

C++快速入门

(V1.0 测试版)

上册

UNIDY

前言

欢迎使用本教程！

本教程是一份快速入门教程，旨在让您用最短的时间掌握大部分 C++ 基本语法知识。

祝您学习愉快！

测试版前言

该版本的教程是正式发行前的测试版，因而可能有编写不当的地方，甚至会有各种谬误。

在此，邀请您加入用户体验改善计划。这个计划是自愿的。如果您选择加入该计划，你可以拥有反馈意见和建议的资格。当您对本教程的某一部分的编排有更好的建议，或是认为某一节的讲解难以理解，抑或是发现了内容的错误，您可以及时向我们反馈。这将有助于我们进一步完善这份教程。

本书虽然是快速教程，也就是说您完全可以花一天的时间掌握大部分基本知识，但是，您如果的精力有限，也不必急于这样快速地学完。总之，选择最适合自己的学习方式即可。

最后，再次祝您学习愉快！

C++快速入门（上册）

【概述】

欢迎使用本教程。

本教程的作者为 UNIDY，是一个只会在屏幕上打出随机字符的垃圾程序猿。但是他好歹稍微懂一些 C++ 的基本语法，虽然自己做不出什么成果来，但是还是希望能够让更多朋友走进 C++ 的大门。

C++ 与 C 语言不同，相传不同在两个加号上。其中一个加号，是指 Standard Template Library（标准模板库，简称 STL），提供了大量好用的函数库与模块；另一个加号，是指 Object Oriented Programming（面向对象的程序设计，简称 OOP），体现了 C++ 作为面向对象的中级语言的特性，这一特性将随着你对 C++ 学习的逐步深入有所体会。简而言之，C++ 是对 C 语言的改进。

许多程序员会开玩笑说，自己找不到对象，但是对着这些面向对象的语言，就好像面对着对象一样了。

【基础介绍】

在你进行程序设计前，需要先认识两个工具：**集成开发环境**（Integrated Development Environment，简称 IDE）和**编译器**（Compiler）。

IDE 是你编写代码的环境，称之为集成的，是因为它集合代码编辑器、编译器、调试器和图形用户界面于一体，集成了编写、解析、编译、调试等功能。不同的 IDE 价格不同，功能各异，使用体验也因而有所差异。有的 IDE 只是简单地实现了语法标注、自动缩进等功能，但有些 IDE 内置有良好的**类型推导**、**代码补全**等机制。选择哪一款 IDE，取决于你的应用需求、使用习惯和经费因素¹。

编译器的主要功能是将一种（高级）语言转换为另一种（低级）语言，例如 C++ 编译器能够将你所编写的代码（*.cpp）转化为机器可以识别的二进制码，通常生成一个可执行文件（*.exe），运行这个可执行文件，即可实现代码的机器运行。

编写并运行一个小型 C++ 程序，一般的流程是：写代码→编译→运行→反复调试（除非你能一遍写对）。

【适用读者】

本教程适用于对编程感兴趣，希望通过 C++ 入门或借助算法辅助自己研究问题的非计算机专业的读者。如果你符合以下描述，那么你很适合阅读本教程：

- 希望掌握一门新的程序语言；
- 学校课程中需要用到 C++ 语言；
- 想要利用 C++ 帮助自己完成有关问题的研究；
- 愿意通过学习算法来提升自己程序设计乃至解决问题的思维品质；
- 期待从编程中获得快乐与成就感。

如果你符合以下描述，那么不太建议你阅读本教程：

- 是 IT 专业的学生（该教程的深♂度可能无法达到要求）；
- 有志于开发游戏或手机 APP（出门右转 Java）；
- 急需借助程序完成数学建模项目（出门右转 MATLAB）；
- 打算参加大型计算机编程比赛项目（应当采取更有针对性的训练方式）；
- 学校老师统一讲授 Python 的小学生（怎么现在小学生都会编程了）。

【约定】

¹ 有些 IDE 提供学生版，可以凭学生证获得优惠。有些学校统一为学生批量订购 IDE，学生可免费试用。

该教程仍处于在编状态，甚至随时可能咕咕咕，因而它很有可能只能为你起到入门的作用。

该教程配有相应的测试题和测试工具，你可以利用它们帮助学习，但是题目都很水。

该教程内容多有残缺，想要更详细的内容，参见 <https://www.runoob.com/cplusplus/cpp-tutorial.html>。

该教程推荐使用 Dev-C++ 作为 IDE，不是因为它有多好，而是因为对初学者而言，这款 IDE 正合适。相关的安装包在这份新手教程包中都有配套。

【新手上路】

我相信你还是会安装应用程序的吧……

装好后启动 Dev-C++。

如果发现是英文界面，Tools→Environment Options→Language→简体中文。

然后工具→编辑器选项→代码→缺省源→勾选向项目初始源文件插入代码，并把下面这段代码弄进去：

```
#include<iostream>
using namespace std;
```

```
int main(){
```

```
    return 0;
}
```

这段代码将作为每次新建文件后的默认初始代码，然后点确定。

接下来，点进工具→环境选项，把以下几个选项勾上：

- 在 return 之后暂停控制台程序
- 编译时显示
- 编译完成，自动关闭

以上是设置工作，然后 Ctrl+N，Ctrl+S 之类的相信你是会的。

你好，世界！（Hello, world!）

【题目描述】

Hello, world 往往是学习一门新的语言时第一个完成的程序。

在这里，我们先认识一下 C++ 的输入输出语句。

例如：

```
cin>>a>>b;
```

这一行语句实现了将两个数据通过控制台依次输进变量 *a* 和 *b* 中。

通过这种方式输入的数据，输入时数据之间需要用分隔符隔开，一般为空格或换行。

再例如：

```
cout<<a+b<<endl;
```

这一行语句实现了将 *a+b* 输出到控制台中。

endl 是 cout 语句配套的换行符，<<endl 表示在当前位置输出一个换行符，如不需要可以不要。

当然，在你使用变量 *a* 和 *b* 之前，需要对它们进行声明。这一点我们将会在后面接触到。

输入输出的内容实际上可以对应各种类型，只要这种类型可以转化为字符串。而我们要实现的第一个程序，就是将字符串 “Hello, world!” 输出到屏幕上。

字符串是 C++ 的一个常用的数据类型，我们将会在以后详细学习。在 C++ 中，当你想要直接将一个字符串写进代码时，需要用双引号将它包住。你的 IDE 上应该会对字符串用不同的颜色标出。

先自己猜一猜应该怎么写，然后看标参。

【参考程序 01 - Hello, world!】

```
1 #include<iostream>
2 using namespace std;
3
4 int main(){
5     cout<<"Hello, world!"<<endl;
6     return 0;
7 }
```


【一些说明】

main() 函数是 C++ 的主程序，也就是运行一个 C++ 程序时一开始就进入的部分。

进去以后，执行到第 5 行，在屏幕上打出 Hello, world! 的字符串。

执行到第 6 行，读到 return 语句，这一个函数块（4~7 行）结束，在这时也将意味着整个程序的结束。

【准备编译运行】

在窗口顶部，你会看到这样的按钮：.

第一个是编译按钮，快捷键为 F9，它将先对你的代码进行保存，然后编译，并在同一个目录下生成编译后的可执行文件。

第二个是运行按钮，快捷键为 F10，它不会对你的代码进行编译，而是运行已生成的可执行文件。

当然，我们往往是编译运行一起进行的，这时候会用到第三个按钮，快捷键为 F11，它将以上两步操作合并起来了。

不是还有一个灰灰的按钮的么……？那个按钮非常差劲……而且几乎用不到……

在编译时，如果发现代码中有错误，它会在页面底部提示，并终止编译。有些错误提示比较简单，比如少了个分号¹，但你以后可能会遇到很复杂的情况下的很复杂的错误提示，这就需要你的排错功力了（以后再说）。

我希望你不要一开始就被下面提到的这个卡住，但是如果你真的遇到‘cout’ was not declared in this scope，也就是它连 cin，cout 都不认识了，或许检查一下#include<iostream>和using namespace std;有没有缺了即可。

【查看结果】

我相信在一波操作后，你应该成功地得到了黑框白字的运行结果。

在 Dev-C++ 的 IDE 中，最下面会给出运行时间²和返回值。如果返回值为 0，恭喜，程序至少正常结束了，否则就是程序的非正常退出。

这里的返回值就是 main() 函数中 return 语句的返回值，一般为 0，不建议随意修改。

¹ 在 C++ 中，在大部分情况下（比如 include 语句就是例外），分号是语句的结束标志，换行符则不代表语句的结束。

² 指从开始运行到结束运行所经历的时间，也就是说如果你的程序有从控制台输入的部分，那么输入过程的时间也会计算在内。

这时，你的第一个 C++ 程序已经会写了。下面我们来看第二个。

二数之和（add）

【题目描述】

这道题将会让你更加熟悉 C++ 的简单操作，你需要读取两个整数，并输出它们的和。

【输入格式】

从控制台读入数据。

读入两个整数 a 和 b ，中间用分隔符分开。

【输出格式】

输出到控制台中。

输出结果为一个整数，表示 $a+b$ 的结果。

【输入样例】

2 3

【输出样例】

5

【数据规模与约定】

保证 a ， b ， $a+b$ 均在 `int` 型范围内。

【变量的声明与赋值】

一个变量由它的数据类型和数值构成。

比如在上面提到的 `int` 型就是 C++ 中的整数类型，它的范围是 -2147483648~2147483647。

C++ 中常见的数据类型如下表所示：

<code>bool</code>	真值	0~1（ <code>true</code> 和 <code>false</code> ）
<code>char</code>	字符	0~255 ¹
<code>int</code>	整数	-2147483648~2147483647
<code>float</code>	小数	-3.40e±38 ~ +3.40e±38 6~7 位有效数字
<code>double</code>	小数	-1.79e±308 ~ +1.79e±308 15~16 位有效数字
<code>long long</code>	整数	-9,223,372,036,854,775,808~9,223,372,036,854,775,807
<code>string</code>	字符串	

声明一个变量的方式为：

类型名 变量名[=初值][,变量名[=初值]...]

如：

```
int a=1, b=2, c;
```

¹ 在 C++ 中，字符与 0~255 之间的整数通过 ASCII 码一一对应，两者可以直接隐式转换，因而表格中列出的是其对应的整数值。

一个变量，你既可以在声明时给它赋上初值，也可以在以后需要的地方赋值。但无论如何，在调用这个变量的数值之前，必须要先给它赋予初值。对于未赋初值的变量，它的确也会有默认的初始值，但这个初始值在不同的编译器下有时是不一样的。

赋值语句则更加简单些，C++中的赋值语句是这样的：

变量名 = 数值

如：

a=3;

这就将变量 *a* 赋值为 3 了。

赋值语句本身也有返回值，返回值即它所赋的值。我们看下面一行语句：

a=b=3;

很直观，把 *a* 和 *b* 均赋值为 3。不过事实上可以这样理解：

a=(b=3);

首先，*b* 先被赋予了数值 3，并且赋值语句 *b=3* 有一个返回值 3，这个 3 作为新的值继续赋给 *a*。

当然，在这里，括号完全是可以省去的。

关于赋值语句有返回值这件事，现在看起来可能不太能理解它的意义到底在哪里，但是随着学习的深入，你会渐渐认识到这个特性的价值。

现在，你只需要留下这么个印象：在 C++ 中，一个表达式在进行某种操作时，是可以紧接着给出一个返回值的。这点想法在以后遇到时还会再次提到。

回到这道题，想一想，怎么写，然后看标参。

【参考程序 02 - add】

```
1  #include<iostream>
2  using namespace std;
3
4  int main(){
5      int a,b;
6      cin>>a>>b;
7      cout<<a+b<<endl;
8      return 0;
9  }
```

二数之积 (mul)

【题目描述】

这道题将会让你学习一种从文件输入输出的方法，你需要读取两个整数，并输出它们的积。

【输入格式】

从文件 *mul.in* 中读入数据。

输入的数据共一行，为两个用恰好一个空格隔开的整数 *a* 和 *b*。

【输出格式】

输出到文件 *mul.out* 中。

输出结果为一个整数，表示 *a* 乘 *b* 的结果。

【输入样例】

2 3

【输出样例】

6

【数据规模与约定】

保证 a , b , ab 均在 `int` 型范围内。

【从文件中输入输出】

我们使用 `freopen` 语句。在 `main()` 函数的第一行填上这样两条语句：

```
freopen("mul.in", "r", stdin); freopen("mul.out", "w", stdout);
```

它们分别表示从 `mul.in` 中读入（“read”）和输出（“write”）到 `mul.out` 中。

第一个参数是可以变动的（就是改成需要的文件名），后两个参数一般不需要改动。

其它的代码照常，只是输入和输出的地方变成了外部文件。

当然，两条语句不必然同时出现，如果确实有相应的需求，可以只出现一条（比如想要手动从控制台输入而输出到文件的话，就只要后面一条）。

【使用配套的对拍机】

对拍，本是计算机竞赛用语（顾名思义，~~对子就拍一下~~）。

我们的教程配套有对拍机，当你写好程序，编译生成了相应的可执行文件后，启动对拍机即可自动检验比对答案。

我们这样操作：

- 写好代码，以 `mul.cpp` 命名¹，保存到 **03 - mul** 目录下，编译（不运行）；
- 打开 **03 - mul** 文件夹，找到 `mul.bat`，双击执行；
- 查看自己的测评结果。

这一题一共有 10 个测试点，也就是说，如果你能顺利地连敲 10 次，每一次都提示“找不到差异”这样的句子的话，恭喜你，本题过关！如果有错，结合文件夹中给出的测试数据（`*.in`），（`*.ans`）进行调试。

需要注意的是，我们的比对方式是**全文比对**，我们必须约定：所有程序输出的文件都必须以有且仅有一个换行符结尾，且每行末尾不要有多余的空格，否则可能会出现内容其实全部一样但是文件不一样的情况。

这道题，我们就不提供参程了，我相信你是可以完成的。注意哦，要自己手打一遍哦，不要直接复制上一题的代码改一改哦~

计价器 (taxi)

【题目描述】

在出租车的计费标准中，一般有起步价和每公里价格两部分。譬如某市出租车 10 元 3 公里，以后每增加 1 公里价格增加 3 元。

有一些客户希望咨询一段路程对应多少价格。现在希望你能够写代码实现这一功能。

【输入格式】

¹ 文件名应当与试题名称相同。注意试题文件夹前的序号（如本题中的 03）不要放入文件名。

从文件 *taxi.in* 中读入数据。

输入的第一行为一个正整数 n ，表示共有 n 次咨询。

输入的第二行为 n 个正整数，依次表示 n 次咨询的里程数。

同一行输入的相邻两个元素之间，用恰好一个空格隔开。

【输出格式】

输出到文件 *taxi.out* 中。

输出结果共有 n 行，依次表示 n 次咨询对应的价格。

【输入样例】

```
3
2 4 5
```

【输出样例】

```
10
13
16
```

【数据规模与约定】

保证所有数据均不超过 10。

【陷入沉思】

分段计价……看来需要选择语句……

n 次咨询……看来需要循环语句……

【一些运算符】

运算符有好多种，在这里我们先介绍关系运算符和逻辑运算符。

关系运算符：比较两边数值的大小关系，有 $=$ 、 $!=$ 、 $<$ 、 $>$ 、 $<=$ 、 $>=$ 。

特别注意的是，C++ 的相等符号是双等号（与赋值的单等号区分），不等符号是感叹号加上一个等号。

逻辑运算符：或（ $||$ ）、且（ $\&\&$ ）、非（ $!$ ）。

逻辑运算符之间有优先级关系！ $>\&\&>||$ （not and or 谐音 not at all）（但是记不清的话只管给你的表达式加括号来表示优先算括号内的部分好了）。

举点例子吧：

```
x>1 && x<=5
```

```
a<b || c!=d
```

```
x==y || (x!=y && p==q)
```

```
!(m<n)
```

（我希望这些式子是一目了然的例子……）

其中，最后两个的括号都是可以去掉的（但是加上去也无所谓哈）。

【短路原则】

当我们看到这样一个表达式的时候：

```
4>5 && 6<7
```

我们发现 $4>5$ 为假，因此根本不用去管 $6<7$ 的真假，即可断定上述表达式为假。

同理，当我们看到这样一个表达式的时候：

`2<3 || 9<8`

我们发现 `2<3` 为真，因此根本不用去管 `9<8` 的真假，即可断定上述表达式为真。

人如是，机器亦如此。

对于 C++ 编译器，有短路原则：

- 对于表达式 `p && q`，当计算出 `p` 为假时，不再对 `q` 求值，直接得出表达式的结果为假；
- 对于表达式 `p || q`，当计算出 `p` 为真时，不再对 `q` 求值，直接得出表达式的结果为真。

这对于程序而言，是一个小小的优化。但对于我们开发者而言，这一点也可能转化为我们的一些奇技淫巧。以后见到再提。

【if 语句】

C++ 中 if 语句的结构如下：

```
if(条件){  
    分支 1  
}  
else{  
    分支 2  
}
```

当条件为真，执行分支 1；否则，执行分支 2。当某一分支只有一句表达式时，对应的花括号可省略。

当不需要 else 分支时，可简化为：

```
if(条件){  
    分支  
}
```

此时，若条件为假，则不执行该语句块。

显然，在每个分支中，都可以进一步嵌套另一层 if 语句，例如：

```
if(条件 1){  
    分支 1  
    if(条件 2){  
        分支 1-1  
    }else{  
        分支 1-2  
    }  
}  
else{  
    分支 2  
}
```

当然，我们还可以很灵活地利用 else if：

```
if(条件 1){  
    分支 1  
}  
else if(条件 2){  
    分支 2  
}  
else{  
    分支 3  
}
```

虽然这里只列出了一个 else if，但可以想见，else if 的分支数量要多少就能有多少。

在本问题中，价格是关于路程的分段函数，应使用选择结构来实现，相信现在你已经能够写出代码了。

但别急……

【?:语句】

?:语句的结构如下:

表达式 1 ? 表达式 2 : 表达式 3

过于抽象……

举个例子:

```
x<=3 ? y=10 : y=10+3*(x-3);
```

相信你已经懂了……

但我告诉你这条语句还可以简化:

```
y=(x<=3 ? 10 : 10+3*(x-3));
```

其中,最外层的括号可以省略。

也就是说,?:语句有点像 if 语句的简化,但又不太一样。主要的区别如下:

- if 语句的分支可以很长,但?:语句的每个分支只能有一个语句;
- if 语句可以省略 else 分支,但?:语句的各个表达式均不可省略;
- if 语句本身作为语句块没有返回值,但?:语句既可以独立成为一条指令,也可以成为有返回值的一句

表达式(当表达式 1 为真时返回值为表达式 2,否则为表达式 3)。

现在你已经可以十分简洁地写出计价器的表达式了,但是循环 n 次呢……

【while 循环语句¹】

while 语句的结构如下:

```
while(条件){  
    循环体  
}
```

当程序执行到一个 while 语句块时,首先检测条件,若为真,则执行循环体,随后回到条件处重新判断,若为真,则继续上述操作,直到某一次判断结果为假,方退出循环;若一开始就为假,则不执行该语句块。

想一想,我们要让一个循环体重复执行给定的 n 次,条件处应该怎么写呢?还要有什么操作吗?

也许,可以利用这个 n ,每执行一次减去 1,直到 n 减到 0,循环体也就执行 n 次了。大概像这样:

```
while(n>0){  
    执行部分  
    n=n-1;  
}
```

但是当我们了解了更多知识时,会发现上述语句的笔墨可以再减省一点。

【隐式转换】

C++对一些数据类型之间支持**隐式转换**,即当向一个表达式中传入的值的类型与期望的不同时,编译器首先会自动尝试将现有的值的类型向目标类型转化,这一操作不需要人的干预,不能为人所直接看见,因而称其为隐式的。

C++对隐式转换有详细的规范,但我们大不必记住这些条条框框,在实例中接触并掌握即可。

在这里,考虑 int 和 bool 型的隐式转换:

• $(\text{int}) x = \begin{cases} 1, & x = \text{true} \\ 0, & x = \text{false} \end{cases}$ (bool \rightarrow int);

¹ 有关 while 循环的部分希望读者当成重难点学习。while 循环本身的内容并不难,但是编者从中引申出了一些重难点知识,这些是比较关键的。

- $(\text{bool}) x = \begin{cases} \text{true}, & x \neq 0 \\ \text{false}, & x = 0 \end{cases} \text{ (int} \rightarrow \text{bool)}.$

这意味着，在 if 和 while 等语句的条件中，当你传入一个整数时，编译器不会由于你没有传入一个 bool 型报错，而是会将其隐式转换为 bool 型后进行判断。

现在看来，隐式转换似乎非常可爱。但是以后你会接触到一些情况，隐式转换“自作主张”，反倒会出乱子的。

于是我们有了改进版：

```
while(n){
    执行部分
    n=n-1;
}
```

这里，当且仅当 n 为 0 时跳出循环，与之前的写法可以认为是等价的（你如果把 n 初值搞成负数嘛……）。但还是太冗长！

【自增/减运算符】

$n--$ 、 $--n$ 和 $n=n-1$ 的功能近似相同。

$i++$ 、 $++i$ 和 $i=i+1$ 的功能近似相同。

$++$ 和 $--$ 是自增运算符和自减运算符，它们的基本功能是在原变量的基础上 $+1$ 或 -1 。

注意到，正如之前所提到的，任何一个表达式在进行完某种操作后会给出一个返回值。

自增运算符的返回值规则如下：

- 表达式 $x++$ 的返回值为 x ；
- 表达式 $++x$ 的返回值为 $x+1$ 。

自减运算符的返回值规则同理。

例如：

```
x=y=10;
a=(x++);
b=(--y);
```

此时 x 、 y 、 a 、 b 的值分别为 11、9、10、9。其中的括号可省略。

给一个形象的记忆方式： $x++$ 就是先返回 x 再给它 $++$ ，而 $++x$ 则是先给它 $++$ 再返回 x 。

当然，如果你愿意看下面这一节的话，可能会获得更本质的理解……

【左边还是右边？¹】

$x++$ 实现的过程本质大概是这样：

- 将 x 取出，并复制生成一个副本；
- 将 x 的数值 $+1$ ；
- 将副本作为返回值返回。

$++x$ 实现的过程本质大概是这样：

- 将 x 取出，并给它 $+1$ ；
- 将此时的 x 作为返回值返回。

另外，这样看起来，当不需要考虑拿出什么样的返回值的时候， $++x$ 的效率貌似还比 $x++$ 高一些（×

- 当 x 为基本类型时，由于编译器优化结合底层的处理过程，两者效率无差异；
- 当 x 为自定义类型时，由于避免生成副本， $++x$ 的效率貌似真比 $x++$ 高一点点。（超纲了！超纲了！）

¹ 这是选读内容，第一遍学习这一节，如果看不进去，就跳过吧……

【试试看】

现在，把 while 循环体中的 $n=n-1$ 改成一个自减运算，再用它替换掉判断条件，用其返回值做判断。想一想，条件里面，是写 $n--$ 还是 $--n$ 呢？

我们的判断用的是这个表达式的返回值，就相当于是想，是要先减再判断还是先判断再减呢……

~~（不妨就取 n 为 1 想一想……）~~

想清楚后，把计价器这一题写成代码，然后用对拍机测评一下。

福利：从这一题开始，每一题的文件夹下面，都配备了这一题对应的模板文件，供你轻松上手。

“为什么之前的题目不给？”“锻炼你的基本功的啊……”

【解后思考】

- 当路程大于 3 的时候，你的表达式写的是 $10+3*(x-3)$ 还是 $3*x+1$ ？

显然后者比前者多一点人工化简，而对于机器而言，后者的运算也会少那么一点点。

运行一次这点差异可能不算什么，运行 10 次当然也看不出什么，可是当 n 非常非常大的时候呢……从中你能获得什么启发？

- 经过一波操作，我们把一段 while 循环语句已经写得非常简洁了。可是是不是少了点什么？

我如果在运行过程中想要输出我这是第几次执行，是不是搞不定了？

此外，在这个过程中， n 是随时变化的，如果我想要获取原始的 n 的值，是不是歇菜了？

解决办法总是有的，比如可以引入循环变量 i 。但试试看就会发现这会让代码又变得十分冗长……

【for 循环语句】

for 语句的结构如下：

```
for(初始指令; 执行条件; 增量指令){
    循环体
}
```

for 循环是这样执行的：

- 首先，执行且仅执行一次初始指令，在这里你可以进行声明新变量（一般为循环变量）、赋值等操作（当你不需要任何初始指令时，这一栏可以不写，仅保留分号即可）；

- 接下来，判断执行条件，若为真，则进入循环体，否则不执行循环体，直接调到该语句块的后面（若这一栏不写，则默认为真，即程序不进行任何判断）；

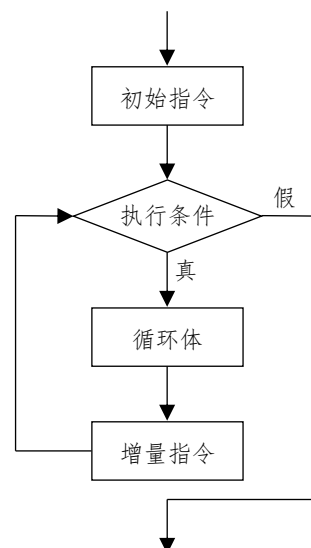
- 进入循环体后，执行循环体（如果循环体只有一条语句，可省略花括号；如果循环体不执行任何语句，空白的花括号 `{ }` 可由一个分号；取代）；

- 执行完循环体后，跳回到一开始的增量指令并执行之，这一指令一般用于对循环变量进行自增操作，当然你也可以用它来做其它事情（当然你也可以这一栏不写，仅保留条件后面的分号即可）；

- 此时，再次判断条件，若条件为真，则再次执行循环体，否则跳出循环。

不如看右边的流程图……

我们再来看下面这个例子¹：



¹ 由于编者偏爱 `++i` 而非 `i++` 的写法，在 for 循环的例子中，尽管两者无区别，但你将大多看见形如前者的表达式。

```
for(int i=1; i<=n; ++i){  
    cout<<i<<endl;  
}
```

在这个例子中，循环体就被循环执行了 n 次。

了解了 for 循环之后，你可以将计价器的题目从 while 循环改成 for 循环（~~应该只要改一行字吧……~~），然后运行测评一下，并初步体会一下两者的区别。

【for 比 while 多了什么】

for 循环与 while 循环相比，多出了初始指令和增量指令，且在 for 循环括号中的三个表达式往往与循环变量息息相关。这意味着，利用 for 循环，我们可以更方便地管理循环变量。

当然，像上述例子中 for 循环括号内的三个表达式只是大多数情况下的样例，在实际使用时可以更加灵活。

【笑话】

（一个分号引起的悲剧）一个男生追求一个女生。

男：

```
for(;;) cout<<"I love you!"<<endl;
```

女：

```
for(;;); cout<<"I love you, too!"<<endl;
```

【变量的作用域】

一个变量，除非它在所有的代码块的外部被声明，否则它只负责包含它的最内层的代码块。在所有的代码块的外部（一般是程序的头部）被声明的变量称为**全局变量**，在某个代码块的内部被声明的变量称为**局部变量**。如：

```
1  #include<iostream>  
2  using namespace std;  
3  
4  int x=5;  
5  
6  int main(){  
7      int s=0,n=100;  
8      for(int i=1; i<=n; ++i)  
9          s=s+i;  
10     cout<<s<<endl;  
11     return 0;  
12 }
```

在这个例子中， x 是全局变量， s 、 n 、 i 均为局部变量。其中， s 和 n 为 `main()` 的局部变量， i 为 for 循环的局部变量。

这意味着，如果你在第 10 行的位置试图输出 i ，编译器是会报错的，因为变量 i 不负责 for 循环之外的部分。

当一个局部变量与某个处在该代码块之外的变量重名时，内层变量会覆盖外层变量。如：

```
1  #include<iostream>  
2  using namespace std;  
3
```

```
4  int x=5;
5
6  int main(){
7      int x=10;
8      cout<<x<<endl;
9      return 0;
10 }
```

这里，输出 x 的值应当为 10。

这一覆盖机制是完全符合 C++ 标准和规范的，但是，当你写出这样的程序时，需要格外小心，分清楚自己写的变量到底是哪里的变量。

【思考题】

这道题主要考察你对 for 循环的执行逻辑的理解程度。

```
1  #include<iostream>
2  using namespace std;
3
4  int main(){
5      int i,n=10,s=0;
6      for(i=1; i<=n; ++i)
7          s=s+i;
8      cout<<i<<" "<<s<<endl;
9      return 0;
10 }
```

这里输出的 i 和 s 的值应当分别是多少呢？先独立思考，再写程序验证一下你的想法。如有需要，再回头看看 for 循环的流程图。

你理解清楚了这一点后，可别得意，for 循环的挑战还在后面等着你呢……

素数判断 (isprime)

【题目描述】

我们称一个大于 1 的整数为素数 (prime)，如果除了 1 和它自身外，它不再有其它的因数。请你写一个程序，判断一个大于 1 的整数是不是素数。

【输入格式】

从文件 *isprime.in* 中读入数据。

输入数据只有一个正整数 n ，表示待判断的整数。

【输出格式】

输出到文件 *isprime.out* 中。

输出结果为一个字符串，当输入的 n 为素数时输出 YES，否则输出 NO。

【输入样例 1】

7

【输出样例 1】

YES

【输入样例 2】

20

【输出样例 2】

NO

【数据规模与约定】保证 n 大于 1 且 n 不超过 10000。**【思考算法】**

算法的思路还是比较容易获取的，就是用 for 循环对大于 1 且小于 n 的所有整数逐个遍历。一旦有可以整除 n 的数，就判断 n 不是素数；如果遍历完成发现均不能整除 n ，则判断 n 为素数。

不过仔细想想如何实现，会发现需要解决一些问题：

- 循环的初始指令应当写什么？
- 循环的执行条件应当如何写？
- 整除关系应当如何表达？
- 当整个 for 循环结束时，如果都没有发现整除关系，我们当然可以放心大胆地说 n 为素数，可一旦发现能够整除 n 的数，我们还需要继续遍历下去吗？那又应该如何处理呢？
-

为了解决这些问题，我们还需要更多的知识。

【取余运算符%】

C++中，整数 a 和整数 b 的取余运算规则如下：

$$a \% b = a - \text{fix}\left(\frac{a}{b}\right) \cdot b$$

其中， $\text{fix}(x)$ 表示绝对值不大于 $|x|$ 的距离 x 最近的整数。

过于抽象……换种说法……

其实，这里主要是正负号的迷惑行为。当 a 和 b 均为正整数时， $a \% b$ 就是我们熟悉的带余除法的余数。关键是当 a 和 b 出现负整数时……

我们按照如下规则进行操作：

- $a \% b$ 的绝对值与 $|a| \% |b|$ 相等；
- $a \% b$ 的正负号与 a 的保持一致。

例如 $5 \% 3 = 5 \% (-3) = 2$ ， $(-5) \% 3 = (-5) \% (-3) = -2$ 。

假如你对初等数论有一定的了解的话，应当可以明白为什么 C++ 中的 % 严格意义上不应称为“取模运算符”。事实上，在另外一些语言（例如 Python）中，% 的含义会更接近数论意义上的取模运算，例如在那里， $-5 \% 3$ 的值为 1。如果你将来会迁移到这些语言，或是正从这些语言转入 C++，你可以关注一下这点差异。

特别地，当 b 整除 a 时， $a \% b = 0$ 。

【break 语句和 continue 语句】

在循环语句的循环体中或是在多路分支语句¹的一个分支中，break 语句可以强行终止并跳出该代码块（如果有多层嵌套，仅跳出最内层的代码块）。

例如，在这个例题中，一旦发现循环变量 i 可以整除 n ，就可以使用 break 语句跳出循环：

```
for(...){
    if(n%i==0){
        break;
    }
}
```

停！总不能就这样什么都不做就跳出去了，至少得要留下 NO 的判断结果啊……

那好，这样行了吧：

```
for(...){
    if(n%i==0){
        cout<<"NO"<<endl;
        break;
    }
}
```

```
cout<<"YES"<<endl;
```

呃……那输出完 NO 的时候跳出来岂不是还得把 YES 也给顺带输出了……

这样还不行。但办法总是有的。

事实上，有不止一种思路可以解决这个问题，读者可以先思考一下。下面介绍的这种操作，可能形式上不是解决这个问题的最精巧的方法，但这种手法具有一定的普适性。

我们先声明一个 bool 型的旗帜变量 flag，并初始化为 true；一旦发现整除关系，就把 flag 标记为 false：

```
bool flag = true;
for(...){
    if(n%i==0){
        flag=false;
        break;
    }
}
```

然后根据 flag 的结果输出即可。

简单补充一下 continue 语句。与 break 语句不同，continue 语句的作用则是强行跳过循环体剩下的未执行的部分，但并不跳出循环。此后，对于 for 循环，continue 语句将引导循环直接执行增量指令和条件判断，若满足条件则继续循环；对于 while 循环和 do...while 循环²则将引导循环直接执行条件判断部分。

【最后一点】

循环变量 i 的循环范围是什么？容易想到， i 的初始值应当设定为 2。而上界呢？真的要检验到 $n-1$ 吗？

经过简单的数学推理，可以发现， i 只需要跑到 \sqrt{n} 即可。也就是说： $i \in [2, \sqrt{n}]$ 。³

可是…… $\leq \sqrt{n}$ 又怎么处理呢？思考……（先不急着翻页）

¹ 即 switch 语句，限于编排原因不作介绍，如有兴趣可上网查阅相关资料。switch 语句对特定情况下的选择条件语句进行了优化，该用的时候用处自然是有的，但编者个人认为它的语法结构有些奇诡。

² 先执行后判断的 while 循环，限于编排原因不作介绍，如有兴趣可上网查阅相关资料。

³ 把 $n-1$ 改进到 \sqrt{n} ，看似不起眼的一点改进，可是当 n 充分大的时候，就可以看出两者明显的差异了。通过这点改进，可以在一定程度上起到提高程序运行效率的作用。

注意到当 i 和 n 为正整数时, 有 $i \leq \sqrt{n} \Leftrightarrow i * i \leq n$ 。问题解决! 一段完整的代码已经呼之欲出了! 赶紧写好测评一下吧!

哥德巴赫猜想 (goldbach)

【题目描述】

相传, 哥德巴赫曾经大胆猜想: 任一大于 2 的偶数都可写成两个素数之和。
虽然人类至今未能破解这一难题, 但我们可以对数据比较小的情况进行探究。
现给你一个大于 2 的偶数 n , 请你求出所有的有序素数对 (p, q) ($p \leq q$) 满足 $n=p+q$ 。

【输入格式】

从文件 *goldbach.in* 中读入数据。
输入数据只有一个整数 n , 表示待分解的偶数。

【输出格式】

输出到文件 *goldbach.out* 中。
输出结果共有 k 行, 按照 p 从小到大的顺序依次输出求得的结果, 其中 k 为满足条件的素数对的数目。
输出结果的每一行由两个整数 p 和 q 组成, p 和 q 之间用恰好一个空格隔开。

【输入样例】

30

【输出样例】

7 23
11 19
13 17

【数据规模与约定】

保证 n 为大于 2 的偶数且 n 不超过 10000。

【基本算法】

考虑到 $p \leq q$, 我们让循环变量 p 从 2 遍历到 $n/2$ (在 C++ 中, 整数除以整数的结果为取整后的结果, 具体而言是向 0 取整后的结果), 相应的 q 每次跟着取 $n-p$ 。判断若 p 和 $n-p$ 均为素数, 则将其输出。

但是实际操作时应当会遇到一个问题: 本题需要判断两次素数, 难道要写两遍素数判断?

【初识函数】

C++ 允许用户自定义函数, 例如本题中即可自定义一个判断素数的函数。

C++ 函数定义的一般格式如下:

```
返回类型 函数名([参数类型 参数名称[, 参数类型 参数名称...]]){  
    函数体  
}
```

其中, 括号内的部分是参数列表, 用于接收传入该函数的参数。当函数不需要传入任何参数时, 参数列表可省略 (比如你所看到的 `main` 函数)。

用素数判断的具体的例子辅助理解。

```
1  #include<iostream>
2  using namespace std;
3
4  bool prime(int n){
5      for(int i=2; i*i<=n; ++i)
6          if(n%i==0)
7              return false;
8      return true;
9  }
10
11 int main(){
12     int x;
13     cin>>x;
14     cout<<(prime(x)?"YES":"NO")<<endl;
15     return 0;
16 }
```

在这个例子中，第 4~9 行是自定义函数的部分，第 11~16 行是主程序。~~（你当然应该知道 C++ 程序运行时只会从主程序开始运行，函数只有在调用的地方才会执行。）~~

我们重点关注第 4~9 行。

第 4 行是函数类型和参数被声明的地方。

- bool 指函数的返回值的类型，也就是说该函数的返回值是 bool 型数据（是或不是素数）；
- prime 指函数的名称，在调用该函数时使用这个函数名；
- int n 是参数列表，该函数只有一项参数；
- int 指参数的类型，即传入的 n 是 int 型数据；
- n 是传入形式参数的名称，在函数体中调用传入的参数时统统用在这里声明的变量名。

第 5~6 行是熟悉的语句，不再赘述。

第 7 行和第 8 行是 return 语句。当程序执行到 return 语句时，将 return 语句后面的表达式运算的结果（在这个例子中直接是 true 和 false 两个值）作为函数的返回值返回，同时将会跳出函数。

例如，当传入的 n 不是素数时，程序会在某一个时刻执行到第 6 行判断为真，进而执行第 7 行，是个 return 语句。此时，程序会将 false 作为 prime 函数的返回值，并跳出函数体。此时 prime(n)的结果即为 false。

而当传入的 n 是素数时，程序永远没有机会执行第 7 行，而是在循环结束时执行第 8 行。这时，prime 函数的返回值就是 true，也即 prime(n)的结果为 true。

这样，一个完整的素数判断函数就写完了。

下面再来看主程序部分。主要是第 14 行，即函数调用处。

在这个例子中，prime(x)调用 prime 函数。当程序执行至此，希望计算 prime(x)的值时，它将 x 的值作为参数传入 prime 函数，并返回函数的运行结果。

注意，此处传入的表达式（如本例中的 x ）既不必与函数声明处的变量名（如本例中的 n ）保持一致，也不必拘泥于非得只能传入某一个变量。它可以是任意的表达式，而对于 C++ 程序则会将这个表达式先计算好，然后将结果以一个统一的形式变量（如本例中的 n ）的形式传给函数进行进一步运算。

当然，参数列表中传入的参数可以不止一个。需要传入多个参数时，每个形式参数声明之间用逗号隔开。例如请你试着实现函数 maximum，满足：传入两个 int 型参数，返回值也为 int 型，它需要对传入的两个参数进行大小比较，并且返回较大的那个。

还是，先动手写一写，然后再翻页看解答。~~（这个不是例题，所以没提供模板，自己写自己调试好了……）~~

【自己实现的最大值函数】

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

比较一下，看看自己写的怎么样，相信你写的应该和这个差不多的吧（

你要是想出了更简单的写法当然更好咯（

不过，C++标准库中，已经自带了许多有用的函数，比如较大值和较小值函数。也就是说，当你需要使用这些常用的基本函数时，可以直接用，不必自己再写了¹。

【先认识一些常用函数】

C++中自带²的较大值函数为 `max(a,b)`，较小值函数为 `min(a,b)`，绝对值函数 `abs(x)`，可以自己调用一下试试看³。

当你包含（include）了更多头文件后，你可以使用更多函数。

- 数学库（`cmath`）

在 `#include<iostream>` 下一行添加 `#include<cmath>`，你还可以使用正弦函数 `sin(x)`，余弦函数 `cos(x)`，正切函数 `tan(x)`，自然对数函数 `log(x)`，常用对数函数 `log10(x)`，指数函数（ a^b ）`pow(a,b)`，开平方函数 `sqrt(x)`⁴等。

- 算法库（`algorithm`）

继续添加 `#include<algorithm>`，你还可以使用最大公约数函数 `__gcd(a,b)`⁵等（以后会介绍更多）。

在 Dev-C++ 的 IDE 中，初次编译后⁶，你可以按住 Ctrl 键，再用鼠标点击你调用处的函数名，可以跳转到这个函数的定义⁷。

【但是有的函数不需要返回值怎么办……】

是的，有的函数不需要返回值，而只是针对传入的参数（有时甚至都不一定有）进行一定的操作。

此时，统一用 `void` 作为这种函数的返回值。

¹ C++标准库自带的函数大多为泛型函数，它使得一个函数可以适用于更广泛的情境。你目前不需要理解它具体是什么意思，但要明白，在一些特定的情况下，你自己写的函数存在着比 C++自带函数更高效的可能性。如果遇到这种情况，建议你选择使用自己写的函数。

（下面的内容敲黑板……）不过，C++标准库是一整个开发团队共同完成的，因而在多数情况下，请不要对自己写的函数过于自信。我们会将别人写好的工具、模板或者库称为轮子，而将自己尝试写这些工具的行为称为造轮子。当你试图造的轮子早已有人实现过（而你又无法比他们实现得更好）时，这种行为就会被称为重复发明轮子。诚然，自己造轮子对自己的编程实力的提升有一定的帮助，但业界有云：有轮子为什么要自己造，自己造的又不一定圆。因此，当你使用轮子只是为了辅助自己完成某个项目中的小功能时，如果你不能造出更好、更适合（毕竟别人的轮子有的不适合自己的）的轮子，也不需要通过了解轮子的内部原理来提升自己的认知，还是就用别人的轮子吧。

如何比较自己和别人轮子的效率（这里特指时间效率）呢？很简单，对于同样的功能，把自己的轮子和别人的轮子分别用 for 循环做个一百万遍，小量放大嘛（

² 严格意义上，C++不自带任何函数，只是这些函数在包含 `iostream` 头文件时已经出现了，所以在此称之为自带的。

³ C++对于同名函数的处理有重载、隐藏、覆盖三种机制。（假如这份教程我真的坚持编完的话，读者都会学习到这三种机制。）由于尚未到达适合介绍这些机制的时候，你可以先自己探索一下定义了好多好多个同名函数会怎么样，在什么情况下会“打架”。

⁴ 除非你需要用到 \sqrt{x} 的精确值，`sqrt(x)` 函数是不建议使用的，因为它使用的算法导致计算该函数效率较低（你自己手开平方的精确值也很令人头秃的嘛……）。举个例子，比如在素数判断函数中，如果你要使用 `i<=sqrt(n)`，会比 `i*i<=n` 的效率要低。

⁵ 这是一个“藏得很深”的函数，你看它前面还故意加了两个下划线（__）来隐藏自己呢（滑稽

⁶ 因为 Dev-C++ 做得比较菜，你一般需要先编译一下，它才会准备处理解析你所调用的函数。在更优秀的 IDE 中没有这种情况。

⁷ 限于目前掌握的知识，C++自带函数的定义你可能会看得云里雾里。不过不要着急，等学了泛型函数后你就能看懂了。

以计价器为例题，举个不恰当的例子（因为在真正的使用中不会有这么蠢的情境的……）。

```
1  #include<iostream>
2  using namespace std;
3
4  void solve(int x){
5      cout<<(x<=3?10:3*x+1)<<endl;
6  }
7
8  int main(){
9      int n,x;
10     cin>>n;
11     while(n--){
12         cin>>x;
13         solve(x);
14     }
15     return 0;
16 }
```

这里，solve 函数不需要任何返回值，因为它只是在对传入的 x 进行计算并输出的操作，在形式上用 void 作为返回值类型。如果你试图获取 solve(x) 的值，程序会报错。

计算机程序中定义函数的一大目的在于增强代码的可读性，并在一定程度上将重复的代码块（如哥德巴赫猜想这一题）减省掉，因而不一定需要有返回值。这不同于数学中的函数概念。

【有返回值的函数，还可以特意不要它的返回值】

在 C++ 中，单独的表达式本身也可以自成一个语句。比如你在代码中加入这样一行：

```
1+1;
```

程序会对 1+1 计算一下，然后就像什么都没发生一样。

一个函数的调用何尝不是一个表达式呢？可以想见，无论它有没有返回值，我们都可以使它单独成为一条语句（而不接收其返回值，如有）啊。只是执行到这条语句时会产生一些效果而已。

如果你还记得之前提到过的想法“一个表达式在进行某种操作时，是可以紧接着给出一个返回值的”，我们现在结合起来看，应该可以给出一个目前看来更加完备的想法：

一个表达式既可以表示某种运算，也可以表示某种操作，甚至可以两者兼备；但无论何种目的，这个表达式都可以选择是否给出返回值；而即使表达式能够给出返回值，我们也可以选择是否使用这个返回值。

这仅仅是编者的一点小想法，拿不上台面的，但也希望这点想法可以帮助你更好地理解 C++ 的语法特性。

【回到原题】

在刚刚的介绍中，我们认识了 C++ 的自定义函数，了解了自定义函数的语法格式，学习了如何定义和调用函数，还熟悉了一些常用的 C++ 函数。在函数的参数及数据类型方面，我们学习了如何声明参数列表以及如何理解函数的返回值。然而，正如刚刚认识函数的节标题“初识函数”一样，受限于目前的认识程度，我们现在所了解的，只是函数的一点基本功能。随着我们学习的深入，以及对一些功能需求的增加，我们会逐步掌握更多更深入的知识。

现在我们回到原题，相信一切已经豁然开朗了。

我们在“基本算法”中已将解题思路分析清楚，唯一的困惑就是两次判断素数如何处理。现在我们知道，通过自定义素数判断函数即可实现。而 prime 函数在上面的分析中也已给出，万事俱备，只等你的发挥！

【进一步思考】

- 不出意外，if 语句会有两大类写法：

```
if(prime(p) && prime(n-p))  
    cout<<p<<" "<<n-p<<endl;
```

和

```
if(prime(p))  
    if(prime(n-p))  
        cout<<p<<" "<<n-p<<endl;
```

两者的主要区别在于一个用了&&关系运算符，一个用了嵌套的if语句。

有人说，选择第二种好，因为第一步判断已经将 p 不是素数的情况筛选在外了，免去了第二步判断的麻烦，效率更高。请问这种说法合理吗？（提示：想一想之前提到的短路原则。）

- 这样的方法已经最简了吗？还有优化的余地吗？

目前看来，由于这一题所给数据比较弱，恐怕没有什么更有效的优化方案了。但是不妨设想：如果这题像计价器那样有多次查询呢？每次查询都要把 2、3、4、5……是否为素数重新判断一遍，是不是有点浪费了呢？

面对这种情形，有大致两条处理路线：

- 每进行一次素数判断，就把判断的结果保存到相应的位置，这样下一次再对相同的素数判断时，就把“记忆”中的结果拿出来，不必再进行重复的枚举和判断；
- 在正式运行前进行“预处理”，对可能需要判断的所有数（例如 $2 \sim n$ ）进行判断，将结果保存到一张表中，在正式运行时查表即可。

两条处理思路都很常用，如同鱼和熊掌，各有优劣。在这里，我们选择第二条路进行讲解。

素数表 (primetable)

【题目描述】

请输出从 2 到 n 的所有素数。

【输入格式】

从文件 *primetable.in* 中读入数据。

输入数据只有一个整数 n ，含义见题目描述。

【输出格式】

输出到文件 *primetable.out* 中。

输出结果共有 k 行，每一行只有一个整数，按照从小到大的顺序依次输出求得的结果，其中 k 为 2 到 n 中所有素数的总数。

【输入样例】

10

【输出样例】

2

3

5
7

【数据规模与约定】

本题只有 1 个测试点，在这个测试点中， n 的值为 1000000。

【需要新的数据结构——数组】

一张素数表应该如何存？

我们需要这样一种数据结构：它可以存放一系列连续的数据，并且可以轻松地读写指定位置的数据。这种数据结构就是数组。

数组的声明如下：

数据类型 数组名称[数组大小]；

数组大小是指创建这个数组最大可能容纳的元素个数，程序会向系统请求相应的内存，因此一旦声明后这个大小不可改变。

例如，我要声明一个由 int 型数据组成的长度为 10 的数组 a ，就应该写：

```
int a[10];
```

数组可以在声明时初始化，初始化的方式是在后面直接赋上用花括号括起来的元素列表。例如：

```
int a[10]={1,2,3,4,5};
```

这样，程序就将 1、2、3、4、5 依次赋给数组 a 的前 5 项了。此时，数组 a 的后 5 项默认初始化为 0。通过这种方式初始化的，花括号内的元素个数不得超过数组的大小。

特别地，当我们用一个空的花括号进行初始化时，可以实现将数组所有的元素初始化为 0。例如：

```
int a[10]={};
```

此时，数组 a 的所有元素均为 0。

【从 0 号位置开始计数】

重中之重：在程序员的眼中，所有的序号应该从 0 开始记！

例如在这个例子中：

```
int a[10]={1,2,3,4,5};
```

下标的取值范围为 0~9（而不是 1~10），且数组初始化的 5 个值从 0 号位置开始赋值，因而数组数据与下标的对应关系如下表所示：

下标	0	1	2	3	4	5	6	7	8	9
数据	1	2	3	4	5	0	0	0	0	0

这点错乱可能会让你感觉很不爽，那你可以这样操作：

```
int a[10]={0,1,2,3,4,5};
```

这样，数据与下标的对应关系就是你所熟悉的了：

下标	0	1	2	3	4	5	6	7	8	9
数据	0	1	2	3	4	5	0	0	0	0

但是要注意咯，你虽然声明的时候将大小设定为 10，但由于数组的下标从 0 开始记，因此没有下标 10。

面对这种情形，我们常用的操作是，在声明时适当把数组大小放宽一点，最好比预定的最大容量要多出 5~10 个元素，像这样：

```
int a[15]={0,1,2,3,4,5};
```

这就可以在一定程度上降低数组越界造成的风险。（不过在下面的例子中我还是用 $a[10]$ 进行介绍。）

从 0 开始计数，这一点使得 i 号元素在实际存储中处于第 $i+1$ 个位置。当然，我们只需要在脑海中留下这点印象，实际操作时，我们一般只谈数组的下标，并且记住有 0 号位置这么一回事就可以了。

【数组数据的读写】

继续用刚刚的例子：

```
int a[10]={0,1,2,3,4,5};
```

当我们需要读取 i 号（这个号码当然是从 0 开始记的）元素时，我们使用表达式 `a[i]`。例如

```
cout<<a[3]<<endl;
```

的结果为 3。当然，你也可以通过 `cin` 语句输入对应的元素的值。

现在，你想将这个元素的值修改为 6，你的操作是：

```
a[3]=6;
```

此刻，数组内的元素如下表所示：

	0	1	2	3	4	5	6	7	8	9
a	0	1	2	6	4	5	0	0	0	0

这样就实现了数组元素的访问和修改了。

【数组元素的批量操作】

我们在这里先介绍四类操作：清空、批量赋值、计数、求和。

• 清空

我们已经学会了在声明数组时就将数组清空，但当一个数组已经使用过，我们需要再次将其清空，总不能重新声明一个新的数组吧……太奢侈了……

在头文件 `cstring` 下有函数 `memset`，在程序开头加上 `#include<cstring>` 后即可使用。

例如，当数组 `a` 已经使用过，我们现在要将其清空为 0，这样写

```
memset(a,0,sizeof a);
```

即可实现。

`memset` 函数有三个参数，第一个参数可用于传入待清空的数组名，第三个参数一般使用 `sizeof` 关键字传入待清空部分的大小¹，第二个参数看似比较自由，但可以取的有效值有限。

第二个参数的取值需要满足一定的要求。对于 `int` 型数组，一般只能取 0 或 -1，相应地表示将数组的全体元素设置为 0 或 -1（其实还有更多常用取值，以后用到再谈）。对于 `bool` 型数组，则取 `true` 或 `false`。

`memset` 函数简洁而高效，可也有一定的局限性。当 `memset` 函数无法满足我们的需求时，我们就要用到 `for` 循环。

• 批量赋值

举个例子，应该就懂了：

```
for(int i=0; i<10; ++i)
    a[i]=i;
```

此时的数组 `a` 相当于就是从 0~9 的一系列数。

• 计数

我们希望看看刚刚赋值好的数组 `a` 中有多少个 3 的倍数。我们的手法是，在循环外定义一个计数变量 `cnt`（约定俗成，也可使用 `count` 或者 `c`）并初始化为 0，在循环内每找到一个符合条件的就给 `cnt` 加一。像这样：

```
int cnt=0;
for(int i=0; i<10; ++i)
    if(a[i]%3==0)
```

¹ 这里的大小不是数组的元素个数，而是数组在内存中占用的字节数。


```
    ++cnt;
cout<<cnt<<endl;
```

这时，计数变量 *cnt* 中就记有了数组 *a* 中 3 的倍数的个数。

- 求和

我们希望将 *a* 的 10 个元素之和求出来。方法与上面类似，在循环外定义求和变量 *s*（约定俗成，*sum* 等亦可）并初始化为 0，遍历逐个函数，累计相加。像这样：

```
int s=0;
for(int i=0; i<10; ++i)
    s+=a[i];
cout<<s<<endl;
```

这里出现了一个新的运算符：**+=**。

s+=a[i]与**s=s+a[i]**等价，而代码看起来更简洁。当我们需要给一个变量自己增加某个特定的数值时，应当使用**+=**运算符来简化自己的代码。

与**+=**运算符同理的，还有**-=**、***=**、**/=**、**%=**等，都表示一边进行运算一边顺便更新原来变量的值。

这样，通过上面的语句，就实现了输出数组 *a* 中全体元素之和的目的。

多一点思考，假如我要输出 *a* 中全体元素之积，我需要修改哪些地方？（提示：一共有两处需要修改。）

以上简单介绍一下 C++ 数组常用的批量操作，如果你想要掌握得更扎实一些，可以先自己将这些操作写一写，再继续向下看哦~

【向更高维度进发】

C++ 支持多维数组，它与上面介绍的一维数组形式上的不同在于，后面跟了更多中括号。

二维数组是最常用的多维数组，例如我们要存储一个矩阵，一般就要用到二维数组。像这样：

```
int matrix[2][2];
```

我们就声明了一个二阶矩阵。

多维数组同样可以用花括号的形式初始化，例如：

```
int matrix[2][2]={1,2},{3,4}};
```

我们就得到了矩阵 $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ 。

相应的，访问和修改数组元素也和刚刚所学基本相同，例如：

```
cout<<matrix[0][1]<<endl;
```

就输出了第 0 行第 1 列²的数值 2。再例如：

```
matrix[0][1]=0;
```

我们的矩阵就变成了 $\begin{bmatrix} 1 & 0 \\ 3 & 4 \end{bmatrix}$ 。

当然，我们现在需要建立的素数表只是一个线性的数表，用一维数组即可搞定。

【筛法求素数】

我们理想中的素数表大概像这样：

2	3	4	5	6	7	8	9	10	11	12	13
√	√	×	√	×	√	×	×	×	√	×	√

为了实现这个目的，我们需要一个 bool 型数组：

```
bool prime[1000010];
```

¹ 在这个例子中，由于初始化时每一行都已经填满，因此内层花括号可以省略：`int matrix[2][2]={1,2,3,4};`

² 约定俗成，在二维数组的表达形式中，先行后列。

筛法求素数的核心原理是，先假设一张表中全体元素均为素数，然后将各个数的倍数（除了本身）依次划掉，被划掉的元素就不再是素数了，而幸存下来的就是素数。

下面我们用 2~30 的素数表为实例，具体阐释它的原理。

首先，将所有的数都先设定为素数：

```
memset(prime,true,sizeof prime);
```

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

现在用 for 循环对所有数逐个讨论。

看到 2，发现还没有被划掉，看来是素数，我们需要再嵌套一层 for 循环把 2 的倍数（除了自身）全部划掉，就变成了这个局面：

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

现在 2 的任务完成了，下面轮到 3 了，也是同样的操作：

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

3 处理完成后，再往下看，看到 4，发现 4 已经不是素数了。本来，也是要将 4 的倍数全部划掉的，但现在看来没有这个必要了（因为 4 要划掉的，已经先被 2 划完了）。

所以轮到 5：

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

然后 6 不是素数，跳过，再是 7：

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

虽然数据规模导致了它一点动静都没有，但按照算法还是要继续看下去。

当然，再接下来还是一点动静都没有，所以上面的这张表就是最终的表格了。

通过这一波分析，相信你已经基本明白如何进行操作了。下面就是写代码的功夫了。

这个代码里面要写到什么呢……

bool 型数组，for 循环，if 语句……

如果说稍有一点难点，就是 for 循环的嵌套：外层 for 循环遍历表中全体元素，内层 for 循环负责把当前元素的倍数全部划掉。逻辑还是很清晰的。

需要注意的是循环变量的管理。一般来说，外层变量想不到起什么名字就用 *i*，相应地，内层变量想不到起什么名字就用 *j*。（我觉得你应该明白内外层循环变量绝对不能用同一个名字吧……）

而对于内层循环而言，外层循环的循环变量就是一个可以自由使用的新的变量。比如这一题中我们用 *i* 表示 2、3、5、7……这些负责划掉别人的数，那么内层的 *j* 相应地可以使用诸如 *j+=i* 之类的操作。

你如果足够有信心，可以先尝试着自己写一写代码。如果还是云里雾里，可以先看给出的参考程序，搞明白后自己再重新打一遍。

【参考程序 07 - primetable】

```
1 #include<iostream>
```

```

2  #include<cstring>                                //memset 函数要用到
3  using namespace std;
4
5  /*
6   参考程序
7   07 - primetable
8   请不要照抄
9   */
10
11 int main(){
12     freopen("primetable.in","r",stdin); freopen("primetable.out","w",stdout);
13     int n; cin>>n;
14     bool prime[1000010];                            //数组适当开大一点
15     memset(prime,true,sizeof prime);                //将 prime 数组默认全部初始化为真
16     for(int i=2; i<=n; ++i)                          //对 2~n 进行穷举
17         if(prime[i]){                                //只有 i 是素数才需要用它来划其它的数
18             cout<<i<<endl;                          //顺带先把 i 输出
19             for(int j=i*2; j<=n; j+=i)                //j 穷举所有 i 的倍数, 从 i*2 开始
20                 prime[j]=false;                      //将 j 从素数表中划掉, 标记为 false
21         }
22     return 0;
23 }

```

啊呀……不好意思, 一不小心又放了你不认识的语法了……

//表示注释, 在这一行中, 从//到最后的内容都是注释, 编译器不读的。

只注释一行不过瘾? /*...*/是块状注释, 可以轻松注释掉一整块内容。

注释的作用, 主要有两个。一是让自己的代码易于被别人所理解, 二是在自己写代码的过程中起到暂时隐藏某段代码的作用。第二条作用可能会在你反复调试代码的时候用得多一些。

扯远了, 还是回过来把这题做好吧……做好了还有下一个呢……

选择排序 (selection)

【题目描述】

对于一组数据, 我们常常需要将它们按照一定顺序排序。

通过这道题, 你将学习选择排序这一排序算法, 并使用选择排序将一组数据按照从小到大的顺序排序并输出。

【输入格式】

从文件 *selection.in* 中读入数据。

输入的第一行为一个正整数 n , 表示共有 n 个数据。

输入的第二行为 n 个正整数, 表示待排序的 n 个数据。

同一行输入的相邻两个元素之间, 用恰好一个空格隔开。

【输出格式】

输出到文件 *selection.out* 中。

输出结果仅 1 行，为 n 个用恰好一个空格分开的数据，表示输入的 n 个数据从小到大排序的结果。注意，请不要在行尾输出多余的空格。

【输入样例】

```
5
2 6 3 1 5
```

【输出样例】

```
1 2 3 5 6
```

【数据规模与约定】

保证 n 不超过 10000。

【选择排序算法】

我们直接从实例入手理解这个算法。我们将使用一些拟人化的口吻。

第一轮（从 2 出发）：

2	6	3	1	5
---	---	---	---	---

一切开始。处在最左边的 2 很得意，可他向右看了看，2、6、3、1、5，发现 1 是 5 个人中最小的。

“1 才应该是处于最左边的呀……”

于是 2 提出要和 1 交换位置。

1	6	3	2	5
---	---	---	---	---

此时，1 心满意足地站到了最左边，那个属于他的位置。

第一轮结束（最小的数 1 确定）。

第二轮（从 6 出发）：

1	6	3	2	5
---	---	---	---	---

1 已经确定，不需要再管了，因此 6 向右看，6、3、2、5，发现 2 是剩下 4 个人中最小的。

“看来我应该和 2 换一换……”

于是 6 和 2 交换了位置。

1	2	3	6	5
---	---	---	---	---

此时，1 和 2 都心满意足，站在了各自的位置上。

第二轮结束（前两小的数确定）。

第三轮（从 3 出发）：

1	2	3	6	5
---	---	---	---	---

当 3 向右看的时候，发现 3、6、5 中只有自己最小，而 1 和 2 又已经确定了，那么自己当仁不让处在这个位置。

第三轮结束（前三小的数确定）。

第四轮（从 6 出发）：

1	2	3	6	5
---	---	---	---	---

这时，只有 6 和 5 还没敲定了，而 5 又是两个人中较小的，自然，两个人交换一下，就完成了。

1	2	3	5	6
---	---	---	---	---

第四轮结束（排序完成）。

现在再来理解选择排序算法的一般算法。

为了方便说明，我们统一将 n 个数据依次存入 $a[1], a[2], \dots, a[n]$ 。

我们容易看出，这里需要用一层 for 循环控制轮数：

```
for(int i=1; i<n; ++i)
```

接下来，for 循环里面应该做什么呢？

结合之前的实例阐述，我们发现，需要在 $a[i]$ 到 $a[n]$ 之间找到最小的那个数的下标，如果不是 i ，那就要将两个位置的数据进行交换。

看来，还需要嵌套一层循环。这一层循环如何实现呢？

我们不妨再次回到第一轮实例，以这一轮为例，看看这一步细节是如何操作的。

一切开始。处在最左边的 2 很得意……

2	6	3	1	5
---	---	---	---	---

他以为自己就是 5 个人中最小的了。但他还没有把握……

于是他向右看，6，比自己大；再向右看，3，还比自己大。

直到他看到了 1，发现 1 比自己小。这虽然不能说明 1 一定是最小的，但至少“5 人中最小”的称号是轮不到 2 的了。没办法，这一称号只能暂时移交给 1。

2	6	3	1	5
---	---	---	---	---

既然 2 已经没有资格再声称自己是“5 人中最小”了，那么与别人比较的权利也只有 1 拥有了。

当然，1 也只需要再比较一次。他向右看，5，比自己大。稳了！

这一轮比较结束，1 成为了真正意义上的“5 人中最小”者。

“1 才应该是处于最左边的呀……”

于是 2 提出要和 1 交换位置……

现在一切都明朗了。上面的情境中所描述的“5 人中最小”的称号，对应的就是我们试图找到的最小数的下标；而这个位置的变动的过程也就是我们寻找这个下标的过程。

来看代码怎么写。我们先将这个下标暂时性地定为 i ：

```
int min_pos=i;
```

内层的 for 循环就从下一个位置 $i+1$ 开始遍历：

```
for(int j=i+1; j<=n; ++j)
```

一旦发现某一个下标对应的数比当前的这个最小值要小：

```
if(a[j]<a[min_pos])
```

不好意思，只能让位，更新 min_pos ，让新的 min_pos 再往后比较：

```
min_pos=j;
```

这样，内层循环就完事了。至此， min_pos 存放的，也就是我们要找的在 $a[i]$ 到 $a[n]$ 之间找到最小的那个数的下标了。

但是这一轮（第 i 轮）还没有结束。如果最小的那个数（ $a[min_pos]$ ）不处于应该在下标（ i ）上：

```
if(min_pos!=i)
```

我们需要交换这两个位置上的数。

交……交换？

交换两个数很简单，只需要第三个数的介入。

例如我们需要交换 a 和 b 的值，引入第三个数 t ，让它暂存 a 的数据。接着，将 b 赋给 a ， a 的值就变成了 b 的值。可 b 也需要 a 的值呀，那只要问 t 把暂存的 a 的数据拿回来即可。在这个问题中，就像这样：

```
int t=a[i];
```

```
a[i]=a[min_pos];
```

```
a[min_pos]=t;
```

但是交换两个数还可以更简单。使用 C++ 的 swap 函数，轻松搞定：

```
swap(a[i],a[min_pos]);
```

于是，外层的 for 循环也解决了。

下面就剩输出了。

我们再来看看输出文件格式要求：

输出结果仅 1 行，为 n 个用恰好一个空格分开的数据，表示输入的 n 个数据从小到大排序的结果。

注意，请不要在行尾输出多余的空格。

我们之前可能都习惯了每一个 `cout` 最后都加一个 `<<endl`，不过需要明白，加上 `<<endl` 的目的在于输出一个换行符。可当我们不需要输出换行符的时候，`<<endl` 完全不必加上。

而且，输出换行符，不只有 `<<endl` 这一种途径。

【尝试】

看看这条语句会输出什么：

```
cout<<"Hello,\nWorld!\n";
```

如果一切正常，应当得到的结果是：

```
Hello,
```

```
World!
```

这意味着，在字符串中，`\n` 同样表示换行符。

现在，我把输出的模块贴在下面，自己仔细理解一下，相信读者一定能够明白这是什么意思。

```
for(int i=1; i<=n; ++i)
```

```
    cout<<a[i]<<(i<n?" ":"\n");
```

非常简洁，但也能完成任务了。

到这里，我想，你把上面的几行代码拼起来，也能完成这道题了吧……

不过还是希望读者能够独立手打一遍哦（

【但看了这个你可能会感觉被坑了】

在 C++ 的 `algorithm` 库中，有 `sort` 函数可以实现排序功能。需在开头添加 `#include<algorithm>`。

当我们需要对数组 a 进行从小到大排序，排序的下标范围是 $[p, q)$ （注意左闭右开）时，我们的语句为：

```
sort(a+p,a+q);
```

等价地，若排序部分的初始下标为 p ，排序部分共有 l 个元素，则我们的语句同样可以写成：

```
sort(a+p,a+l+p);
```

其中， a 必为数组名， p 、 q 、 l 等必为整数值。

例如，在本题中，我们用下面的语句，即可替代长长的二重循环：

```
sort(a+1,a+n+1);
```

而且，当你学习了更多知识后会明白，`sort` 的效率远高于这里介绍的排序算法。

下面，把中间的二重循环替换成 `sort` 函数，再运行测评一下，看看结果如何。

【万能头文件】

不知你现在有没有一种感觉……我们前前后后已经用了好多个不同的头文件了：

`iostream`、`cmath`、`cstring`、`algorithm`……

而且以后可能还会遇见更多。值得庆贺的是，C++ 提供了万能头文件，有了它，再也不用写那么多 `include` 语句了。

把现在程序所有的 `#include<...>` 替换成 `#include<bits/stdc++.h>`，并跳转到本教程的第 2 页，按照“新手上路”一节中设置缺省源的方法，把原来的缺省源中的头文件也换成这个万能头文件，从此摆脱写一堆头文件的烦恼啦¹！

¹ 此处用语略有夸张。`windows.h` 头文件（以后会遇到）由于是基于 Windows 系统的，并没有被“万能头文件”纳入。

【从大到小排序】

我如果要用 sort 实现从大到小排序呢？

sort 函数有可选的第三个参数。先不说理论，看例子。

```
1  #include<bits/stdc++.h>
2  using namespace std;
3
4  bool cmp(int x,int y){
5      return x>y;
6  }
7
8  int main(){
9      int n,a[10010]; cin>>n;
10     for(int i=1; i<=n; ++i) cin>>a[i];
11     sort(a+1,a+n+1,cmp);
12     for(int i=1; i<=n; ++i)
13         cout<<a[i]<<(i<n?" ":"\n");
14     return 0;
15 }
```

这里，与从小到大排序的不同点在于两个：

- 第 4~6 行新定义了一个函数 `bool cmp(int, int)`;
- 第 11 行调用 sort 函数的地方将 `cmp` 这个函数作为参数传递给了 sort 函数。

当 sort 函数接收了第三个参数后，它是有任务的。它要保证：

对于任意的排序范围内的下标 i 和 j ($i < j$)，均有 `cmp(a[i],a[j])` 为真。

具体的内部实现原理在这里先不讲，目前先自己打一打，会用即可。因为我们有下面这道题。

成绩排名 (student)

【题目描述】

一场考试下来，几家欢喜几家愁，因为老师要给同学们排成绩了。

某班有一些学生，他们参加了考试，也都有了对应的成绩。现在，你将收到一份花名册。请你设计程序将学生的成绩按照从高到低排序，并按次序输出学生的姓名。

【输入格式】

从文件 `student.in` 中读入数据。

输入的第一行为一个正整数 n ，表示共有 n 名学生。

输入的第 2 到 $n+1$ 行是这 n 名学生的成绩信息，每一行由两个用恰好一个空格隔开的数组成，第一个是学生的姓名，为字符串类型，第二个是学生的成绩，为整数类型。

【输出格式】

输出到文件 `student.out` 中。

输出结果共 n 行，依次表示排序后学生的姓名。

【输入样例】

```
5
Alice 6
Bob 8
Daniel 7
Leo 10
Tom 9
```

【输出样例】

```
Leo
Tom
Bob
Daniel
Alice
```

【数据规模与约定】

本题只有 1 个测试点。这个测试点与样例相同。（毕竟这题造测试数据太难了……）

【自己定义一种新的数据结构】

C++允许定义新的数据结构，需要使用 `struct` 语句。

`struct` 语句的语法结构如下：

```
struct 结构名称{
    成员数据类型 1 成员名称 1;
    成员数据类型 2 成员名称 2;
    .....
};
```

例如，我们希望创建一个新的数据结构来表示学生，结构名称就叫 `Student`¹。根据题意，一个 `Student` 结构需要有两个成员数据，即姓名（`name`）和成绩（`score`）。因此，`Student` 结构应该这样定义：

```
struct Student{
    string name;
    int score;
};
```

为了让 `Student` 结构全局可用，应当将 `Student` 的定义放在所有函数之外、需要用到 `Student` 的所有函数之前。这样，定义了 `Student` 结构，`Student` 就和 `int`、`bool`、`string` 之类的基本等同。例如，我们要声明一个学生 `a`：

```
Student a;
```

当我们希望访问 `a` 的成员数据时，需要用成员访问运算符（`.`）（就是一个点），例如 `a.name`，`a.score`。

我们可以对 `a` 的成员数据进行赋值：

```
a.name="Alex"; a.score=9;
```

我们也可以取出 `a` 的成员数据的值，并把它放到输入输出流上：

```
cout<<a.name<<" "<<a.score<<endl;
```

当然，为了解决这道题，我们需要声明一个数组：

¹ 约定俗成，自定义类型的名称用首字母大写的规范进行命名。


```
Student s[10];
```

【为 Student 类型写一个 cmp 函数】

在这一题中，我们为了要让两个 Student 之间可以比较从而进行排序，需要为 Student 类型专门写好 cmp 函数。

这是我们之前写的 cmp 函数：

```
bool cmp(int x,int y){  
    return x>y;  
}
```

它实现了两个 int 型数据之间的比较关系的规定。现在我们要将 int 换成 Student：

```
bool cmp(Student x,Student y)
```

然而，Student 类型还不支持进行 > 的运算，因此第二行也得改。改成什么呢？

我们要比较的是两个 Student 各自的 score，因此应该写：

```
    return x.score>y.score;
```

这样，我们的 cmp 就重新写好了。

【来把这题实现吧】

我想，这一题的核心部分我都已经展示出来了，下面就是一些输入输出了，我相信读者一定可以独立把这一题完成了。

分数 (fraction)

【题目描述】

这里的分数不是 score。

这里的分数是 fraction。

你将得到两个分数：

$$x = \frac{a}{b}, y = \frac{c}{d}$$

请你用分数的形式输出：

$$x + y, x - y, xy, \frac{x}{y}$$

分数要化到最简形式。

【输入格式】

从文件 *fraction.in* 中读入数据。

输入数据为一行四个整数 a, b, c, d ，含义见题目描述。

同一行输入的相邻两个整数之间用恰好一个空格隔开。

【输出格式】

输出到文件 *primetable.out* 中。

输出结果共有 4 行，依次表示题目描述中要求得到的四个分数。

每一行为用恰好一个空格隔开的两个整数，分别表示分子和分母

【输入样例】

1 2 1 3

【输出样例】

5 6

1 6

1 6

3 2

【数据规模与约定】

保证所有数据均为正整数。保证 $x > y$ 。

【数据结构¹】

很显然，这题不用 struct 可以更快地做出来。但是我想借本题介绍一些更高级的操作。

我们定义了这样一个结构：

```
struct Fraction{
    int a,b;
};
```

但是我们现在要将其功能变得更加丰富。

【构造函数】

所谓一个 struct 的**构造函数**，是指一个 struct 在被声明时首先被调用的函数，它一般被用于初始化。struct 的构造函数的函数名必须与 struct 的名称同名，函数类型为 void，但 void 必须省略不写。

例如，假如我想让一个 Fraction 在声明时默认初始化为 1/1，那么我可以这样写：

```
struct Fraction{
    int a,b;
    Fraction(){
        a=1;
        b=1;
    }
};
```

当我们之后像这样

```
Fraction x;
```

声明了一个 Fraction 时，**x.a** 和 **x.b** 就都被赋值成 1 了。

不过，对于现在这一道题，暂且不需要这样的初始化。我们使用更加厉害一点的形式：

```
struct Fraction{
    int a,b;
    Fraction(int A,int B){
        a=A;
        b=B;
    }
};
```

¹ 这一题有关内容的讲解很重要，希望读者可以将这一节的内容当成第二个重难点来学习。

这样，在后面声明时也不能简单地使用 `Fraction x`；了，而是可以像这样：

```
Fraction x=Fraction(1,2);
```

我来解释一下这样做的原理。在这一语句中，第一个 `Fraction` 是类型名称，表示变量 x 是 `Fraction` 数据类型的。

第二个 `Fraction` 则是 `Fraction` 这一结构的构造函数的函数名。`Fraction(1,2)`调用了构造函数，构造函数首先构造了一个类型为 `Fraction` 的结构，有数据成员 a 和 b 。接着，构造函数接收到了传入参数 A 和 B ，其中 $A=1, B=2$ 。然后，执行构造函数的函数体， $a=A, b=B$ ，两条语句依次运行结束后，这个新构造的 `Fraction` 结构的 a 和 b 就根据传入的 A 和 B 进行初始化了。

经过一波操作，`Fraction(1,2)`构造好了一个新的 `Fraction`（对应 $1/2$ ），然后通过上面这条赋值语句传给了 x 。

把这种方式 and 下面这样对比着看：

```
Fraction x;
```

```
x.a=1;
```

```
x.b=2;
```

在实际使用时，当然是你觉得哪一种简洁就用哪一种（

【多个构造函数】

然而，用了这种形式（为了避免指代不明我把代码再摘下来）：

```
struct Fraction{
    int a,b;
    Fraction(int A,int B){
        a=A;
        b=B;
    }
};
```

你会发现，当你试图这样声明一个 `Fraction` 结构时：

```
Fraction x;
```

编译器会报错。为何？

编译器遇到一句孤零零的 `Fraction x`；时，会主动寻找构造函数，并且希望构造函数的形式是这样的：

```
Fraction(){
    ...
}
```

也就是说是不需要传入参数的，毕竟这才是默认的构造函数。

如果找不到构造函数也就罢了，就像最原始的那种一样：

```
struct Fraction{
    int a,b;
};
```

现在倒是找到了构造函数，但找到的却是 `Fraction(int,int)`型，自带传入参数，无法匹配，那不报错还得了？

为了解决这个问题，我们可以假装写一个它要的默认构造函数，像这样：

```
struct Fraction{
    int a,b;
    Fraction(){}
    Fraction(int A,int B){
```

```

        a=A;
        b=B;
    }
};

```

你会发现新添的 `Fraction(){}` 什么都没做，就是为了给孤零零的 `Fraction x`; 寻找构造函数时找一个目标。

看上去可行。但有一点小小的顾虑：定义了两个 `Fraction` 函数，不怕两个函数自己先打架么？

【初始函数重载】

这里就提到了 C++ 处理同名函数的一种机制：**函数重载**。

在同一作用域内，如果有一组函数，它们具有相同的函数名，但它们的参数列表各不相同，那么这组函数就被称为一组**重载函数**，处理一组重载函数的机制称为**函数重载**。

这样的定义中有着潜台词，就是在同一作用域内的两个同名函数如果要不发生冲突，必须要保证两者的参数列表各不相同。

一组重载函数的实现方式往往有所差异，但因为这组重载函数的参数列表各不相同，所以当我们对其进行调用时，在大部分情况下，是可以选出一个匹配的函数进行运算的。

一组重载函数的实现方式往往又有一定的共通性，正是这种共通性让这些函数得以共用一个函数名。对于开发者而言，函数重载机制免除了程序员为功能相近的函数起不同的名称的麻烦，只要保证它们参数列表不同即可。

回到上面的例子，自然可以想见，两个 `Fraction` 构造函数因为参数列表不同而不会发生冲突。

而在调用时，语句 `Fraction x`; 会寻找匹配的构造函数 `Fraction(){}` ，而语句 `Fraction x=Fraction(1,2)`; 也会寻找相应的构造函数 `Fraction(int A,int B){...}`，两者均可实现自己的目的。

至此，我们已经能够从给定的数据中构造出指定的 `Fraction` 数据结构了。既然是数据结构，应该可以进行运算的吧……

【运算符重载】

我们的目标是，对于

```
Fraction x,y;
```

要让 `x+y`, `x-y`, `x*y`, `x/y` 都能通过编译。

就现在这样写肯定通过不了。为什么？因为 `+`、`-`、`*`、`/` 四则运算符号现在还不认识 `Fraction` 结构。我们要做的工作，就是让它们认识我们自己创造出的 `Fraction` 结构。

注意到，运算符其实也是一种函数。函数能重载，运算符当然也能重载。

以重载 `+` 运算符为例。如果将 `+` 运算符视为一个函数，那么定义在 `Fraction` 结构上的 `+` 应当接收的参数为两个 `Fraction`，返回的是一个表示两者之和的新的 `Fraction`。函数声明的形式应该像这样：

```
Fraction operator +(Fraction x,Fraction y)
```

这和我们之前熟悉的自定义函数形式很像，唯一的区别是，函数名变成了 `operator +`。可以理解为是通过 `operator` 关键字，将 `+` 运算符转换为了对应的加法函数的函数名。另外三个运算符也是同理。

里面怎么写呢？利用 $\frac{a}{b} + \frac{c}{d} = \frac{ad+bc}{bd}$ ，我们可以将分子和分母的表达式写出来：

```
int X=x.a*y.b+x.b*y.a;    //分子
int Y=x.b*y.b;            //分母
```

写成这样，根据题意，别忘了约分。约分的操作是，分子和分母同时除以两者的最大公约数。最大公约数函数之前已经介绍过，是 `__gcd`，因此：

```
int gcd=__gcd(X,Y);
```

然后，我们的返回值应该也是个 Fraction，且经过约分，那就有：

```
return Fraction(X/gcd,Y/gcd);
```

把这些都拼起来，一个重载好的+运算符就完成了：

```
Fraction operator +(Fraction x,Fraction y){  
    int X=x.a*y.b+x.b*y.a;  
    int Y=x.b*y.b;  
    int gcd=__gcd(X,Y);  
    return Fraction(X/gcd,Y/gcd);  
}
```

下面，请自己写好-、*、/的重载函数。

写好之后，我们来看主程序。

主程序我们先声明三个 Fraction：

```
Fraction x,y,z;
```

然后输入：

```
cin>>x.a>>x.b>>y.a>>y.b;
```

接着，当我们需要得到 $x + y$ 时，就能放心大胆地这样写：

```
z=x+y;
```

然后输出：

```
cout<<z.a<<" "<<z.b<<endl;
```

另外三个同理，请自行独立完成，然后编译测评，看看是不是全都过关了。

如果过关了，这不仅意味着你已经熟练掌握了 struct 的基本功能，更意味着，至此，你掌握的 C++ 语法，已经足够应付绝大多数基础的 C++ 问题了。

【写在这一册结尾的话】

我们的标题是：C++快速入门。

那么它的特点应该是，比同类教程可以让你更快更顺利地走进 C++ 的大门。

假如一切如我所预计的那样，现在的你应该已经在较短的时间内具备了基本的编写 C++ 程序的能力。

但也正因如此，这一册教程仅仅是让你熟悉了 C++ 常用的语法而已。

论其效果，仅仅 10 道习题，并不能让你有足够的熟练度。

论其深度，好多语法还有很多更深层次的功能没有介绍到。

论其广度，还记得在本册开头提到的 C++ 的两个加号吗？标准模板库的功能我们才熟悉了其冰山一角，面向对象的编程特性我们更是都没有接触到。

那怎么办？

习题不够？我们将定期推送精选习题，进一步提升你的程序设计能力。你也可以借助在线题库（如洛谷 <http://www.luogu.org>），寻找海量试题。

深度和广度不够？别急，上册的主要功能在于快速让你熟悉 C++ 的基本语法，而下册，更多更深的 C++ 语言特性将会逐个浮出水面。