C 语言笔记

目 录

C 语言程序基本格式

示例:

```
#include <stdio.h>

int main(void) {
    printf("Hello, World!\n");   //打印Hello World!到控制台
    return 0;
}
```

这是一个打印 Hello World!到控制台的程序，里面展示了很多语法.

main 函数

操作系统需要执行程序，就需要知晓，程序从哪里开始执行.因此，我们需要提供一个程序入口点.C 语言程序的入口点是 main 函数，程序代码使用{}进行囊括.

```
#include <stdio.h>

int main(void) {
    //程序代码 ...
}
```

> 所有符号都要采用英文符号!

程序代码

```
    printf("Hello World!\n");
```

printf 是一个函数，通过 printf("内容")调用.

句尾的;是**必须的**.

最顶上这句,

```
#include <stdio.h>
```

是引入系统库为我们提供的函数，包括 printf 在内.

注释

在写代码过程中，可以添加注释.

```
//单行注释

/*
 *多行注释
 *这是CLion自动生成的程序
 */
```

注释有两种，单行注释和多行注释.注释并不算程序的一部分，可以写任意内容.

最后的

```
return 0;
```

可以不写，编译器会自动处理.

> C99 起 main 函数末尾若缺失 return 语句，等效于 return 0;

基本数据类型

程序离不开数据.例如，我们需要保存一个数字或者是字母，就是作为数据进行保存.

不同数据的类型可能不同，例如 1 是一个整数，0.5 是一个小数，a 是一个字符.不同数据类型占据的空间也不同.

计量单位

计算机底层实际上只能表示 0 和 1.如果要储存一个十进制的数字，例如 3，需要用到 2 个 2 进制位来保存，即 11，占用两个位置；十进制的 15，二进制为 1111，占用 4 个位置，即 4 个 bit 位.

> 快速算法：$15 = 2^3 \times 1 + 2^2 \times 1 + 2^1 \times 1 + 2^0 \times 1$.
>
> 原理：二进制数的位权定义:第 i 位(最低位为 0)的权值为2^i.
>
> 因此，n+1 位 2 进制数 1000…000，等于2^n.

一般的，占用 8 个 bit 位，为一个字节(B).在大多数系统中，1 个字（word）等于 CPU 一次能处理的位数.例如在 16 位操作系统中，2 字节为一个字(16bit)；在 64 位系统中，8 字节为一个字(64bit).它们都是计量单位.

> 通用说法里“字”常被当作 CPU 位数，但 x86-64 汇编里 word 固定指 16 位，dword 32 位，qword 64 位.

我们常说，内存 1G、2G 等，这些单位进制如下:

单位	大小
1B	8bit
1KB	1024B
1MB	1024KB
1GB	1024MB
1TB	1024GB
1PB	1024TB

> 部分厂商按 1000 进制换算.

不同位数的系统下，基本数据类型的大小可能不同.本文仅记录 64 位操作系统的情况.

原码、反码和补码

在计算机中，所有数字都是用 0,1 这两个二进制数表示的.但是这样仅能表示正数.

原码

为了表示正负，我们可以让第一个 bit 位专门来保存符号.

> 例如:
>
> 现在有 4 个 bit 位供我们保存数据，则这 4 个 bit 位能保存的数据范围就是:
>
> - 最大: 0111 ⇒ $(2^2 \times 1 + 2^1 \times 1 + 2^0 \times 1)$ ⇒ 7
> - 最小: 1111 ⇒ $-(2^2 \times 1 + 2^1 \times 1 + 2^0 \times 1)$ ⇒ -7

虽然原码表示简单，但是在做加减法运算时，很麻烦.以 4bit 位为例:

1 + (-1) ⇒ 0001 + 1001

计算机难以计算.因此，我们发明了反码.

反码

正数的反码是其本身，负数的反码是在其原码的基础上，符号位不变，其余各位取反.

那么，1 ⇒ 0001，-1 ⇒ 1110.

再进行计算:

1 + (-1) ⇒ 0001 + 1110 = 1111 ⇒ -0.

直接相加，计算简化了很多.

这样 1111 ⇒ -0，0000 ⇒ +0.但是，在实数范围内，0 没有正负之分；并且，如果有最高位进位，通常还要“循环进位”加回最低位.

因此，我们发明了补码.

> 反码先于补码被发明原因：计算机取反极其简单廉价.

补码

正数的补码是其本身，负数的补码是在其原码的基础上，符号位不变，其余各位取反，最后+1（即负数的补码是在其反码的基础上+1）.

那么，1 ⇒ 0001，-1 = 1111.

再进行计算:

1 + (-1) ⇒ 0001 + 1111 = (1)0000 ⇒ +0.

此时，永远不会再出现-0.

并且，现在 4bit 位能表示的范围为：$-8 \sim +7$.

> 表示范围：
>
> - 最大：$0111 \Rightarrow +7$
> - 最小：$1000 \Rightarrow -8$
> - $-8 \sim +7$ 而非 $-7 \sim +8$ 原因：计算机需要从符号位就能判断正负.

数学原理

目标

用 n 个 bit 表示有符号整数，并且让减法可以用加法器完成.

核心工具：模 2^n

n 个 bit 能表示的无符号数范围是 $0 \sim 2^n - 1$.因此，所有运算都在模 2^n 下进行，那么减一个数就等价于加上它的加法逆元：

$$-x \equiv 2^n - x \pmod{2^n}$$

这就是补码（two's complement）的严格数学基础.

补码：直接实现模逆元

若正数 x 的 n 位二进制为 b，则 $-x$ 的补码就是 $2^n - x$ 的 n 位二进制.

把它变形：

$$2^n - x = (2^n - 1 - x) + 1 = \tilde{b} + 1$$

其中 $\tilde{b}$ 是把 b 的每一位取反.所以补码"取反加 1"的口诀正是来源于：

$$-x \equiv \tilde{x} + 1 \pmod{2^n}$$

> 补码中，$-x$ 对应的"大数"是 $2^n - x$.

反码：离模逆元还差 1

反码（one's complement）把 $-x$ 表示为：

$$\tilde{x} = 2^n - 1 - x$$

注意这不是 $-x$ 的模逆元.它在模 2^n 下真正对应的是：

$$2^n - 1 - x \equiv -x - 1 \equiv -(x + 1) \pmod{2^n}$$

也就是说，反码表示的"大数"比补码少了 1.它对应的真值不是 $-x$，而是 $-(x + 1)$.

> 反码中，$-x$ 对应的“大数”是 2^n-1-x，比补码的 2^n-x 少 1.

要做 $a-b$（a,b为正数），反码的做法是把 $-b$ 表示成 2^n-1-b，再做无符号加法：

$$a+(2^n-1-b)=(a-b)+(2^n-1)$$

- 当 $a<b$ 时，$a-b$ 为负，结果没有溢出2^n-1，计算结果是 $2^n-1-(b-a)$，正好是 $-(b-a)$ 的反码表示.结果正确.
- 当 $a>b$ 时，$a-b$ 为正，结果溢出2^n-1，最终的计算结果是 $a-b-1$，比正确值少了 1.需要把进位加回最低位（循环进位）才能得到 $a-b$.
- 当 $a=b$ 时，得到 2^n-1，即反码中的 -0.

补码则没有这个麻烦：

$$a+(2^n-b)=(a-b)+2^n$$

计算结果直接就是 $a-b$ 模 2^n 的结果.

其它情况（$a,b<0$等）同理可证得.

> 前置数学知识见附件模运算前置知识.pdf.

整数类型

整数就是不包含小数点的数据，例如 1，99，666 等数字.整数包含以下类型：

- int - 占用 4 个字节，32 个 bit 位，表示范围：$-2147483648\sim2147483647$.
- long - 占用 8 个字节，64 个 bit 位.（在 64 位 Linux/macOS 上占用 8 个字节，在 64 位 Windows 上占用 4 个字节.）
- short - 占用 2 个字节，16 个 bit 位.

一般默认使用 int.

浮点类型

浮点类一般用来保存小数.

> 称为浮点类型而非小数类型的原因：
>
> 浮点数采用类似科学计数法的形式存储，包含符号位、指数和尾数.通过指数调整尾数的小数点位置，因此小数点是“浮动”的，不像定点数那样固定.

- float - 单精度浮点，占用 4 个字节，32 个 bit 位.
- double - 双精度浮点，占用 8 个字节，64 个 bit 位.

使用浮点类型需要考虑精度问题，精度是有限的.

> 关于浮点数的更多讲解见附件浮点数补充.pdf.

字符类型

除了保存数字以外，C 语言还支持字符类型.我们的每一个字符都可以用字符类型保存.

- char - 占用一个字节，可以表示所有的 ASCII 码字符.

高四位 / 低四位		ASCII非打印控制字符											ASCII 打印字符																								
		0000					0001						0010		0011		0100		0101		0110		0111														
		0					1						2		3		4		5		6		7														
		+进制	字符	ctrl	代码	字符解释	+进制	字符	ctrl	代码	字符解释		+进制	字符	+进制	字符	+进制	字符	+进制	字符	+进制	字符	+进制	字符	ctrl												
0000	0	0	BLANK NULL	^@	NUL	空	16	▶	^P	DLE	数据链路转意		32		48	0	64	@	80	P	96	`	112	p													
0001	1	1	☺	^A	SOH	头标开始	17	◀	^Q	DC1	设备控制 1		33	!	49	1	65	A	81	Q	97	a	113	q													
0010	2	2	☻	^B	STX	正文开始	18	↕	^R	DC2	设备控制 2		34	"	50	2	66	B	82	R	98	b	114	r													
0011	3	3	♥	^C	ETX	正文结束	19	‼	^S	DC3	设备控制 3		35	#	51	3	67	C	83	S	99	c	115	s													
0100	4	4	◆	^D	EOT	传输结束	20	¶	^T	DC4	设备控制 4		36	$	52	4	68	D	84	T	100	d	116	t													
0101	5	5	♣	^E	ENQ	查询	21	§	^U	NAK	反确认		37	%	53	5	69	E	85	U	101	e	117	u													
0110	6	6	♠	^F	ACK	确认	22	■	^V	SYN	同步空闲		38	&	54	6	70	F	86	V	102	f	118	v													
0111	7	7	•	^G	BEL	震铃	23	↨	^W	ETB	传输块结束		39	'	55	7	71	G	87	W	103	g	119	w													
1000	8	8	◘	^H	BS	退格	24	↑	^X	CAN	取消		40	(	56	8	72	H	88	X	104	h	120	x													
1001	9	9	○	^I	TAB	水平制表符	25	↓	^Y	EM	媒体结束		41	)	57	9	73	I	89	Y	105	i	121	y													
1010	A	10	◙	^J	LF	换行/新行	26	→	^Z	SUB	替换		42	*	58	:	74	J	90	Z	106	j	122	z													
1011	B	11	♂	^K	VT	竖直制表符	27	←	^[	ESC	转意		43	+	59	;	75	K	91	[	107	k	123	{													
1100	C	12	♀	^L	FF	换页/新页	28	∟	^\	FS	文件分隔符		44	,	60	<	76	L	92	\	108	l	124	|												
1101	D	13	♪	^M	CR	回车	29	↔	^]	GS	组分隔符		45	-	61	=	77	M	93	]	109	m	125	}													
1110	E	14	♫	^N	SO	移出	30	▲	^6	RS	记录分隔符		46	.	62	>	78	N	94	^	110	n	126	~													
1111	F	15	☼	^O	SI	移入	31	▼	^-	US	单元分隔符		47	/	63	?	79	O	95	_	111	o	127	Δ	^Back space												

编码表中包含了所有我们常见的字符，包括运算符号、数字、大小写字母等.

> 某些无法显示的字符需要使用转义字符来表示，如换行用\n 表示.

变量

前面我们了解了 C 语言中的基本数据类型，那么如何使用呢?

我们可以创建不同类型的变量.

变量的使用

我们可以直接声明一个变量，并利用这些变量进行基本的运算.

```
数据类型 变量名称 = 初始值; //初始值可以不用在定义变量时设定.
//  = 是赋值，可以将等号后面的值赋给前面的变量，
//右边可以写一个数字（常量）、变量名称、算式.
```

例如，现在我们想要声明一个整数类型的变量:

```
int a = 10; //变量类型为int，名称为a，初始值为10.
```

```
int a = 10, b = 20;
//多个变量可以另起一行，也可以像这样逗号隔开，注意类型必须一样.
```

命名变量时遵循以下规则:

- 在同一作用域内，不能重复使用其它变量已经使用过的名字.
- 只能包含英文字母、下划线和数字，并且严格区分大小写，例如 a 和 A 不算同一个变量.
- 虽然可以包含数字，但是不能以数字开头.
- 不能是关键字（包括上面提到的所有数据类型，还有一些关键字会在后面认识）.
- （建议）使用英文单词而不是拼音，多个词使用驼峰命名法或者通过_相连接.

初始值可以是一个常量数据（如 10，0.5 这样的数字），或是运算表达式的结果，还可以是其他变量，这样会将其它变量的值作为初始值.

我们可以使用变量进行一些基本的运算:

```
#include <stdio.h>

int main(void) {
    int a = 10;
    int b = 20;
    int c = a + b;
}
```

此处使用了+运算符，与数学中的加法运算作用相同，会将左右两边的变量值加起来，得到结果，我们可以将运算结果作为其它变量的初始值.

那么，让我们验证一下:

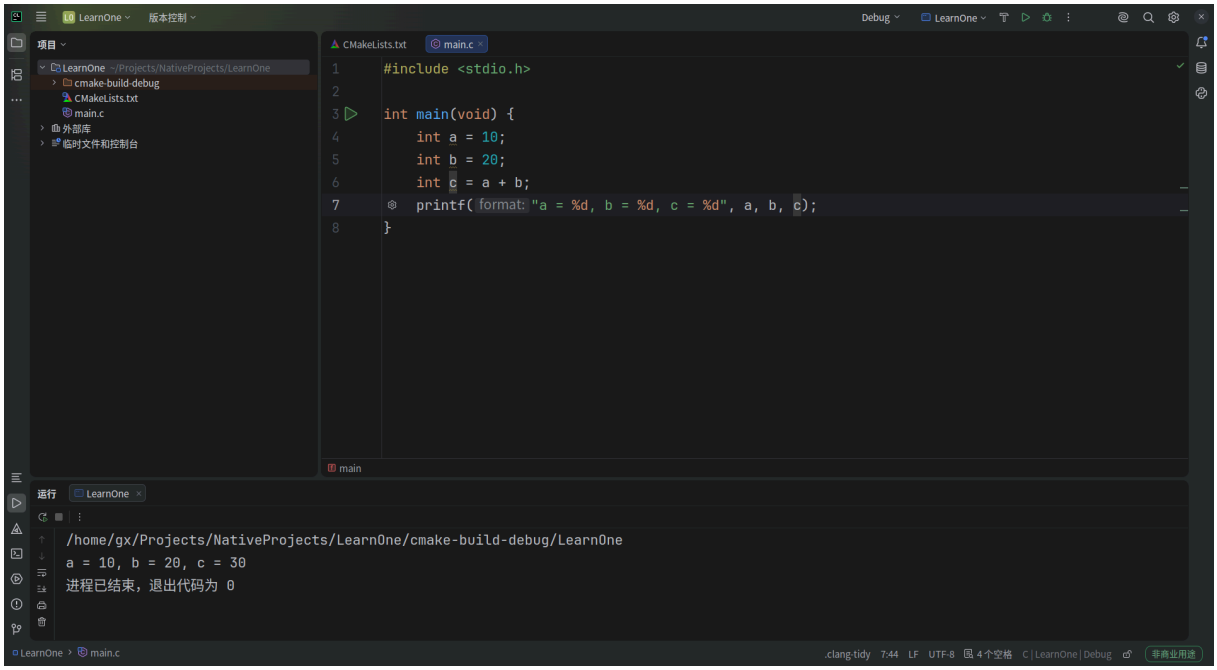
```
#include <stdio.h>

int main(void) {
    int a = 10;
    int b = 20;
    int c = a + b;
    printf(c);
}
```

但是我们发现这样写是错误的.实际上，printf 的第一个参数必须是格式字符串，不能把变量直接作为第一个参数传入.

```
printf("c的结果是：%d", c);
//使用%d来代表一个整数类型的数据（占位符），在打印时会自动将c的值替换上去.
```

然后，我们会看到成功打印了 c 的结果是：30.

当然，我们也可以一次性打印多个，只需要填写多个占位符即可：



除了使用 %d 打印有符号整数之外，还有：

格式控制符	说明
%c	输出一个单一的字符
%hd、%d、%ld	以十进制、有符号的形式输出 short、int、long 类型的整数
%hu、%u、%lu	以十进制、无符号的形式输出 short、int、long 类型的整数
%ho、%o、%lo	以八进制、不带前缀、无符号的形式输出 short、int、long 类型的整数
%#ho、%#o、%#lo	以八进制、带前缀、无符号的形式输出 short、int、long 类型的整数
%hx、%x、%lx %hX、%X、%lX	以十六进制、不带前缀、无符号的形式输出 short、int、long 类型的整数.如果 x 小写，那么输出的十六进制数字也小写；如果 X 大写，那么输出的十六进制数字也大写.
%#hx、%#x、%#lx %#hX、%#X、%#lX	以十六进制、带前缀、无符号的形式输出 short、int、long 类型的整数.如果 x 小写，那么输出的十六进制数字和前缀都小写；如果 X 大写，那么输出的十六进制数字和前缀都大写.
%f、%lf	以十进制的形式输出 float、double 类型的小数
%e、%le %E、%lE	以指数的形式输出 float、double 类型的小数.如果 e 小写，那么输出结果中的 e 也小写；如果 E 大写，那么输出结果中的 E 也大写.

%g、%lg %G、%lG	以十进制和指数中较短的形式输出 float、double 类型的小数，并且小数部分的最后不会添加多余的 0.如果 g 小写，那么当以指数形式输出时 e 也小写；如果 G 大写，那么当以指数形式输出时 E 也大写.
%s	输出一个字符串

同样的，如果我们要进行小数的运算并打印结果：

```
#include <stdio.h>

int main(void) {
    double a = 2.5;
    double b = 3.5;
    double c = a * b;
    printf("a = %lf, b = %lf, c = %lf\n", a, b, c);
    printf("c = %.1lf", c);
}
```

则会在控制台打印：

```
a = 2.500000, b = 3.500000, c = 8.750000
c = 8.8
进程已结束，退出代码为 0
```

字符同样可以打印：

```
#include <stdio.h>

int main(void) {
    char a = 'A';
    printf("变量a是%c, 对应ASCII码是%d", a, a);
}
```

将会打印结果：

```
变量a是A, 对应ASCII码是65
进程已结束，退出代码为 0
```

如果不对变量设初始值呢?

```
#include <stdio.h>

int main(void) {
    int a, b;
    printf("变量a是%d, b是%d", a, b);
}
```

我们发现，会打印出：

```
变量a是32767，b是458529904
进程已结束，退出代码为 0
```

可以看到，如果不设初始值，那么它的值并不确定，而不是默认为 0.

至于为什么不是 0 也并不确定，会在后续内存分配篇章中讲解.

有时候，我们希望将一些一成不变的值用于运算，例如π.我们可以使用不可变的变量，称其为常量.

定义常量和定义变量较为相似，区别在于定义常量时，需要在前面添加一个 const 关键字，表示这是一个常量.

```c
#include <stdio.h>

int main(void) {
    const double pi = 3.14;
}
```

尝试 pi = 1，就可以看到，常量在设定后不允许修改.

无符号数

我们知道，所有数据底层都是采用二进制进行保存，而第一位则是用于保存符号位.如果不考虑符号位， 那么所有数据都是按照正数来进行表示.

例如，char 类型：

- 考虑符号位表示范围：$-128 \sim 127$.
- 不考虑符号位表示范围：$0 \sim 255$.

我们可以直接使用这些不带符号位的数据类型.

```c
#include <stdio.h>

int main(void) {
    unsigned char c = -65;
    printf("%u", c);
}
```

由于使用了无符号数，控制台会打印出：

```
191
进程已结束，退出代码为 0
```

这是由于-65 的补码表示为 10111111.无符号位情况下，10111111 表示 191.

我们也可以直接以无符号数形式打印.

```c
#include <stdio.h>

int main(void) {
    int i = -1;
    printf("%u", i);
}
```

此时控制台将会输出无符号 int 的最大值.

```
4294967295
进程已结束，退出代码为 0
```