信手相连——面向过程的程序设计 (C 语言)

1. 数据、运算与控制

崔冠宇

cuiguanyu@ruc.edu.cn

https://github.com/GuanyuCui/RUC-infohands-2020

信息学院
中国人民大学

2020-2021 学年秋季学期
(11.7, 10:00-11:30, 教二 2109)

目录

进度

上次完成了基础知识的讲解，今天开启新的内容——数据类型与控制结构。

- ✓ 基础知识（计算机结构、代码风格与规范……）；
- → 数据类型（长短整、单双精度浮点、字符……）；
- → 控制结构（顺序、选择、循环）；
- 数组；
- 函数；
- 指针；
- 自定义类型（枚举、联合体、结构体）；
- 文件；
- 提高知识（递归、一点数据结构、大作业指导）。

常量与变量——标识符

标识符（identifier）是用来命名包括常量、变量、数组等实体的有效字符序列。命名规则如下：

1. 仅能由大小写字母、阿拉伯数字以及下划线（_）[1]组成；
2. 不能以数字作为首字符；
3. 区分大小写；
4. 不能与保留关键字重复，如 `int`、`if`、`switch`。

[1]小知识：一般以双下划线开头的变量是标准库内部的变量，所以一般不用双下划线开头的变量，以免冲突。

常量与变量——常量

常量是指在运行时不可改变的量，主要有以下几类：

1. 字面量：比如 1.0f (float)，1024 (int)，'a' (char)，"C++" (const char *)。
2. 宏定义的符号常量：比如 #define PRICE 30。因为宏定义本质上是预处理阶段由预处理器扫描源代码进行替换，所以在后续使用中不能对其赋值。
3. const 关键字定义的常变量：如 const int a = 1023。此处可能难以理解，a 是变量，但是 const 关键字会提示编译器检查源代码，不允许直接显式地修改该变量的值。

```
1 const int a = 0;
2 //a = 2;  // 报错!
3 int * p = (int *)&a;
4 *p = 2;  // 允许，但不提倡!
```

常量与变量——变量

变量，与常量相反，是指在运行时可改变的量。变量的基本属性是变量类型、变量名和变量值。

定义一个变量（以及初始化）的最基本的方式如下：

```
typename variable_name[ = init_value];[2]
```

以下的方式也是有效的：

```
1 int a, b, c;  // 同时定义多个同类变量
2 char d = 'a', e = 'b', f;  // 定义时部分赋值
3 int g = 3, h = 3;  //定义时全部赋值
4 // int g = h = 3; 错误!
5 double i;
6 i = 1.0;  // 定义后初始化
```

C 语言标准要求任何变量必须先定义后使用。

[2]以后均用"[]"表示可选内容。

数据类型——整型

指存储整数值的变量，主要是 [unsigned] short、int、long、long long 等类型的变量。

默认形式是十进制，可以通过在数字前加 0 表示八进制数、加 0x 表示十六进制数[3]。除此之外，还有后缀 u/U 表示无符号字面量、l/L 表示长整型、ll/LL 表示长长整型，而且 u 可以与后两种后缀搭配。

另外，由于数据类型有字节长度限制，所以会出现称为溢出的现象。

C 可以进行类型间的隐式转换，但要注意是否会有截断等问题。

整数在内存中实际上是以二进制补码的形式存储的，而且还有大小端机器的问题，这里超纲了，有兴趣的同学请自行了解。

[3]对大家现阶段来说基本用不到，但对于后面较为深入的课程来说，还是有用的。

数据类型——浮点型

指的是存储小数值的变量，主要是 `float`、`double`、`long double` 等类型的变量。不加 f/F 后缀指明为单精度浮点型（`float`）时，编译器默认小数字面量是双精度浮点型（`double`）。

字面上有两种写法，一是普通小数形式如 12.345，二是指数写法如 1.3e-4（注意指数写法的要求：e/E 前必须有数字，后面必须是整数）。

IEEE 754 规定了计算机存储浮点值的标准，但是超出了课程的要求，所以此处不做说明，有兴趣的同学可以自行了解。

数据类型——字符型

指的是存储单个字符值的变量，主要是 char 类型的变量。

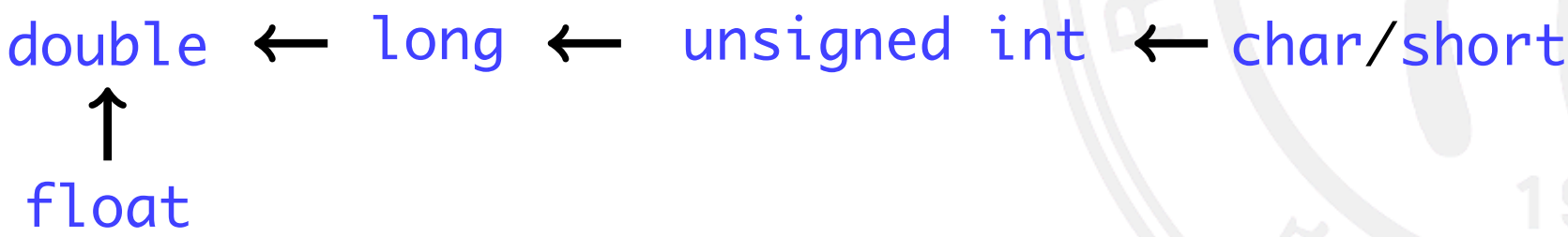
字面上书写字符变量时，为了和标识符区分，需要在字符外加一层半角单引号一如 'a'，'1'（注意 1 != '1'）等可以作为字符变量的值（字符常量），但是 '12'、"1"、"123" 都不是合法的字符变量的值。

请大家注意，"字符串常量"是指用半角双引号包裹的，形如 "Hello world!\n" 这样的常量。C 语言与 Python、C++ 不同，并没有存储字符串变量专用的数据类型，对于存储字符串的需求，需要程序员自行使用字符数组来实现。

对于一些无法直接写出的字符，如回车、TAB 等字符，则需要用转义字符表示。

混合运算

当一个表达式中有多个类型的值时，C 语言标准要求要先将所有类型的值转换成其中级别最高（一般而言，占用字节多的类型级别高）的类型后，再进行运算。下面是数据类型的级别（箭头指向级别高的类型）：



运算符与表达式

运算符有许多种类：算术（+ - * / %）、关系（> < == >= <= !=）、逻辑（&& || !）、位（& | ∧ ~ << >>）、赋值（== 及复合赋值）、条件（A:B?C）、逗号（,）、指针（* &）、字节数（sizeof）、强制类型转换（(typename)）、分量（. ->）、下标（[]）、函数调用（()）等。

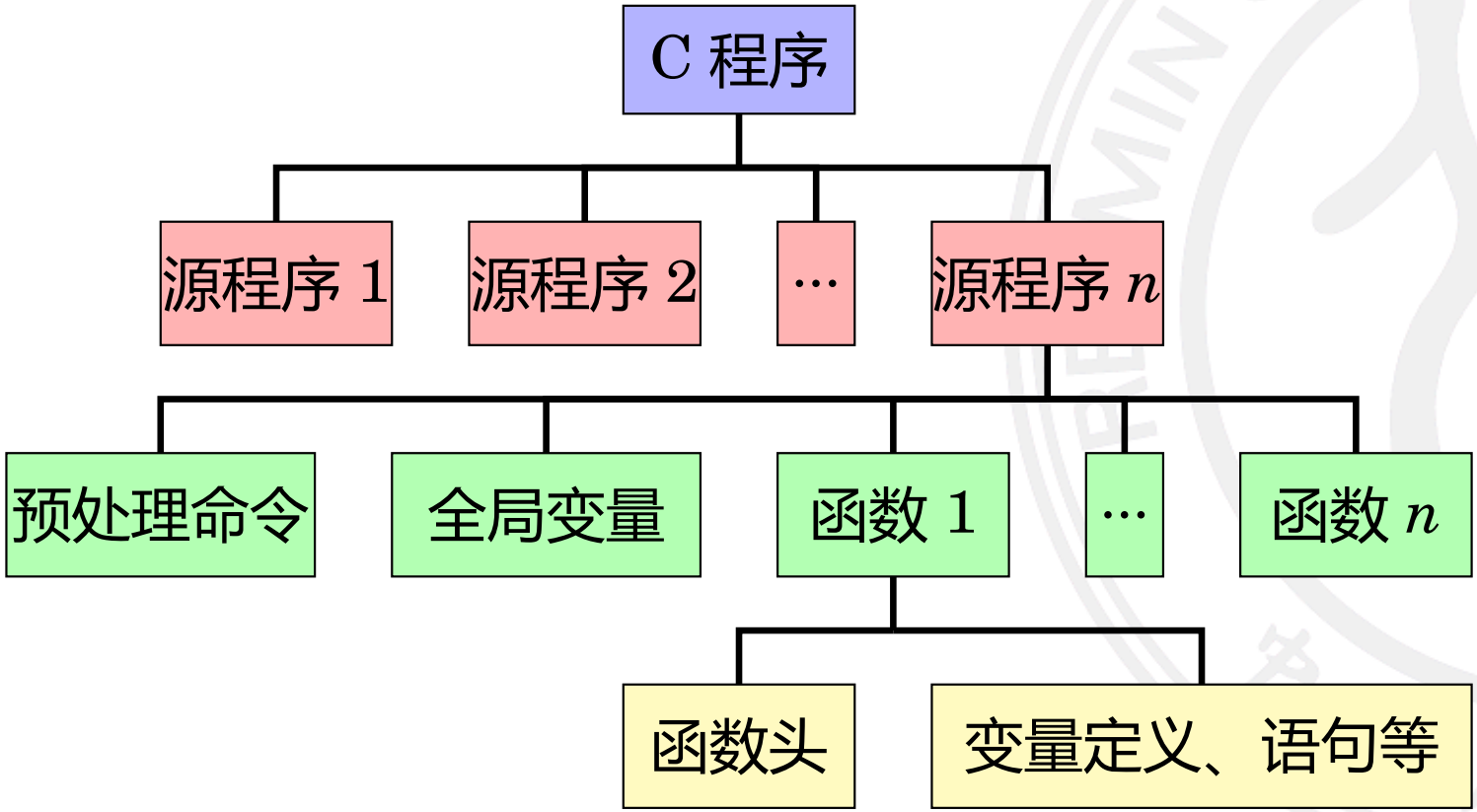
表达式求值时的总体原则是按照优先级先高后低，从左至右的顺序进行。重点关注求值时的范围，许多结果的错误与值溢出有关。除此之外还要注意类型转换，回忆类型转换的特点。

赋值表达式较为复杂，不仅有复合赋值运算符，还需要考虑左值右值（思考：变量可以作为哪一种？常量呢？表达式呢？），更进一步，还有更复杂的嵌套赋值[4]。只要紧扣赋值表达式是从右至左赋值，只有变量可以作为左值的原则即可。

[4]因为赋值表达式本身也是表达式，凡是表达式就有值，自然可以作为右值。

C 程序的构成

下面介绍一下 C 程序的基本构成：



C 语言语句

C 语言的语句一般而言以分号为分割，主要有以下几类：

1. 控制语句：用来完成某些功能，如条件（`if`）、分支（`switch`）、循环（`for`、`while`）、跳转（`goto`）、返回（`return`）、继续（`continue`）和中断（`break`）等；
2. 函数调用语句：例如 `printf(...)`；
3. 表达式语句：表达式 `+;`[5]；
4. 空语句：单独一个 `;` 叫做空语句；
5. 复合语句：用大括号括起来的一组语句称为复合语句，大括号后面不加分号。

[5]注意，表达式可以嵌套，但语句不能嵌套。例如 `(a=b)>0` 是合法的，而 `(a=b;)>0` 是不合法的。

关系运算符与关系表达式

关系运算符表示两个值之间的（大小）关系。关系运算符的优先级如下，其中同一类的优先级相同，第一类高于第二类：

1. <、<=、>、>=；
2. ==、!=。

整体来说，算术运算符高于关系运算符，关系运算符高于赋值运算符。

用关系运算符连接两个表达式形成的表达式称作关系表达式。关系表达式的值是逻辑值，当关系成立，值为 1，否则为 0。

逻辑运算符与逻辑表达式

逻辑运算符表示变量间的逻辑关系。C 语言规定了三个逻辑运算符，它们的优先级从高到低分别为：

1. ！逻辑非，相当于命题逻辑中的 $\neg$；
2. && 逻辑且，相当于命题逻辑中的 $\wedge$；
3. || 逻辑或，相当于命题逻辑中的 $\vee$。

逻辑非（！）的优先级高于算术运算符，逻辑与（&&）和逻辑或（||）的优先级低于关系运算符。

用逻辑运算符连接表达式或逻辑值形成的表达式称作逻辑表达式。自然地，逻辑表达式的值是逻辑值，真的逻辑值为 1[6]，否则为 0。

[6]在进行逻辑运算时，任何非零数都被认为是真，但结果为真的表达式的值一般为 1。

程序结构与顺序结构

大家高中数学可能学过，算法中的三种基本结构是：

1. 顺序结构；
2. 选择结构；
3. 循环结构。

C 语言执行时，默认是从上往下一条条语句执行的，所以默认的结构就是顺序结构。

下面介绍其他两种程序结构——选择结构和循环结构。

选择结构——if 语句

选择结构主要是对条件的选择与分支。最基本的（二分支）选择结构是由 if 语句实现的，主要有下面的几种形式：

1. if(表达式) 语句，当表达式成立时执行 if 后面的语句，否则不执行；
2. if(表达式) 语句 else 语句；
3. 上述两种的组合，如 if(表达式) 语句 else if(表达式) 语句。

另外有条件表达式，表达式 A ? 表达式 B : 表达式 C，当表达式 A 的值为真时，整个条件表达式的值为表达式 B 的值，否则为表达式 C 的值。

注意，当语句 if、else 后面如果跟多条操作时，需要将其用大括号括起来，成为一条复合语句。单条语句可以不加大括号，但由于 if 是就近匹配的，可能会发生问题，所以推荐全部加大括号。

为什么推荐 if 语句后总是加大括号

例题（符号函数）

请实现一个程序，输入一个整数（int），如果是正数，输出 1；如果是负数，输出 −1；如果是 0，则输出 0。

有人写出了如下代码：

```
1 y = -1;
2 if(x != 0)
3     if(x > 0) y = 1;
4 else y = 0;
```

但实际运行程序会发现结果是不正确的。这是因为，不像 Python 根据缩进确定层次，C 语言的缩进只是给程序员看的，不用大括号区分的话，else 只会和最近的 if 语句进行匹配。

选择结构——switch 语句

对于多分支的选择结构，是由 switch 语句实现的，格式如下：

```
switch (表达式)
{
    case 常量表达式1: 语句1  // 必须是常量!
    case 常量表达式2: 语句2
    //...
    case 常量表达式n: 语句n
    //[default: 语句 n+1]
}
```

switch 语句相较 if 语句复杂，以下几点提示，请大家注意：

选择结构——switch 语句

1. switch 语句执行时，首先计算括号内表达式的值，然后与诸 case 标签匹配，若有匹配则跳转执行，否则跳转至 default 之后的语句；[7]
2. case 后面接的常量表达式应该不同，否则就无法判断如何跳转了；
3. (重要！) case 相当于"标号"的功能，当执行完一个 case 后面的语句后，如果没有 break，将继续往下执行；
4. 多个 case 可以共用一组执行语句，如：

```
1 switch (expr)
2 {
3     case 1:
4     case 2: 语句
5 }
```

[7]若没有写 default，则直接跳出 switch 语句。

循环结构——if-goto 语句

循环结构在需要重复执行某些特定语句时使用，主要由 if-goto、while、do-while 以及 for 语句等实现。

下面介绍最简单（但是最不推荐使用）的 if-goto 结构。

goto 是无条件跳转[8]，与 switch 有些类似，都需要跳转至一个标签处，标签的格式是“标识符 +:”，如 LABEL_1:。

goto 与 if 共用可以达到循环的目的；goto 单独使用还可达到多层循环体跳转至循环体外。

由于使用 goto 语句容易破坏程序的结构，所以不推荐使用这种方式实现循环结构。

[8]类似于汇编中的 jmp 命令。事实上，底层都是通过跳转来实现循环的。

循环结构——while 语句与 do-while 语句

while 语句的基本格式是 while(表达式) 语句。它的功能是先计算表达式的值，当表达式的值不为 0 时，运行后面的语句并跳转至循环开头。

do-while 语句的基本格式是 do 语句 while(表达式)。它的功能是先运行语句，然后计算表达式的值，若表达式的值不为 0，则跳转至循环开头。

类似于 if，如果需要执行多条语句，则需要将它们用大括号括起来。

比较 while 语句和 do-while 语句，可以发现，当表达式初始状态为非 0 时，两种语句的结果相同；当表达式初值为 0 时，两种语句结果不同。

循环结构——for 语句

for 语句一般用于循环次数确定或指定步长迭代的情况，基本格式是 for(表达式 1; 表达式 2; 表达式 3) 语句。

for 语句的执行过程比较复杂：

1. 执行表达式 1；
2. 计算表达式 2 的值，若为真，则跳转至 3 ；若为假，转至 5 ；
3. 执行循环体语句；
4. 执行表达式 3，转 2 ；
5. 循环结束。

循环结构——for 语句

for 语句的三个表达式可以是空语句，此时有各种变式：

1. 省略表达式 1 或 3，则不会执行它们（此时初始化的工作可在 for 之前，循环变量的更新可在循环体内实现）；
2. 省略表达式 2，相当于循环条件始终为真；
3. 同时省略表达式 1、3，相当于 while 语句；
4. 将表达式 1、2、3 都省略，相当于 while(1)；

虽然说一般而言，表达式 1 承担循环变量初始化的工作，表达式 3 承担更新循环变量的工作，但是并不是严格的规定，只是约定俗成而已。

循环结构——break 语句与 continue 语句

在使用循环结构时，有时也会出现需要提前结束循环或跳过本轮循环的情况，此时就要对应使用 break 和 continue 语句。注意仔细理解两语句的区别。

比如要求将 0-100（含端点）中的奇数输出：

```
1 for(int i = 0; i <= 100; i++)
2 {
3     // 若是偶数则跳过本轮循环
4     if(i % 2 == 0)
5     continue;
6     printf("%d", i);
7 }
```

循环结构——break 语句与 continue 语句

再比如要求不使用 sqrt() 函数，将第一个平方大于 200 的正整数输出：

```
1 int i = 0;
2 while(1)
3 {
4     // 如果平方大于200则跳出循环
5     if(i * i > 200)
6     break;
7     i++;
8 }
9 printf("%d", i);
```

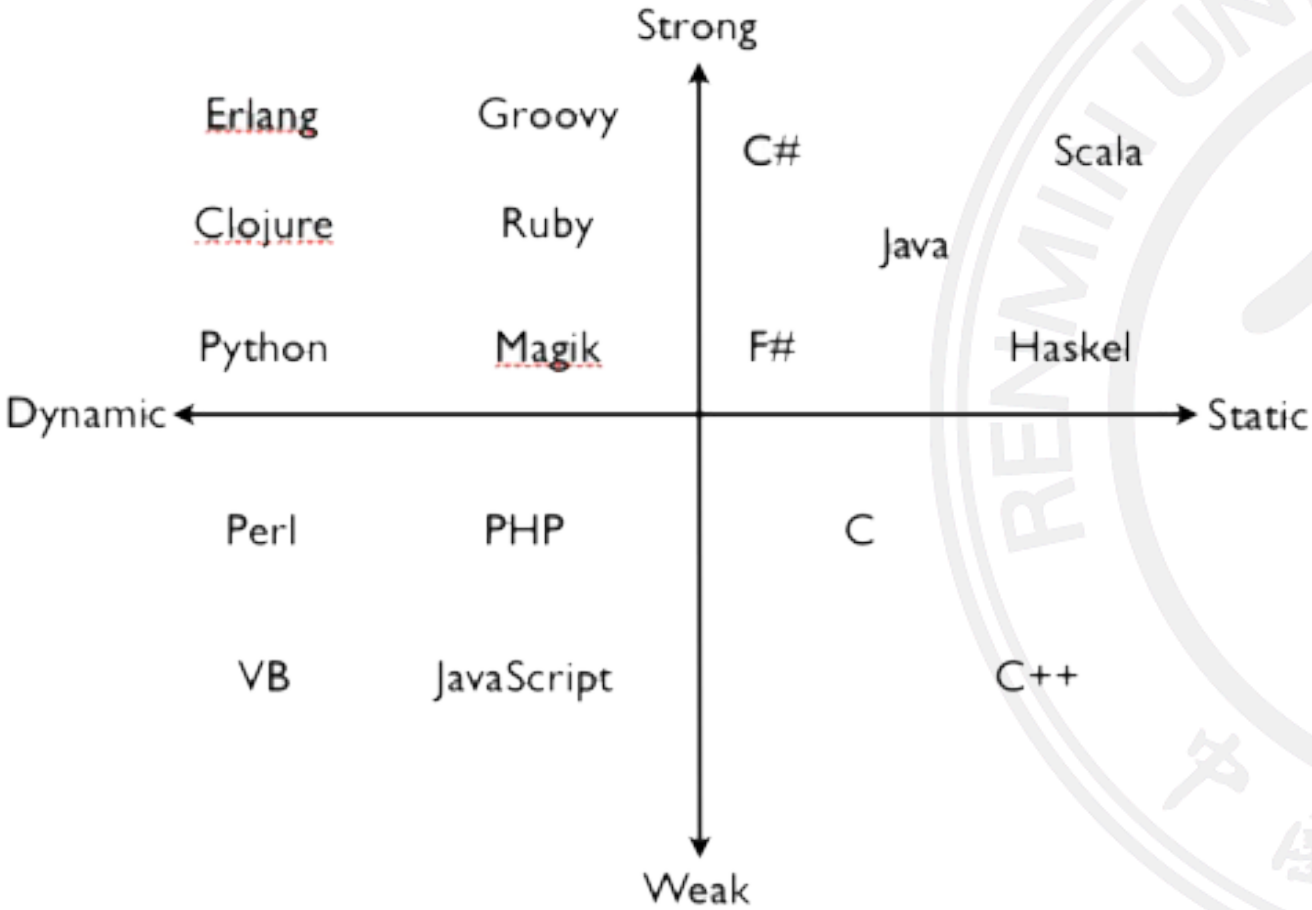
下面讲一些补充知识。

强类型、弱类型、静态类型与动态类型

这些概念并没有完全统一的标准，以下是一种理解。

- 强类型：偏向于不容忍隐式类型转换。譬如说 haskell 的 `int` 就不能变成 `double`；
- 弱类型：偏向于容忍隐式类型转换。譬如说 C 语言的 `int` 可以变成 `double`；
- 静态类型：编译的时候就知道每一个变量的类型，因为类型错误而不能做的事情是编译时语法错误。
- 动态类型：编译的时候不知道每一个变量的类型，因为类型错误而不能做的事情是运行时错误。

强类型、弱类型、静态类型与动态类型



容易让人晕头转向的 const 关键字详解

const 关键字有下列两种用法容易被混淆：

1. `const typename identifier;`
2. `typename const identifier`。

当 typename 是 int、double 等基本类型时，两种写法功能等价，如：

```
1 const int a = 0;
2 int const b = 0;  // 二者等价
3 const double pi_1 = 3.1416;
4 double const pi_2 = 3.1416;  // 二者等价
```

需要注意的是，既然 const 关键字提示编译器不允许更改变量的值，那么在初始化时就需要赋初值。即：

```
1 const int a;  // 正确，但没用
```

容易让人晕头转向的 const 关键字详解

当 typename 是 int *、char * 等指针类型[9]时，两种写法功能不等价，如：

```
1 const int * pa;  // 指针值可变，指向的值不可变
2 int const * pb;  // 同上
3 int * const pc;  // 指针值不可变，指向的值可变
4 const int * const pd;  //  二者都不可变
5 int const * const pe;  //  同上
```

简单记忆 / 理解方法：const 仅修饰后面离的最近的部分。

[9]按进度来说超前了，大家可以暂时不理解，等讲到指针时会再提。

变量的本质

物理上，变量存放于内存中（回忆上节中的内存模型），是一些占据了内存空间的二进制位。

本质上讲，汇编层面上并不区分变量的类型。底层，或者说汇编语言是一种无类型的语言，它对变量的识别甚至都没有人类可读的命名，只有变量存储的某个地址和变量所占用的长度（字节数）。编译器将 C 语言转换成汇编语言时，只保留了给变量分配的地址和变量所占的长度，变量的名字只是给人类看的。

如 movb 表示传送一个字节（byte，8 位）的数据，movw 表示传送一个字（word，2 字节，16 位）的数据，movl 表示传送一个长字 / 双字（long，4 字节，32 位）的数据，movq 表示传送一个四字（quad，8 字节，64 位）的数据。

char的本质——ASCII 码

char 类型很有意思，它用来表示我们看到的部分字符。本质上讲，char 类型就是一个占一个字节的整数而已，计算机中有专门的部分负责将这个数对应到字符显示出来，当然也可以将字符对应回数字。其中的对应规则是统一的，被称为 ASCII 码，由美国提出。

有时候 char 的数字本质也很有用，这表示它可以做数学运算，如：`int a = 'a', 'a' + 1 == 'b', '1' - '0' == 1`。在诸如“输出字母表”等应用中，可以利用这个特性。

ASCII 码表可以很方便的查询，书、搜索引擎、man ascii 命令（Linux、macOS）等方式均可。

char的本质——ASCII 码

1. man ascii (less)

The **decimal** set:

0	nul	1	soh	2	stx	3	etx	4	eot	5	enq	6	ack	7	bel
8	bs	9	ht	10	nl	11	vt	12	np	13	cr	14	so	15	si
16	dle	17	dc1	18	dc2	19	dc3	20	dc4	21	nak	22	syn	23	etb
24	can	25	em	26	sub	27	esc	28	fs	29	gs	30	rs	31	us
32	sp	33	!	34	"	35	#	36	$	37	%	38	&	39	'
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	126	~	127	del

FILES

/usr/share/misc/ascii

HISTORY

An **ascii** manual page appeared in Version 7 AT&T UNIX.

BSD June 5, 1993 BSD

(END)

C 语言字符串

如果有同学提前了解过的话，可能会知道 C++ 有专门处理字符串变量的类型 `std::string`，但是很不幸，C 语言并没有字符串类型，只能由程序员手动维护字符数组来保存“字符序列”。

有同学可能会问：比如 `printf("Hello world!")` 不是字符串嘛？答案是否定的，这是字符串常量，本质还是字符数组，只不过由于是常量，被编译器放在程序的静态区，使用的时候只用一个指针（类型是 `const char *`）来定位它。

自增自减运算符 ++ --

这里要提醒的是，++ / -- 与变量的顺序对结果有一定影响。当运算符在变量左侧的时候，表示先操作变量，再返回变量的值；当运算符在变量右侧的时候，表示先返回变量的值，然后操作变量的值。如下代码所示：

```
1 int a = 0;
2 int b = a++;
3 // a == 1, b == 0;
```

```
1 int a = 0;
2 int b = ++a;
3 // a == 1, b == 1;
```

大家可能会发现有些地方的题目乐于考察 a = b+++c; 这样的奇怪语句的计算顺序，这其实很无聊，C 语言对这样"奇怪"的表达式并没有做求值顺序规定（n1570, P76. 即未定义行为，undefined behavior，UB），所以编译器怎么实现都对。大家在平时写代码的时候注意加括号，尽量拆分复杂语句，提高可读性即可。

遗留问题——%lld 还是 %I64d?

C/C++ 标准有一个历史遗留问题——整数类型的长度并没有规定死。比如，标准规定 `int` 类型保证至少有 16 位长度，但现在的 32/64 位系统上，`int` 几乎都是 32 位[10]，这也就导致了早期一些系统上的整数类型的长度并不统一，十分混乱。

这也导致了格式控制符的混乱，比如许多部署在 Windows 上的 OJ 系统中，`long long` 的格式控制符的形式是 `"%I64d"`，而在部署在 Linux 系统上的 OJ 系统的格式控制符的形式是 `"%lld"`。不过，现在后者被选作标准。

（Windows 的一些自作主张真的是很烦人啊 -_- ）

[10] https://zh.cppreference.com/w/cpp/language/types

避免溢出的小技巧

我们知道，C 语言的数据类型由于占用内存固定，所以是有表示范围的，对于超出范围的数则不能正确表示。在特定情况下，我们可以使用一些小技巧来避免溢出。

例题（连乘积的余数）

第一行输入一个正整数 M，第二行输入一串正整数（以 0 结尾，0 不计算在内），求出它们的乘积除以 M 的余数。

遇到逻辑简单，但是又涉及数值运算的题就要特别小心了，很可能题目考察的重点并不在于控制结构和思路，重点更可能是数据的表示范围以及溢出的防止。

避免溢出的小技巧

```
1  int main()
2  {
3      long long M;
4      scanf("%lld", &M);
5      long long product = 1, input = 1;
6      while(input != 0)
7      {
8          scanf("%lld", &input);
9          if(input != 0)
10             product *= input;
11     }
12     printf("%lld", product % M);
13     return 0;
14 }
```

避免溢出的小技巧

测试数据:

```
191
108 966 58 14 34 57 18 2147483646 61 21474
    83646 59 0
```

正确的输出的结果应该是 169。但是上面的程序错了，为什么？——乘积在运算过程中溢出了。

注意到最终要求的是余数，这启发我们考虑余数的性质，而不是算出积来再求余数。下面从同余入手，补充一点数学知识：

补充数学知识——同余

定义

如果两个整数 a, b 除以 m 所得余数相同，则称它们模 m 同余（也称 a 同余于 b 模 m），记作 $a \equiv b (\text{mod} \quad m)$。

性质

以下均省略 $(\text{mod} \quad m)$

1. (自反性) $\forall a \in \mathbb{Z}, a \equiv a$;
2. (对称性) $\forall a, b \in \mathbb{Z}$, 若$a \equiv b$, 则$b \equiv a$;
3. (传递性) $\forall a, b, c \in \mathbb{Z}$, 若$a \equiv b$且$b \equiv c$, 则$a \equiv c$;
4. (保持加减) $\forall a, b, c, d \in \mathbb{Z}$, 若$a \equiv b$且$c \equiv d$, 则$(a \pm b) \equiv (c \pm d)$;
5. (保持乘法) $\forall a, b, c, d \in \mathbb{Z}$, 若$a \equiv b$且$c \equiv d$, 则$(ab) \equiv (cd)$.

补充数学知识——（二元）关系

X 与 Y 上的二元关系 R 是 $X \times Y$ 的一个子集，即我们把一些 (x, y) 收集起来就构成了一个关系。若 $(x, y) \in R$，则称它们有关系 R，记作 xRy。当 $X = Y$ 时，我们称 R 是 X 上的关系。

例: $\mathrm{id}_X = \{(x, x) | x \in X\}$ 是 X 上的恒等关系。

对于 X 上的关系 R，它们可能有下面特殊的性质：

- 自反性: $\forall x \in X((x, x) \in R)$
- 反自反性: $\forall x \in X((x, x) \notin R)$
- 对称性: $\forall x, y \in X((x, y) \in R \rightarrow (y, x) \in R)$
- 反对称性: $\forall x, y \in X((x, y) \in R \rightarrow (y, x) \notin R)$
- 传递性: $\forall x, y, z \in X(((x, y) \in R \wedge (y, z) \in R) \rightarrow (x, z) \in R)$

（你注意到了吗？函数是一种特殊的关系。）

补充数学知识——等价关系、等价类

当 X 上的二元关系 R 满足自反、对称、传递性时，称 R 是 X 上的等价关系。在不至混淆的情况下，一般把等价关系记作 $\sim$。

记 $[x] = \{y \in X | xRy\}$，称 $[x]$ 是 x 在关系 $\sim$ 下的等价类。可以证明，这些等价类是 X 的不相交子集，且并集是 X[11]。

容易验证，同余关系是等价关系。给定 m，则 $[0] = \{km | k \in \mathbb{Z}\}$，$[1] = \{km + 1 | k \in \mathbb{Z}\}$，…，$[m-1] = \{km + (m-1) | k \in \mathbb{Z}\}$，它们构成了 $\mathbb{Z}$ 的划分。

$\{[0], [1], [2], \cdots, [m-1]\}$ 一般记作 $\mathbb{Z}_m$，称作 $\mathbb{Z}$ 的模 m 剩余类。

更一般地，给定一个集合 X 和上面的一个等价关系 $\sim$，由它导出的各个等价类的集合称为 X（关于关系 $\sim$）的商集，一般记作 $X/\sim$。

[11] 给定 X 的一群子集，若它们互不相交，且并集为 X，则称它们是 X 的划分。

避免溢出的小技巧

```
1  int main()
2  {
3      long long M;
4      scanf("%lld", &M);
5      long long product = 1, input = 1;
6      while(input != 0)
7      {
8          scanf("%lld", &input);
9          if(input != 0)
10             product = (product * input) % M;
11     }
12     printf("%lld", product);
13     return 0;
14 }
```

避免溢出的小技巧

有了上面的实例讲解，你能解决下面的问题吗？

例题（自然数平方和）

已知自然数平方和公式为

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}。$$

给定 n（$1 \le n \le 2.64 \times 10^6$，保证和能用 `unsigned long long`[12] 存储），请编写程序，利用给定公式输出 $\sum_{i=1}^{n} i^2$ 的值。

[12] 范围 $0 \sim 2^{64} - 1$（约 1.84×10^{19}）。

避免溢出的小技巧

给大家十分钟时间尝试一下（坏笑）。

避免溢出的小技巧

估计和的上限为 $\frac{2.64\times10^6\times2.64\times10^6\times5.28\times10^6}{6} = 6.13\times10^{18}$，确实在 unsigned long long 的表示范围内。

有的同学可能会这么写：

```
1 // 读取 n 的值的过程略
2 unsigned long long sum = n * (n + 1) * (2 * n
      + 1) / 6;
3 // 输出和 sum 的过程略
```

但是仔细分析，在程序计算 $n\times(n+1)\times(2n+1)$ 时，中间结果约为 3.68×10^{19}，将会溢出。那这么修改行不行呢？

```
1 sum = n / 6 * (n + 1) * (2 * n + 1);
```

从纯数学的角度看，这样没问题，但是计算机并不全是数学的，n / 6 并不一定除尽，于是可能产生误差。

避免溢出的小技巧

改成 double 也是不行的，因为这个和是一个整数，用 double 表示也可能有误差。

我们可以把题干的式子改写为

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} = \boxed{\frac{n(n+1)}{2}\frac{(2n+1)}{3}}$$

将代码改写成：

```
1 sum = n * (n + 1) / 2 * (2 * n + 1) / 3;
```

注意到 n 和 $n+1$ 一定有偶数，所以第一项为整数，再乘以 $2n+1$ 时不会超过 1.84×10^{19}，所以不会溢出。

总之，对于有乘有除的情况，有可能可以通过改变乘除顺序使得中间结果不会溢出。

避免溢出的小技巧

避免溢出的方法总结：

- 换用表示范围更大的数据类型（int→long→long long，还有 unsigned 系列）。
- 利用同余的性质，每次求余数；
- 变化求值顺序，使得中间结果不溢出；
- 大杀器：实现大整数算法。

逻辑短路

逻辑短路是 C 语言里一个很有意思的性质。在逻辑且运算中，只要第一个表达式为假，不需要执行第二个表达式就可以得出整个表达式为假；类似的，在逻辑或运算中，只要第一个表达式为真，不需要执行第二个表达式就可以得出整个表达式为假。

C 语言采取了这种特性，如下面的这段代码：

```
1 int a = 0;
2 int b = 1 || (a = 2);
3 printf("%d", a);
```

运行发现输出的 a 的值为 0。

有时候也要注意这种现象，尽量不在逻辑运算中夹杂赋值等变更性语句，以免结果不符合预期。

致谢

感谢大家到场支持，也欢迎大家积极提出意见和建议。大家也可以谈谈想要在信手相连的课程中听什么内容，我将酌情做出调整。