编程基石部分

适合上机实际编程的知识点总结，适合对c++的学习，和在复试上机前再复习

有些人认为学过 C++之后就没有必要学 C 语言，甚至觉得 C 语言太麻烦而不学，这是不太正确的。因为就机试使用的语法而言，除了输入和输出部分，其余顺序结构、分支结构、循环结构、数组、指针都是几乎一样的，学习 C 语言并不会带来什么负担；而让 C++使用者觉得麻烦的`scanf`函数和`printf`函数，虽然必须承认`cin`和`cout`可以不指定输入输出格式比较方便，**但是在具体的编译器上，cin 和 cout 消耗的时间比 scanf 和 printf 大得多，很多题目可能输入还没结束就超时了。**当然可以在某次使用`cin`和`cout`超时、改成`scanf`和`printf`后通过的时候，痛下决心以后使用`scanf`和`printf`（其实就是 C 和 C++混搭的写法）。顺便指出，`cout`和`printf`请***不要同时***在一个程序中使用，这样可能会导致在编译器上出现编译问题。

最后再次强调，讲义中会使用一些代码作为举例，希望能够亲自照着敲一遍理解一下（如果时间不够，可以在复试准备上机前尝试），这样对语言的学习大有帮助。

接下来，就让我们开始学习 C 语言吧。

先来看一段 C 语言小程序：

```
#include <stdio.h>
int main() {
    int a, b;
    scanf("%d%d", &a, &b);
    printf("%d", a+b);
    return 0;
}
```

请学弟妹在编译器中输入这段代码，并保存其为`.cpp`文件（C 语言的文件后缀名为`.c`，但是为了使用 C++中的一些好用的特性，请把文件后缀名改为 C++的文件后缀名`.cpp`）。

这个程序分为两个部分：**头文件**和**主函数**，我们分别来介绍一下它们。

头文件

在上面的代码中，`#include <stdio.h>`这一行就是头文件。其中，`stdio.h`是标准输入输出库，如果在程序中需要输入输出的话，就需要加上这个头文件。不过一般来说程序都是需要输入输出的，所以基本上每一个 C 程序都需要它。

`stdio`的全称是`stantard input output`，`h`就是`head`的缩写，`.h`是头文件的文件格式。于是我们可以这样理解：`stdio.h`就是一个文件，这个文件中包含了一些跟输入输出有关的东西，如果我们的程序需要输入输出的话，就要通过`#include <XXX>`的写法来 include（包含）这个头文件，这样我们才可以使用`stdio.h`这个文件里的输入输出函数。

既然`stdio.h`是负责输入输出的话，那么自然还会有负责其他功能的头文件。例如，`math.h`负责一些数学函数，`string.h`负责跟字符串有关的函数。我们只需要在我们需要使用对应的函数的时候，将它们的头文件`include`到这个程序中来即可。

此外，在 C++的标准中，`stdio.h`更推荐使用等价写法：`cstdio`，也就是在前面加一个`c`，

然后去掉 `.h` 即可。所以 `#include <stdio.h>` 和 `#include <cstdio>` 的写法是等价的，`#include <math.h>` 和 `#include <cmath>` 等价，`#include <string.h>` 和 `#include <cstring>` 也等价。妹在程序中看到这种写法应当能明白它的意思。

主函数

```
int main() {
    ... //函数内容
    return 0;
}
```

化简起来看，上面的代码就是主函数。主函数是一个程序的入口位置，整个程序从主函数开始执行。一个程序最多只能有一个主函数。

下面我们来看一下省略号中的内容，现在暂时只需要大致了解每个语句的作用即可，因为我们会在后面仔细讲解这些语法。

```
int a, b;
```

这句话定义了两个变量 a 和 b，类型是 `int` 型（简单来说就是整数）。

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

`scanf` 用来读入数据，这条语句以 `%d` 的格式输入 a 和 b，其中 `%d` 就是 `int` 型的输入输出标识。简单来说就是把 a 和 b 作为整数输入。

```
printf("%d", a + b);
```

`printf` 用来输出数据，这条语句计算 a + b 并以 `%d` 格式输出。上面说过，`%d` 就是 `int` 型的输入输出标识，所以就是把 a + b 作为整数输出。因此这份代码的主函数实现了读入两个数 a 和 b 然后输出 a+b 的功能。

接下来进入正题，讲解一下各个C 语言中需要使用的语法。

> 注意：下文使用的代码在平常自己联系的时候请保存成 `.cpp` 后缀的文件（即 C++文件），然后使用 C++（或 G++）进行提交。由于 C++向下兼容 C，因此采用这种方式可以尽可能防止一些因为 C 与C++之间的区分而导致的编译错误。

2.1 基本数据类型

2.1.1变量的定义

变量是在程序运行过程中其值可以改变的量，需要在定义之后才可以使用，定义格式如下：

```
变量类型 变量名;
```

并且，变量可以在定义的时候就赋初值：

```
变量类型 变量名 = 初值;
```

变量名一般来说可以任意取，只是需要满足几个条件：

> (1)不能是 C 语言标识符（标识符不多，比如`for`、`if`、`or`等都不能作为变量名，因为它们在 C 语言中本身有含义）。所以我随便取成`TJU`什么的都没关系，不过尽量还是取成有实际意义的变量名，这样可以提高程序的可读性。
>
> (2)变量名的第一个字符必须是字母或下划线，除第一个字符的其他字符必须是字母、数字或下划线。
>
> (3)大小写区分，因此`Tju`和`tju`可以作为两个不同的变量名。

2.1.2变量类型

接下来说下变量类型。

一般来说，基本数据类型分为整型、浮点型、字符型，C++中又有布尔型。每种类型里面又可以分为若干种（为了方便记忆，只列出常用的）：

	类型	取值范围	大致范围
整型	int	-2147483648～+2147483647（即$-2^{31}\sim+(2^{31}-1)$）	$-2*10^9\sim2*10^9$
	long long	$-2^{63}\sim+(2^{63}-1)$	$-9*10^{18}\sim9*10^{18}$
浮点型	float	$-2^{128}\sim+2^{128}$（实际精度 6~7 位）	实际精度6~7位
	double	$-2^{1024}\sim+2^{1024}$（实际精度15~16位）	实际精度15~16位
字符型	char	$-128\sim+127$	$-128\sim+127$
布尔型	bool	0(false) or 1(true)	0(false) or 1(true)

整型

整型一般可以分为短整型`(short)`、整型`(int)`和长整型`(long long)`，其中短整型`(short)`一般用不到，所以不在这里讲述。我们要聊的是整型`(int)`和长整型`(long long)`，其中整型`int`也被称为`long int`，长整型`long long`也被称为`long long int`。

对整型`int`来说，一个整数占用 32 位，也即 4 个字节，取值范围是$-2^{31}\sim2^{31}-1$。如果对范围不太有把握的话，可以记住绝对值在范围以内的整数都可以定义成`int`型。定义举例：

```
int num;
int num = 5;
```

对长整型`long long`来说，一个整数占用 64 位，也即 8 个字节，取值范围是$-2^{63}\sim2^{63}-1$，也就是说如果题目要求的整数取值范围如果超过 2147483647 的话（例如10^{10}或者10^{18}），就得用`long long`来存储。定义举例：

```
long long bignum;
long long bignum = 123456789012345LL;
```

> 注意，如果`long long`型赋大于$2^{31}-1$的初值，需要在初值后面加上`LL`，不然会编译错误。

除此之外，整型数据都可以在前面加个`unsigned`表示无符号型，例如`unsigned int`和`unsigned long long`，占用的位数和原先相同，但是把负数范围挪到整数上来了。也就是说，`unsigned int`的取值范围是$0\sim2^{32}-1$，`unsigned long long`的取值范围是$0\sim2^{64}-1$。一般来说很少会出现必须使用`unsigned int`和`unsigned long long`的情况，因此初学者只需要熟练使用`int`和`long long`即可。

下面我们来看一个例子：

```
#include <cstdio>
int main() {
    int a = 1, b = 2;
    printf("%d", a+b);
    return 0;
}
```

这段代码首先定义了 `int` 型变量 a 和 b，并分别赋初值 1 和 2，然后使用 `printf` 输出了 `a+b`。其中 `%d` 是 `int` 型的输出格式。

数据结果：

```
3
```

简单说来，整型需要记住的是，看到题目要求10^9以内、或者说 32 位整数，就用 `int` 型来存放，而如果是10^{18}以内（例如10^{10}）、或者说 64 位整数，就要用 `long long` 型来存放。

浮点型

浮点型通俗讲就是小数，一般可以分为单精度（`float`）和（`double`）。 对单精度 `float` 来说，一个浮点数占用 32 位，其中 1 位作为符号位、8 位作为指数位、23 位作为尾数位（了解即可），可以存放的浮点数的范围是-2^{128}~$+2^{128}$，但是其有效精度只有 6~7 位。这对于一些精度要求比较高的题目是不合适的。定义举例：

```
float fl;
float fl = 3.1415;
```

对双精度 `double` 来说，一个浮点数占用 64 位，其中依照浮点数的标准，1 位作为符号位、11 位作为指数位、52 位作为尾数位，可以存放的浮点数的范围是-2^{1024} ~ 2^{1024}，其有效精度有 15~16 位，比 `float` 优秀许多。定义举例：

```
double db;
double db = 3.1415926536;
```

> `float` 精度是6-7位，其实因为 `float` 尾数23位，可以存储2^{23}为7位，能精确表示的就是6位。 `double` 精度为15~16位，其实因为 `double` 尾数52位，可以存储2^{52}为16位，能精确表示的就是15位。

下面给出一个浮点型相关的程序:

```
#include <cstdio>
int main() {
    double a = 3.14, b = 0.12;
    double c = a + b;
    printf("%f", c);
    return 0;
}
```

这段代码定义了 `double` 型变量 a 和 b，并分别赋初值 3.14 和 0.12，然后把 a+b 赋值给c，最后输出 c。其中 `%f` 是 `float` 和 `double` 型的输出格式。

输出结果：

```
3.26
```

因此，对浮点型来说只需要记住一点，不要使用`float`，碰到浮点型的数据都应该用`double`来存储。

字符型

(1) 字符变量和字符常量字符型变量的定义方法如下。

```
char c;
char c = 'e';
```

如何理解字符常量？可以先考虑这么一个角度：假设我们现在使用`int num`的方式定义了一个整型变量`num`，那么`num`就是一个可以被随时赋值的变量；而对于一个整数本身（比如 5），它并不能被改变、不能被赋值，那么我们可以把它叫做整型常量。同样的，如果是通过`char c`的方式定义了一个字符，那么 c 在这里就被称作字符变量，它可以被赋值；但是如果是一个字符本身（例如小写字母`'e'`），它是一个没有办法被改变其值的东西，这和前面的整数 5 是一样的，所以我们把它叫做字符常量。事实上，对单个的字符`'Z'`、`'J'`、`'U'`，我们都可以把它们称作字符常量。字符常量可以赋值给字符变量，就跟整型常量可以被赋值给整型变量一样。

在 C 语言中，字符常量使用 ASCII 码统一编码，标准 ASCII 码的范围是 0~127，其中包含了控制字符或通信转用字符（不可显示）和我们平时使用的可显示字符。在我们的键盘上，通过敲击可以在屏幕上显示的字符就是可显示字符，比如 0_9、$_A$Z、a~z 等都是可显示字符，它们的 ASCII 码分别是 48 ~ 57、65 ~ 90、97 ~ 122，不过具体数字不需要记住，只要知道小写字母比大写字母的 ASCII 码值大 32 即可。

> 注：字符常量必须用单引号标注起来，以区分是作为字符变量还是字符常量出现。正如上面的例子，使用`char c`的方式定义了字符变量 c 之后，如果出现了字符常量`'c'`又不加单引号，那么就会产生误解。为此，C 语言规定，字符常量（必须是单个字符）必须用单引号标注，例如上面提到的`'Z'`、`'J'`、`'U'`就都使用了单引号标注，以表明它们是字符常量。

最后来看一个小程序：

```
#include <stdio.h>
int main(){
    char c1 = 't', c2 = 'j', c3 = 117;
    printf("%c%c%c", c1, c2, c3);
    return 0;
}
```

输出结果：

```
tju
```

有些同学可能会感到奇怪，为什么字符型变量`c3`可以被赋值整数 117，而且最后居然输出了字符`'u'`？其实在计算机内部，字符就是按 ASCII 码存储的，`'u'`的 ASCII 码就是 117，因此将 117 赋值给`c3`其实就是把 ASCII 码赋值给`c3`。而且我们还会发现，在赋值的时候， 117 并没有加单引号。因此这种写法是成立的。（最后说明一下，`%c`是`char`型的输出格式。）

(2) 转义字符

上面提到，ASCII 码中有一部分是控制字符，是不可显示的。像键盘上的回车（换行）、删除、Tab 等都是控制字符。那么在程序中怎样表示一个控制字符呢？很简单，一个右斜线加一些特定的字母。打个比方，换行就是使用`\n`来表示，Tab 键通过\t 来表示。在实际做题目时比较常用的就只有下面两个，希望学弟妹能够记住。

> ***`\n`代表换行***
>
> ***`\0`代表空字符NULL，也即ASCII 码为0***

再来看一个小程序：

```
#include <stdio.h>
int main() {
    int num1 = 1, num2 = 2;
    printf("%d\n\n%d", num1, num2);
    return 0;
}
```

输出结果：

```
1

2
```

我们可以发现，在`printf`中使用了两个`\n`来换行，说明在`num1`输出后要连续换行两次再输出`num2`，就得到了上面的结果。

(3) 字符串常量

字符串是由若干个字符组成的串，在 C 语言中没有单独一种基本数据类型可以存储（**C++中有`string`类型**），只能使用字符数组的方式。因此我们这里先介绍字符串常量。

上面提到，字符常量就是单个使用单引号标记的字符，那么此处的字符串常量则是由双引号标记的字符集，例如`IloveTJU`就是一个字符串常量。

字符串常量可以作为初值赋给字符数组，并使用`%s`的格式输出。

我们来看下面这个小程序：

```
#include <stdio.h>
int main() {
    char str1[25] = "I love TJU";
    char str2[25] = " so it is.";
    printf("%s, %s", str1, str2);
    return 0;
}
```

输出结果：

```
I love TJU, so it is.
```

上面的代码中，`str1[25]`和`str2[25]`均表示由 25 个`char`字符组合而成的字符集合，我们称其为字符数组。我们在`printf`中使用两个`%s`分别将它们输出。

最后指出，不能把字符串常量赋值给字符变量，因此

```
char a = 'abcd';
```

的写法是不允许的。

布尔型

布尔型在 C++中可以直接使用，但在 C 语言中必须添加 `stdbool.h` 头文件才可以使用。布尔型变量又称 `bool` 型变量，它的取值只能是 `ture`（真）或者 `false`（假），分别代表非零与零。在赋值的时候，我们可以直接使用 `true` 或 `false` 进行赋值，或是使用整型常量对其进行赋值，只不过整型常量在赋值给布尔型变量时会自动转换为 `true`（非零）或者 `false`（零）。注意“非零”是包括正整数和负整数的，即 1 和-1 都会转换为 `true`。但是对计算机来说，`true`和 `false` 在存储时分别为 1 和 0，因此如果使用`%d` 输出 `bool` 型变量的话，则 `true` 和 `false` 会输出 1 和 0。

下面来看一个例子（请将文件后缀名设为`.cpp`，否则需要添加`#include <stdbool.h>`头文件）：

```
#include <stdio.h>
int main() {
    bool flag1 = 0, flag2 = true;
    int a = 1, b = 1;
    printf("%d, %d, %d\n", flag1, flag2, a==b);
    return 0;
}
```

运行结果：

```
0, 1, 1
```

上面的代码中，我们把 `flag1` 赋值为 0，`flag2` 赋值为 `true`（也就是 1），然后将它们输出。比较有疑问的是为什么把 `a==b`（a 等于 b）当作整数输出的时候会是 1。事实上，系统会把 `a==b` 作为一个条件，判断其是否为真。显然 a 和 b 均为 1，是相等的，因此 `a==b` 为真（`true`），也即输出 1。

强制类型转换

有时候我们会需要把浮点数的小数部分切掉来变成整数，或是把整型变为浮点型来方便做除法（因为整数除以整数在计算机中视为整除操作，不会自动变为浮点数），或是其他很多情况，我们都会要用到强制类型转换，即把一种数据类型转换成另一种数据类型。

强制类型转换的格式如下：

```
(新类型名)变量名;
```

这其实很简洁，只需要把需要变成的类型用括号括着写在前面就行了。下面给出一个例子来说明：

```c
#include <stdio.h>
int main() {
    double r = 12.56;
    int a = 3, b = 5;
    printf("%d\n", (int)r);
    printf("%d\n", a / b);
    printf("%.1f", (double)a / (double)b);
    return 0;
}
```

输出结果：

```
12
0
0.6
```

这个例子使用了`(int)r`来把`r`强制转换成`int`型并输出，使用了`(double)a`把`a`转换成浮点型来做除法，输出格式中`%.1f`是指保留一位小数输出。

需要指出，如果将一个类型的变量赋值给另一个类型的变量，却没有写强制类型转换操作，那么编译器将会自动进行转换。但是这并不是说任何时候都可以不用写强制类型转换，因为如果是在计算的过程中出现需要转换类型，那么我们就不能等它算完再在赋值的时候转换。

符号常量和const常量

符号常量通俗地讲就是“替换”，即用一个标识符来替代常量，又称“宏定义”或者“宏替换”。格式如下：

```c
#define 标识符 常量
```

例如下面这个例子是把圆周率 pi 设置为 3.14：

```c
#define  pi  3.14
```

于是在程序中凡是使用`pi`的地方将在程序执行前全部自动替换为`3.14`。注意末尾不加分号。例如下面这个程序就是计算半径为3的圆的近似面积的代码：

```c
#include <cstdio>
// 将pi定义为3.14
#difine pi 3.14
int main() {
    double r = 3;
    printf("%f\n", pi * r * r);
    return 0;
}
```

输出结果：

```
28.26
```

另一种定义常量的方法是使用 `const`，格式如下：

```
const 数据类型 变量名 = 常量;
```

仍然用`pi`来举例：

```
const double pi = 3.14;
```

下面的程序我们输出一个半径为 3 的圆的近似周长：

```
#include <stdio.h>
const double pi = 3.14;
int main() {
    double r = 2;
    printf("%f\n", 2 * pi * r);
    return 0;
}
```

输出结果：

```
18.84
```

我们把这两种写法都称为常量是因为它们一旦确定其值后就无法改变，例如 pi = pi + 1的写法就是不行的。这两种方法采用哪种都随意，一般都不会出错。

注意：define 除了可以定义常量外，其实可以定义任何语句或片段。格式如下：

```
#define 标识符 任何语句或片段
```

例如我们可以写一个这样的宏定义：

```
#define ADD(a, b) ((a) + (b))
```

这样就可以直接使用 ADD(a,b)来代替 a+b 的功能：

```
#include <stdio.h>
#define ADD(a, b) ((a) + (b))
int main() {
    int num1 = 3, num2 = 5;
    printf("%d", ADD(num1, num2));
    return 0;
}
```

输出结果：

```
8
```

有同学会问，为什么要在上面加那么多括号呢？直接`#define ADD(a, b) a+b`不可以吗？或者说保险点写成`#define ADD(a, b) (a + b)`？实际上必须加那么多括号，这是因为宏定义是直接将对应的部分替换，然后才进行编译和运行。因此像下面这种程序，就会出问题：

```
#include <stdio.h>
#define CAL(x) (x * 2 + 1)
int main() {
    int a = 1;
    printf("%d\n", CAL(a + 1));
    retrun 0;
}
```

输出结果：

```
4
```

这个结果跟一些学弟妹心里想的结果可能不太一致，学弟妹可能觉得应该是 5 才对。实际上这就是宏定义的陷阱，它把替换的部分直接原封不动替换进去，导致`CAL(a + 1)`实际上是`(a + 1 * 2 + 1)`，也就是`1 + 2 + 1 = 4`，而不是`((a + 1) * 2 + 1)`。

总之，尽量不要使用宏定义来做除了定义常量以外的事情，除非给能加的地方都加满括号。

运算符

运算符就是用来计算的符号，常用的运算符有算术运算符、关系运算符、逻辑运算符、条件运算符、位运算符等。由于确实经常会用到，所以下面对每种运算符都进行解释。

算术运算符

算术运算符有很多，比较常用的是下面几个：

> (1)`+`加法运算符，将前后两个数相加。
>
> (2)`-`减法运算符，将前后两个数相减。
>
> (3)`*` 乘法运算符，将前后两个数相乘。
>
> (4)`/`除法运算符，取前面的数除以后面的数得到的商。
>
> (5)`%`取模运算符，取前面的数除以后面的数得到的余数。
>
> (6)`++`自增运算符，令一个整型变量增加 1。
>
> (7) `--`自减运算符，令一个整型变量减少 1。这些运算符都有一些细节可说，不妨都来看看。

首先，(1)(2)(3)这三个运算符没有特别大的需要注意的问题，可以直接用，举例如下：

```c
#include <stdio.h>
int main() {
    int a = 3, b = 4;
    double c = 1.23, d = 0.24;
    printf("%d %d\n", a + b, a - b);
    printf("%f\n", c * d);
    return;
 }
```

输出结果：

```
7 -1
0.295200
```

接着是除法运算符，需要注意当被除数跟除数都是整型的时候，并不会得到一个 `double` 浮点型的数，而是直接舍去小数部分（即向下取整）。例如下面这个例子：

```c
#include <stdio.h>
int main() {
    int a = 5, b = 4, c = 5, d = 6;
    printf("%d %d %d\n", a / b, a / c, a / d);
    return 0;
 }
```

输出结果：

```
1 1 0
```

可以看到，5/4 直接舍掉了小数部分变成了 1，而 5/6 则直接变成了 0。

另外，除数如果是 0，会导致程序异常退出或是得到错误输出 `1.#INF00`，因此在出现问题时请检查是否有某种情况下除数为零。

到这里为止的加减乘除四种运算符，其优先级顺序和四则运算的优先级相同。

取模运算符，返回被除数与除数相除得到的余数，例如下面这个例子：

```c
#include <stdio.h>
int main() {
    int a = 5, b = 3, c = 5;
    printf("%d %d\n", a % b, a % c);
    return 0;
}
```

输出结果：

```
2 0
```

与除法运算符一样，除数不允许为 0，因为当出问题的时候首先应该先考虑除数是否有可能为零。取模运算符的优先级和除法运算符相同。

再来讨论自增运算符。自增运算符有两种写法：`i++`或是`++i`。这两个都可以实现把 i 增加 1 的功能，但是也有不同的地方。它们的区别在于`i++`是先使用 i 再将 i 加 1，而`++i`则是先将 i 加 1 再使用 i。

看起来是不是很绕？我们来看下面的例子：

```
#include <stdio.h>
int main() {
    int a = 1, b = 1, n1, n2; n1 = a++;
    n2 = ++b;
    printf("%d %d\n", n1, a);
    printf("%d %d\n", n2, b);
}
```

输出结果：

```
1 2
2 2
```

首先我们先看`n1 = a++`这句：这里`n1`先获得`a`的值，再将`a`加 1，因此`n1`和`a`分别为 1 2；接着是`n2 = ++b`这句：先将`b`加 1，`n2`再获得`b`的值，因此`n2`和`b`分别为 2 2。

自减运算符和自增运算符一样，也有`i--`和`--i`这两种写法，作用是将`i`减 1，细节上和自增运算符相同。

关系运算符

常用关系运算符共有六种：`<`、`>`、`<=`、`>=`、`==`、`!=`，实现的功能如下表所示。

	含义	语法	返回值
<	小于	a < b	表达式成立时返回真(1, true)，不成立时返回假(0, false)
>	大于	a > b	
<=	小于等于	a <= b	
>=	大于等于	a >= b	
==	等于	a == b	
!=	不等于	a != b	

逻辑运算符

常用逻辑运算符有三个：`&&`、`||`、`!`，分别对应“与”、“或”、“非”。

	含义	语法	返回值
&&	与	a && b	ab 都真返回真 其他情况返回假
‖	或	a ‖ b	ab 都假返回假 其他情况返回真
!	非	!a	如果 a 为真，则返回假 如果 a 为假，则返回真

条件运算符

条件运算符`?:`是 C 语言中唯一的三目运算符，也就是需要三个参数的运算符，它的格式如下：

```
A ? B : C
```

其含义是：如果 A 真，那么执行并返回 B 的结果；如果 A 假，那么执行并返回 C 的结果。举一个例子来说明：

```
#include <stdio.h>
int main() {
    int a = 3, b = 5;
    int c = a > b ? 7 : 11;
    printf("%d\n", c);
    return 0;
}
```

输出结果：

```
11
```

这段代码中，由于 a > b 不成立（3 < 5），因此返回冒号后面的 11，并将 11 赋值给 c。再举一个例子：

```
#include <stdio.h>
#define MAX(a, b) ((a) > (b) ? (a) : (b))
int main() {
    int a = 4, b = 3;
    printf("%d\n", MAX(a, b));
    return 0;
}
```

输出结果：

```
4
```

上面这份代码中，我们使用宏定义来定义了`MAX(a, b)`结构，`((a) > (b) ? (a) : (b))`的意思是当`a > b`时返回 a，否则返回 b。这就实现了取两个数的较大值的功能。

位运算符

位运算符有六个，不过相对上面的几种运算符来说使用较少，学弟妹可能常用的是左移运算符。由于 int 型的上限为2^31-1，因此有时程序中无穷大的数 INF 可以设置成(1 << 31) - 1（注意必须加括号，因为位运算符的优先级没有算术运算符高）。但是一般更常用的是2^30-1，因为它可以避免相加超过 int 的情况。注意，如果把2^30-1写成二进制的形式就是`0x3fffffff`，因此下面两个式子是等价的。

```
const int INF = (1 << 30) - 1;
const int INF = 0x3fffffff;
```

	含义	语法	效果
<<	左移	a << x	整数 a 按二进制位左移 x 位
>>	右移	a >> x	整数 a 按二进制位右移 x 位
&	位与	a & b	整数 a 和 b 按二进制对齐，按位进行与运算（除了 11 得 1，其它均为 0）
|	位或	a | b	整数 a 和 b 按二进制对齐，按位进行或运算（除了 00 得 0，其它均为 1）
^	位异或	a ^ b	整数 a 和 b 按二进制对齐，按位进行异或运算（相同为 0，不同为 1）
~	位取反	~a	整数 a 的二进制的每一位进行 0 变 1、1 变 0 的操作

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1