代码框架

如果你不想深究背后的原理，初学时可以直接将这个“框架”背下来：

```
#include <cstdio>
#include <iostream>

int main() {
  // do something...
  return 0;
}
```

??? note "什么是 include？" `#include` 其实是一个预处理命令，意思为将一个文件“放”在这条语句处，被“放”的文件被称为头文件。也就是说，在编译时，编译器会“复制”头文件 `iostream` 中的内容，“粘贴”到 `#include <iostream>` 这条语句处。这样，你就可以使用 `iostream` 中提供的 `std::cin`、`std::cout`、`std::endl` 等对象了。

    如果你学过 C 语言，你会发现目前我们接触的 C++ 中的头文件一般都不带 `.h` 后缀，而那些 C 语言中的头文件 `xx.h` 都变成了 `cxx`，如 `stdio.h` 变成了 `cstdio`。因为 C++ 为了和 C 保持兼容，都直接使用了 C 语言中的头文件，为了区分 C++ 的头文件和 C 的头文件，使用了 `c` 前缀。

    一般来说，应当根据你需要编写的 C++ 程序的需要来确定你要 `#include` 哪些头文件。但如果你 `#include` 了多余的头文件，只会增加编译时间，几乎不会对运行时间造成影响。目前我们只接触到了 `iostream` 和 `cstdio` 两个头文件，如果你只需要 `scanf` 和 `printf`，就可以不用 `#include <iostream>`。

    可以 `#include` 自己写的头文件吗？答案是，可以。

    你可以自己写一个头文件，如：`myheader.h`。然后，将其放到和你的代码相同的目录里，再 `#include "myheader.h"` 即可。需要注意的是，自定义的头文件需要使用引号而非尖括号。当然，你也可以使用编译命令 `-I <header_file_path>` 来告诉编译器在哪找头文件，就不需要将头文件放到和代码相同的目录里了。

??? note "什么是 `main()` ？" 可以理解为程序运行时就会执行 `main()` 中的代码。

    实际上，`main` 函数是由系统或外部程序调用的。如，你在命令行中调用了你的程序，也就是调用了你程序中的 `main` 函数（在此之前先完成了全局 [变量](./var.md) 的构造）。

    最后的 `return 0;` 表示程序运行成功。默认情况下，程序结束时返回 `0` 表示一切正常，否则返回值表示错误代码（在 Windows 下这个错误代码的十六进制可以通过 [Windows Error Codes 网站](https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-erref/) 进行查询）。这个值返回给谁呢？其实就是调用你写的程序的系统或外部程序，它会在你的程序结束时接收到这个返回值。如果不写 `return` 语句的话，程序正常结束默认返回值也是 `0`。

> 在 C 或 C++ 中，程序的返回值不为 0 会导致运行时错误（RE）。

注释

在 C++ 代码中，注释有两种写法：

1. 行内注释

   以 // 开头，行内位于其后的内容全部为注释。

2. 注释块

   以 /* 开头，*/ 结尾，中间的内容全部为注释，可以跨行。

注释对程序运行没有影响，可以用来解释程序的意思，还可以在让某段代码不执行（但是依然保留在源文件里）。

在工程开发中，注释可以便于日后维护、他人阅读。

在 OI 中，很少有人写许多注释，但注释可以便于在写代码的时候理清思路，或者便于日后复习。而且，如果要写题解、教程的话，适量的注释可以便于读者阅读，理解代码的意图。

输入与输出

cin 与 cout

```cpp
#include <iostream>

int main() {
  int x, y;                          // 声明变量
  std::cin >> x >> y;                // 读入 x 和 y
  std::cout << y << std::endl << x;  // 输出 y，换行，再输出 x
  return 0;                          // 结束主函数
}
```

???+note "什么是变量？" 可以参考 变量 页面。

???+note " 什么是 std ？" std 是 C++ 标准库所使用的 **命名空间**。使用命名空间是为了避免重名。

> 关于命名空间的详细知识，可以参考 [命名空间](./namespace.md) 页面。

scanf 与 printf

scanf 与 printf 其实是 C 语言提供的函数。大多数情况下，它们的速度比 cin 和 cout 更快，并且能够方便地控制输入输出格式。

```cpp
#include <cstdio>

int main() {
  int x, y;
  scanf("%d%d", &x, &y);   // 读入 x 和 y
  printf("%d\n%d", y, x);  // 输出 y，换行，再输出 x
  return 0;
}
```

其中，`%d` 表示读入/输出的变量是一个有符号整型（`int` 型）的变量。

类似地：

1. `%s` 表示字符串。
2. `%c` 表示字符。
3. `%lf` 表示双精度浮点数（`double`）。
4. `%lld` 表示长整型（`long long`）。根据系统不同，也可能是 `%I64d`。
5. `%u` 表示无符号整型（`unsigned int`）。
6. `%llu` 表示无符号长整型（`unsigned long long`），也可能是 `%I64u`。

除了类型标识符以外，还有一些控制格式的方式。许多都不常用，选取两个常用的列举如下：

1. `%1d` 表示长度为 1 的整型。在读入时，即使没有空格也可以逐位读入数字。在输出时，若指定的长度大于数字的位数，就会在数字前用空格填充。若指定的长度小于数字的位数，就没有效果。
2. `%.6lf`，用于输出，保留六位小数。

这两种运算符的相应地方都可以填入其他数字，例如 `%.3lf` 表示保留三位小数。

??? note "“双精度浮点数”，“长整型”是什么" 这些表示变量的类型。和上面一样，会留到 变量 中统一讲解。

??? note "为什么 `scanf` 中有 `&` 运算符？" 在这里，`&` 实际上是取址运算符，返回的是变量在内存中的地址。而 scanf 接收的参数就是变量的地址。具体可能要在 指针 才能完全清楚地说明，现在只需要记下来就好了。

??? note " 什么是 `\n` ？" `\n` 是一种 **转义字符**，表示换行。

    转义字符用来表示一些无法直接输入的字符，如由于字符串字面量中无法换行而无法直接输入的换行符，由于有特殊含义而无法输入的引号，由于表示转义字符而无法输入的反斜杠。

    常用的转义字符有：

    1. `\t` 表示制表符。

    2. `\\` 表示 `\`。

    3. `\"` 表示 `"`。

    4. `\0` 表示空字符，用来表示 C 风格字符串的结尾。

    5. `\r` 表示回车。Linux 中换行符为 `\n`，Windows 中换行符为 `\r\n`。在 OI 中，如果输出

需要换行，使用 `\n` 即可。但读入时，如果使用逐字符读入，可能会由于换行符造成一些问题，需要注意。例如，`gets` 将 `\n` 作为字符串结尾，这时候如果换行符是 `\r\n`，`\r` 就会留在字符串结尾。

6. 特殊地，`%%` 表示 `%`，只能用在 `printf` 或 `scanf` 中，在其他字符串字面量中只需要简单使用 `%` 就好了。

??? note "什么是字面量？"
    “字面量”是在代码里直接作为一个值的程序段，例如 `3` 就是一个 `int` 字面量，``'c'`` 就是一个 char 字面量。我们上面写的程序中的 `"hello world"` 也是一个字符串字面量。

    不加解释、毫无来由的字面量又被称为“魔术数”（magic number），如果代码需要被人阅读的话，这是一种十分不被推荐的行为。

一些扩展内容

C++ 中的空白字符

在 C++ 中，所有空白字符（空格、制表符、换行），多个或是单个，都被视作是一样的。（当然，引号中视作字符串的一部分的不算。）

因此，你可以自由地使用任何代码风格（除了行内注释、字符串字面量与预处理命令必须在单行内），例如：

```cpp
/* clang-format off */

#include <iostream>

 int

    main(){
int/**/x, y;  std::cin
>> x >>y;
                std::cout <<
          y  <<std::endl
     << x

          ;

    return       0;     }
```

当然，这么做是不被推荐的。

一种也被广泛使用但与 **OI Wiki** 要求的码风不同的代码风格：

```cpp
/* clang-format off */

#include <iostream>
```

```
int main()
{
    int x, y;

    std::cin >> x >> y;
    std::cout << y << std::endl << x;

    return 0;
}
```

#define 命令

#define 是一种预处理命令，用于定义宏，本质上是文本替换。例如：

```
#include <iostream>
#define n 233
// n 不是变量，而是编译器会将代码中所有 n 文本替换为 233，但是作为标识符一部分的
// n 的就不会被替换，如 fn 不会被替换成 f233，同样，字符串内的也不会被替换

int main() {
  std::cout << n;  // 输出 233
  return 0;
}
```

??? note "什么是标识符？" 标识符就是可以用作变量名的一组字符。例如，abcd 和 abc1 都是合法的标识符，而 1a 和 c+b 都不是合法的标识符。

    标识符由英文字母、下划线开头，中间只允许出现英文字母、下划线和数字。值得注意的是，关键字（如 `int`,`for`,`if`）不能用作标识符。

??? note "什么是预处理命令？" 预处理命令就是预处理器所接受的命令，用于对代码进行初步的文本变换，比如 文件包含操作 #include 和 处理宏 #define 等，对 GCC 而言，默认不会保留预处理阶段的输出 .i 文件。可以用 -E 选项保留输出文件。

宏可以带参数，带参数的宏可以像函数一样使用：

```
#include <iostream>
#define sum(x, y) ((x) + (y))
#define square(x) ((x) * (x))

int main() {
  std::cout << sum(1, 2) << ' ' << 2 * sum(3, 5) << std::endl;  // 输出 3 16
}
```

但是带参数的宏和函数有区别。因为宏是文本替换，所以会引发许多问题。如：

```cpp
#include <iostream>
#define sum(x, y) x + y
// 这里应当为 #define sum(x, y) ((x) + (y))
#define square(x) ((x) * (x))

int main() {
  std::cout << sum(1, 2) << ' ' << 2 * sum(3, 5) << std::endl;
  // 输出为 3 11，因为 #define 是文本替换，后面的语句被替换为了 2 * 3 + 5
  int i = 1;
  std::cout << square(++i) << ' ' << i;
  // 输出未定义，因为 ++i 被执行了两遍
  // 而同一个语句中多次修改同一个变量是未定义行为（有例外）
}
```

使用 `#define` 是有风险的（由于 `#define` 作用域是整个程序，因此可能导致文本被意外地替换，需要使用 `#undef` 及时取消定义），因此应谨慎使用。较为推荐的做法是：使用 `const` 限定符声明常量，使用函数代替宏。

但是，在 OI 中，`#define` 依然有用武之处（以下两种是不被推荐的用法，会降低代码的规范性）：

1. `#define int long long+signed main()`。通常用于避免忘记开 long long 导致的错误，或是调试时排除忘开 long long 导致错误的可能性。（也可能导致增大常数甚至 TLE，或者因为爆空间而 MLE）
2. `#define For(i, l, r) for (int i = (l); i <= (r); ++i)`、`#define pb push_back`、`#define mid ((l + r) / 2)`，用于减短代码长度。

不过，`#define` 也有优点，比如结合 `#if` 等预处理指令有奇效，比如：

```cpp
#ifdef LINUX
// code for linux
#else
// code for other OS
#endif
```

可以在编译的时候通过 `-DLINUX` 来控制编译出的代码，而无需修改源文件。这还有一个优点：通过 `-DLINUX` 编译出的可执行文件里并没有其他操作系统的代码，那些代码在预处理的时候就已经被删除了。

`#define` 还能使用 `#`、`##` 运算符，极大地方便调试。