的克数已经大于 n 的时候，在往下面枚举已经是没有意义的了。因为所有调料的所放克数都必须是 1~3 的整数，所以后面放完调料的总克数一定会比 n 要更大。因此在这种情况下，我们直接 `break` 退出循环即可。或者是在循环的时候就不遍历那么多，直接用循环条件限制就行了。

```
for (int i = 1; i ⩽ 3; i++) {
  for (int j = 1; j ⩽ min(3, n - i); j++) {
    for (int k = 1; k ⩽ min(3, n - i - j); k++) {
      for (int l = 1; l ⩽ min(3, n - i - j - k); l++) {
        for (int p = 1; p ⩽ min(3, n - i - j - k - l); p++) {
          for (int o = 1; o ⩽ min(3, n - i - j - k - l - p); o++) {
            for (int u = 1; u ⩽ min(3, n - i - j - k - l - p - o); u++) {
              for (int m = 1; m ⩽ min(3, n - i - j - k - l - p - o - u); m++) {
                for (int y = 1; y ⩽ min(3, n - i - j - k - l - p - o - u - m); y++) {
                  for (int t = 1; t ⩽ min(3, n - i - j - k - l - p - o - u - m - y);
t++) {
                    if (i + j + k + l + p + o + u + m + y + t == n) {
                      cout << i << ' ' << j << ' ' << k << ' ' << l << ' ' << p << ' '
<< o << ' ' << u << ' ' << m << ' ' << y << ' ' << t << '\n';
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

值得一提的是，当循环来到了最后一层，最后一种调料的克数已经能够确定了，因此就可以直接进行判定。请读者自行补充。

4.3 贪心

4.3.1 排序法

贪心算法是用计算机来模拟我们人类在面对问题时做出决策的过程的一种算法。并且，它在进行决策的时候总是目光短浅地选择最优的操作，来尽可能地取得最优解。

并不是所有的题目都可以由贪心实现，但是贪心是众多算法的基础。比如搜索中的“最优化剪枝”还有 Dijkstra 算法的中心思想，里面都有一些贪心的元素。

贪心算法往往比较简单，因为人类本身就是贪心的。就比如接下来的一道题：

> 给定 n 个数，你现在要选择最多 m 个数，使得选择的数相加和最大。输出这个最大的相加和。(可能会有负数)

这个问题，就跟你做作业一样，选择一定数量的作业，那么我们每一次都选择量最少的作业，那么最后选择的作业加起来一定也是最少的。本题也一样，每一次都选择最大的那一个数，那么加起来一定也是最多的。

注意这里可能会有负数，但是加负数是没有用的。那么如果我们加到了负数，就说明后面的数都是负数了，那么我们就不加，直接退出即可。

代码当中使用了 `algorithm` 库当中的排序函数 `sort`，并使用了 `greater<int>()` 函数返回的大于号比较器实现从大到小排序。具体可以看附录。这里简单讲解一下：

`sort(begin, end, comp)` 就表示对 `[begin, end)` 之间的所有元素进行排序，比较器为 `comp`。如果不传入该参数的话，就是默认从小到大排序。

```
int solution() {
  cin >> n >> m;
  for (int i = 1; i ⩽ n; i++) {
    cin >> a[i];
  }
  sort(a + 1, a + n + 1, greater<>());   // 尖括号内可以省略
  for (int i = 1; i ⩽ m; i++) {
    ans += a[i] * (a[i] > 0);
  }
  return ans;
}
```

> 现在有 n 个人排队接水，第 i 个人接水的时间为 a_i。现在要你重新排这 n 个人的位置，使得这 n 个人的平均等待时间最少。输出重新排列后的顺序以及平均等待时间。

首先，看到这题，我们的人脑很快就可以想出一个解法：叫打水更快的人站到前面不就行了，这样子后面的人等的不也少了吗？可是，这个解法怎么证明是对的呢？

我们可以设总的打水时间为 S。显然 S 越小，平均值 $\frac{S}{n}$ 也会越小，因此我们需要最小化 S。排列完之后，第 i 个人需要等待前面所有人和自己大水的所有时间，因此第 i 个人总的等待时间就为 $\sum\limits_{j=1}^{i} a_j$。

所以：

$$S=\sum_{i=1}^{n}\sum_{j=1}^{i}a_j=(a_1)+(a_1+a_2)+...+(a_1+a_2+...+a_n)$$

我们发现，a_1 需要相加 n 次，而 a_2 需要相加 $n-1$ 次，但是 a_n 却只需要相加 1 次。那么结果很显然了，让小的加更多次显然是更优秀的。因此我们需要吧时间短的排在前面。

代码使用了结构体和自定义 STL 排序，具体请看类与对象当中的“结构体”章节。

4.3.2 后悔法

4.4 排序

4.4.1 比较排序

4.4.1.1 选择排序

4.4.1.2 冒泡排序

4.4.1.3 插入排序

4.4.1.4 归并排序

4.9.4 折半搜索

5 类与对象

5.1 类的定义

5.2 成员与成员函数

5.3 构造函数与析构函数

5.4 静态成员

5.5 重载运算符

5.6 友元函数

5.7 继承

5.8 虚函数

5.9 抽象

6 附录

6.1 关于命令行的使用

大部分操作系统都是有图形化的，图形化对于普通用户来说是十分便利的，不需要学习过多复杂的指令等。但是，很多时候命令行的操作对于图形化应用来说会要更简单，而且很多服务器只有命令行。因此，学会命令行的使用非常重要。

6.1.1 打开命令行

不同系统打开命令行的方式是不一样的。

- **Windows 系统：**在开始菜单栏搜索命令提示符就可以打开了，命令提示符就是 Windows 系统下的命令行。
- **Linux 系统：**在大多数的 Linux 发行版当中，使用 `Ctrl + Alt + T` 组合键就可以方便地打开命令行。
- **MacOS：**使用启动台，或是聚焦搜索可以直接打开。

6.1.2 Cmd 与 PowerShell 的区别

非 Windows 用户可以忽略。

- **外观方面：**在 Cmd 窗口当中所有文字的颜色都是一模一样的，没有语法高亮，而在 PowerShell 关键字等不同的文字就会被高亮成不同的颜色。
- **命令方面：**Cmd 窗口是 Windows 独有的，很多 Linux 常用的命令都没有支持；而 PowerShell 就可以支持，例如 `ls` 等命令。还有一个显著的区别是在 Cmd 下运行命令默认会索引到当前目录，而 PowerShell 则不会，这导致假设你需要运行 `main.exe` 在 PowerShell 下必须写为 `./main.exe`（或反斜杠）。

- **其他区别：**PowerShell 是基于 .NET 面向对象的，集成了更多的功能。Cmd 是默认在所有 Windows 里面集成的，而 PowerShell 在最新几代里面才会集成。

以下所有命令均为在 Linux 下可以正常执行的，Windows 可能会造成一定差异。

6.1.3 基本命令

- `cd` 用于切换当前工作目录，可以输入相对路径或绝对路径。
- `ls` 查看当前目录下的所有文件。
- `mkdir` 用于在当前工作目录下创建文件夹。
- `rm` 删除目录或文件
- `mv` 移动目录。
- `cp` 拷贝目录。
- `touch` 新增文件，另一种受欢迎的办法是使用 `vim` 创建。
- `vim`（在老版 Linux 中只有 `vi`）编辑文件，没有则创建。
- `sudo` 暂时获得 Root 权限，需要输入密码。

还有更多命令可以上网搜索，一般情况下上面的命令就已经完全够用了。

6.1.4 GCC 基本命令

在命令行中，若已安装 GCC，则可以编译 C 或 C++ 文件（C++ 文件需要使用 `g++` 命令编译）。比如，编译 `main.cpp` 文件并输出二进制文件到当前目录下的代码为（不用输入 `$`，`$` 是输入命令的提示符）：

```
$ g++ main.cpp
```

该操作会在当前目录下生成可执行文件，一般名称为 `a.out`，可以通过 `ls` 命令查看。若想执行该可执行文件，可以使用键入 `./a.out` 完成操作。

若想自定义输出的可执行文件的名称，可以添加 `-o` 参数，后面跟着要重命名的名称。例如，一下操作可以编译 `main.cpp` 源文件，并在当前目录下生成 `app` 可执行文件。

```
$ g++ main.cpp -o app
```

在正式的比赛当中，为了让程序的运行速度更快，一般都会在编译时开启优化。比如可以加入 `-O2` 参数来开启最大速度优化。以下代码可以在对 `main.cpp` 源文件进行编译，并输出 `main.exe` 可执行文件的情况开启 `-O2`。

```
$ g++ main.cpp -o main.exe -O2
```

同样，`g++` 默认的 C++ 版本一般都是编译器所支持的最高版本，而不同的版本之间可能会有一些细微的差别。CCF（中国计算机协会）规定了 C++ 标准应为 ISO C++ 14，并开启 O2 优化。为了获得跟 CCF 类似的体验，可以使用如下的编译指令：

```
$ g++ main.cpp -o main.exe -std=c++14 -O2
```

6.2 STL

6.2.1 算法库

6.2.1.1 algorithm

6.2.1.2 numeric

6.2.2 容器库

6.2.2.1 array

6.2.2.2 vector

6.2.2.3 list

6.2.2.4 forward_list

6.2.2.5 set

6.2.2.6 map

6.2.2.7 deque

6.2.2.8 unordered_set

6.2.2.9 unordered_map

6.2.2.10 stack

6.2.2.11 queue

6.2.2.12 priority_queue

6.3 C++ 11 新特性

6.3.1 语言方面

6.3.1.1 auto & decltype

6.3.1.2 左右值

6.3.1.3 范围 for

6.3.1.4 Lambda 表达式

6.3.1.5 constexpr

6.3.2 类与对象方面

6.3.2.1 final & override

6.3.2.2 default & delete

6.3.2.3 explicit

6.3.2.4 模板的改进