C 语言入门 (new)

前言

学习方法

1. coding and parsing
2. 滚动式学习，通过后续知识的学习复习巩固前面的知识。
3. 通过实践（编程）学习掌握 C 标准，而不是通过学习 C 标准来指导实践（编程）

1 编程基础

1.1 number system - 数字系统

decimal system - 十进制
0, 1, 2, 3, 4, 5, 6, 7, 8, 9

1111 = 1 * 10^3 + 1 * 10^2 + 1 * 10^1 + 1 * 10^0

binary system - 二进制
0, 1

1111 = 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0 = 8 + 4 + 2 + 1 = 15

2^0 = 1
2^1 = 2
2^2 = 4
2^3 = 8
2^4 = 16
2^5 = 32
2^6 = 64
123 = 64 + 59 = 2^6 + 32 + 27 = 2^6 + 2^5 + 16 + 11 = 2^6 + 2^5 + 2^4 + 8 + 2 + 1
= 2^6 + 2^5 + 2^4 + 2^3 + 2^1 + 2^0
= 111 1011

hexadecimal system
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f

octal system - 八进制
0, 1, 2, 3, 4, 5, 6, 7

十进制到二进制转换
将 123 转换为二进制。

123 ÷ 2	= 61 + 1⁄2	⇒ d0 = 1
61 ÷ 2	= 30 + 1⁄2	⇒ d1 = 1
30 ÷ 2	= 15 + 0⁄2	⇒ d2 = 0
15 ÷ 2	= 7 + 1⁄2	⇒ d3 = 1
7 ÷ 2	= 3 + 1⁄2	⇒ d4 = 1
3 ÷ 2	= 1 + 1⁄2	⇒ d5 = 1
1 ÷ 2	= 0 + 1⁄2	⇒ d6 = 1
0 ÷ 2	= 0 + 0⁄2	⇒ d7 = 0

123(10) = d7d6d5d4d3d2d1d0
= 0111 1011
= 0x7b (16)
= 0173 (8) (01 111 011)

38 (10) = 0010 0110 (2) = 0x26 = 046

326 (10) = 0001 0100 0110 = 0x146
= 101 000 110 = 0506

decimal	binary	hex	octal
0	0000	0x0	00
1	0001	0x1	01
2	0010	0x2	02
3	0011	0x3	03
4	0100	0x4	04
5	0101	0x5	05
6	0110	0x6	06
7	0111	0x7	07
8	1000	0x8	010
9	1001	0x9	011
10	1010	0xa	012
11	1011	0xb	013
12	1100	0xc	014
13	1101	0xd	015
14	1110	0xe	016
15	1111	0xf	017

N位二进制最小值：　0
N位二进制最大值：
S = 2^0 + 2^1 + 2^2 + … + 2^(n-1)　　(1)
2S = 2^1 + 2^2 +...2^(n-1) + 2^n　　(2)
　　(2) - (1) -> S = 2^n - 2^0 = 2^n - 1

1.2 数据存储格式

数十亿个两态开关。每个开关始终处于一种状态或另一种状态，并一直保持在该状态，直到控制单元改变其状态或关闭电源。
一个控制单元，可以：
　　检测每个开关的状态。
　　更改该开关的状态。

计算机的内存 https://www.youtube.com/watch?v=EmMI-1FH4XQ

位和位组
数据位数：bit(1), byte(8), word(16), dword(32), quadword(64)

下表显示了数据类型的的大小和范围：

size	size	unsigned range	signed range
bit(1-bit)	2^1	0 to 1	
byte(8 bits)	2^8	0 to 255	-128 to +127
word(16 bits)	2^16	0 to 65,535	−32,768 to +32,767
double-words (32 bits)	2^32	0 to 4,294,967,295	−2^31 to 2^31 - 1
quadword (64 bits)	2^64	0 to 2^64 - 1	-2^63 to 2^63 - 1

补码
下面介绍如何找到负值（非正值）的二进制补码表示。
取一个数的补码：
1. 取一个数的反码（取反）
2. 加1（二进制）

相同的过程用于将十进制值编码为二进制补码以及从二进制补码返回到十进制。以下部分提供了一些示例。

字节示例

例如，要查找字节大小（8 位），-9 和 -12 的二进制补码表示。

9 (8+1) = 0000 1001　　12 (8+4) = 0000 1100

Step 1 1111 0110　　Step 1: 1111 0011

Step 2 1111 0111　　1111 0100

-9 (10) = F7 (16)　　-12 (10) = F4 (16)

请注意，必须指定给定大小（本示例中的字节 ）的所有位。

单字长示例

要查找字长（16 位），-18 和 -40 的二进制补码表示。

18 (16+2) = 0000 0000 0001 0010　　40 (32+8) = 0000 0000 0010 1000

Step 1 1111 1111 1110 1101　　Step 1 1111 1111 1101 0111

Step 2 1111 1111 1110 1110　　Step 2 1111 1111 1101 1000

-18 (10) = 0xFFEE (16)　　-40 (10) = 0xFFD8 (16)

无符号和有符号加法

如前所述，无符号和有符号表示可以为所表示的最终值提供不同的解释。 但是，加法和减法运算是相同的。例如：

241　1111 0001　　-15　1111 0001

+ 7　0000 0111　　+ 7　0000 0111

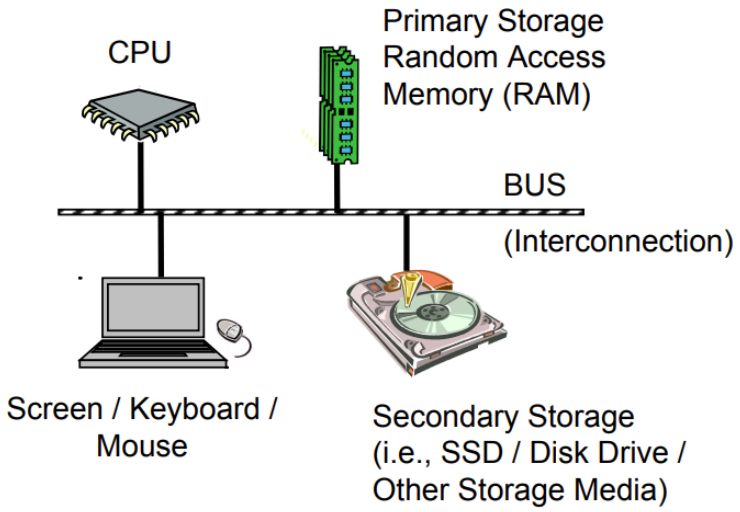
248　1111 1000　　-8　1111 1000

248 = F8　　-8 = F8

0xF8 的最终结果对于无符号表示可以解释为 248，对于有符号表示可以解释为 -8。

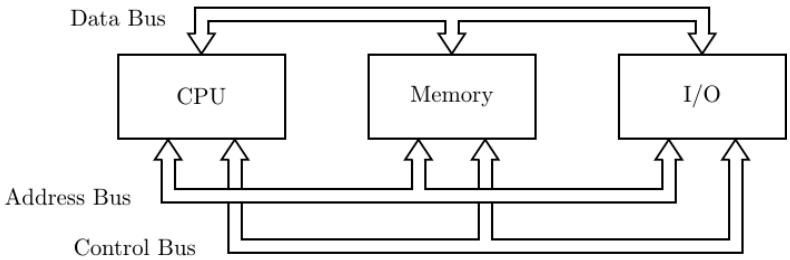
因此，为正在执行的操作明确定义数据的大小（字节、半字、字等）和类型（有符号、无符号）非常重要。

1.3 基础架构



计算机的基本组件包括中央处理单元 (CPU)、主存储器或随机存取存储器 (RAM)、辅助存储器、输入/输出设备（例如，屏幕、键盘、鼠标）以及称为总线的互连 .
该架构通常被称为冯诺依曼架构，由数学家和物理学家约翰冯诺依曼于 1945 年描述。
程序和数据通常存储在二级存储（例如，磁盘驱动器或固态驱动器）上。执行程序时，必须将其从辅助存储器复制到主存储器或 (RAM) 中。CPU 从主存储器或 RAM 执行程序。
主存储器也称为易失性存储器，因为当断电时，信息不会保留，因此会丢失。二级存储被称为非易失性存储器，因为信息在断电时会保留。

计算机硬件由三个独立的子系统组成：



输入/输出 (Input/Output)
与外部世界和大容量存储设备（例如磁盘）进行通信。键盘，监视器，称为字符终端(terminal)，硬盘。ASCII编码，字符表示。

内存 (memory)
为 CPU 的指令及其操作的数据提供存储。
内存由一系列固定大小的单元格(8个开关，8bit，只能表示非负二进制整数)所形成的线性数组组成，每个单元格都有一个唯一的地址并通过其进行访问。单元格的访问同小区的信箱访问类似。
Ascii code: ‘0’ - 48, ‘9’ - 57; ‘a’ - 97, ‘b’ - 98, ‘z’ - 122; ‘A’ - 65, ‘B’ - 66, ‘Z’ - 90
寻址方式（直接地址，偏移量，索引）。（C语言的本质）

中央处理器 (Central Processing Unit)
控制计算机的大部分活动，执行算术和逻辑运算，并包含少量非常快的内存（register）。ALU，通用寄存器，%RIP，%RBP，%RSP。

cpu与内存交互 (独占内存)，代码 (指令) 执行。指令同数据分开，指令在低位，数据在高位。指令是动作，数据是用二进制表示的对象 (整数，小数，字符，复杂数据等)。cpu在内存中读取指令 (%RIP) 并执行相应的操作。

1.4 操作系统

文件管理
非可执行文件 - 文本文件 - .c c 语言源文件 - 由 程序员 编辑生成。
可执行文件 - windows: .exe, linux 设置文件可执行位。- 由 编译器 生成。

文本文件:字符原生编码构成的二进制计算机文件，与富文本相比，其不包含字样样式的控制元素，能够被最简单的文本编辑器直接读取。

ASCII字符的文本文件可以在Unix、Macintosh、Microsoft Windows、DOS和其它操作系统之间自由交互。

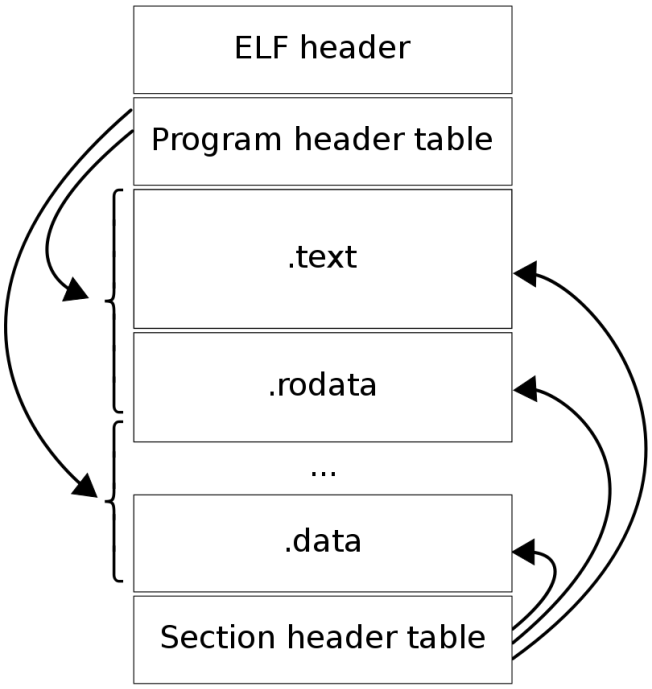
ASCII是由美国国家标准学会（American National Standard Institute，ANSI）制定的，使用标准的单字节字符编码方案，用于基于文本的数据。
0 - 9 (48 - 57), A - Z (65 - 90), a - z (97 - 122)

9 - 0 = 9　　9 (offset) + 1 = 10
57 - 48 = 9　　9 + 1 = 10

A - Z => 26　　90 - 65 +1 = 26
a - Z => 26　　122 - 97 + 1 = 26

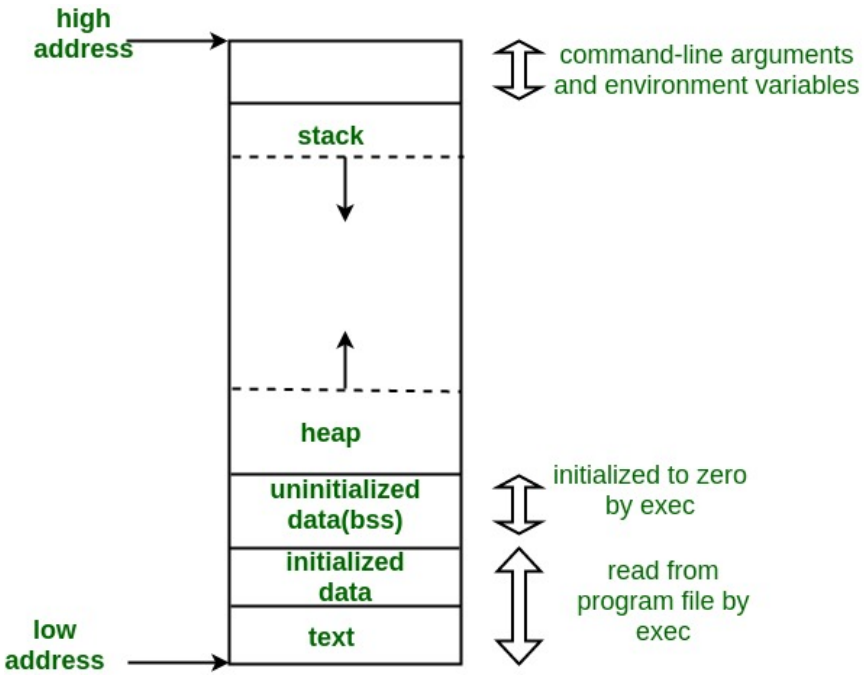
在 C 语言中用 'a' 单引号表示字符，双引号 "a", "abc" 表示一串字符 - 字符串。

elf 可执行文件格式
elf, 可执行与可链接格式 (Executable and Linkable Format), unix/linux的主要可执行文件格式。



进程管理
进程 - 可执行文件的运行实例。
进程的地址空间 (虚拟)。
1. 文本段（即指令）(text)

2. 数据段 (data)
3. 栈 (stack)



text - 文本段：
文本段，是目标文件或内存中的程序段之一，其中包含可执行指令。
通常，文本段是可共享的，因此对于频繁执行的程序，例如文本编辑器、C 编译器、shell 等，只需要在内存中保留一个副本。
文本段通常是只读的，以防止程序意外修改其指令。

data - 数据段
通常简称为Data Segment。数据段是程序虚拟地址空间的一部分，它包含由程序员初始化的全局变量和静态变量。
请注意，数据段不是只读的，因为变量的值可以在运行时更改。

stack - 栈
堆栈区域传统上与堆区域相邻并朝相反的方向增长；当栈指针遇到堆指针时，空闲内存被耗尽。

1.5 开发环境

editor - vscode
compiler - gcc (mingw32 for windows)

vscode 安装：

1. VSCode官网下载对应操作系统版本的安装包；

2. 安装，勾选适当的安装选项; (添加到PATH, 创建桌面快捷方式)

MinGW 编译器安装:

1. sourceforge的mingw项目上下载64位的编译器;
2. 安装选择 Architecture: x86-64, Threads: posix, Exception: seh;
3. 配置环境变量: 右键开始菜单 -> 系统 -> 右面板高级系统设置 -> 最下面环境变量 -> 用户/系统变量 -> 双击path -> 新建 MINGW\bin，重启系统生效。

安装C/C++扩展:
打开 vscode , 搜索安装 vscode c/c++ IntelliSense 扩展.

在线编译器 gdb online

2 hello world

2.1 hello.c

vi/code 编辑器编写第一个程序 "hello world":

```
#include <stdio.h>
int main()
{
	printf("Hello World");
	return 0;
}
```

让我们逐行分析程序。

第 1 行：[#include <stdio.h>] 在 C 程序中，所有以#开头的行都由预处理器处理，预处理器是编译器调用的程序。在一个非常基本的术语中，预处理器接受一个 C 程序并生成另一个 C 程序。生成的程序没有以# 开头的行，所有这些行都由预处理器处理。在上面的例子中，预处理器将 stdio.h 的预处理代码复制到我们的文件中。.h 文件在 C 中称为头文件。这些头文件通常包含函数声明。我们需要 stdio.h 用于程序中使用的函数 printf()。

第 2 行 [int main()] 程序必须有一个开始执行的入口点，从那里开始执行已编译的 C 程序。在 C 中，执行通常从 main() 的第一行开始。写在括号中的空白表示main函数不带任何参数。main() 也可以写成带参数。我们将在以后进行介绍。int 写在 main 之前，表示 main() 的返回类型。main 返回的值表示程序终止的状态。

第 3 和 6 行：[{ 和 }] 在 C 语言中，一对大括号定义了作用域，主要用于函数和控制语句，如 if、else、循环。所有函数都必须以大括号开头和结尾。

第 4 行 [printf("Hello World");] printf()是一个标准库函数，用于在标准输出上打印一些东西。printf 末尾的分号表示语句终止。在 C 中，分号总是用来表示语句的结束。

第 5 行 [return 0;] return 语句从 main() 返回值。操作系统可以使用返回值来了解程序的终止状态。值 0 通常表示成功终止。

如何执行上面的程序：

为了执行上面的程序，我们需要有一个编译器来编译和运行我们的程序。有某些在线编译器，可用于在不安装编译器的情况下启动 C。Windows：有许多免费的编译器可用于编译 C 程序，例如VSC，Code Blocks和Dev-CPP。Linux：对于 Linux，gcc与 Linux 捆绑在一起，VSC等也可以与 Linux 一起使用。

```
$ gcc -o hello hello.c
$ ./hello
Hello World
$
```

2.2 C程序的编译

在带有 gcc 编译器的 Ubuntu 机器上使用的步骤:

```
$ vi filename.c                    /* editor      */
$ gcc filename.c –o filename       /* compiler    */
$ ./filename                       /* execute     */
```

1. 预处理 Pre-processing
2. 编译 Compilation
3. 汇编 Assembly
4. 链接 Linking

3 数据类型，变量，运算符，表达式和语句

data type, variable, expression and statement

在学习编程语言时，您必须先了解变量、如何定义和存储它们（数据类型）、如何执行逻辑和数学运算（运算符）等，然后再了解其他任何编程概念。这些主题可以被认为是学习 C 编程技能的基本必需品。

3.1 数据类型 - data type

C 语言只提供了下列几种基本数据类型，每种数据类型需要不同数量的内存。

1. char 整型/字符型，存储单个字符。1 byte, -128 ~ 127, -2^7 ~ 2^7-1。
2. short (int) 短整型，2bytes. -32768 ~ 32767, -2^15 ~ 2^15-1。
3. int 整型，4bytes，-2^31 ~ 2^31-1。
4. long (int) 长整型，8bytes, -2^63 ~ 2^63-1。
5. float 单精度带浮数，4bytes。(有符号)

6. double 双精度浮点数，8bytes。(有符号)

无符号整型：
unsigned char unsigned short (int) unsigned int unsigned long (int

可以使用 sizeof 运算符来检查变量/数据类型的大小（bytes).

3.2 变量 - variable

变量: 分配了一些内存的存储位置。变量是人类可读的名称，它引用内存中的某些数据。变量是存储在内存中某个地址的某些数据的名称。

内存视为字节数组。数据存储在该"阵列" 中。如果一个数字大于单个字节，则它存储在多个字节中。内存的每个字节都可以通过它的索引来引用。这种对内存的索引也称为地址、位置或指针。

当在 C 中有一个变量时，该变量的值在某个地址(或某个位置)的内存中。但是通过数字地址来引用一个值是很痛苦的，所以我们为它命名，这就是变量。

变量名规则:
1. 首字符 [_a-zA-Z]
2. 由字母，数字和下划线组成。[_a-zA-Z0-9]*
3. 无冲突

变量属性:
1. 地址 - address 存贮在那里。
2. 字节数 - sizeof 占用多少内存 bytes。
3. 数据类型 - datatype integer? char, short, int; floating? float, double

3.3 语句 - statement

简单语句以分号；结尾，通常代表一个最小翻译单元，也是我们分析代码的主要观察点。

```
stmt;
```

符合语句/块语句 - component/block statement: 一条或多条语句组合在一起形成符合语句：

```
{
    stmt1;
    stmt2;
    …
}
```

注意: 块语句本身也是一条语句。

3.4 小结

变量声明/定义 - declaration and definition

```
int i;              # 4 bytes, integer
int j;

unsigned char c;        # 1 byte, char/int

float f;            # 4 bytes, floating

int i, j;
```

example:

```
#include <stdio.h>
int main()
{
        int     i;
        float   j;

        printf("Hello World");
        return 0;
}
```

我们已经声明了几个变量。我们还没有使用它们，而且它们都未初始化。一个保存整数，另一个保存浮点数。

未初始化的变量具有不确定的值。它们必须被初始化，否则你必须假设它们包含一些无意义的数字。

```
#include <stdio.h>
int main()
{
        int     i;
        float   f;

        i = 5;
        f = 2.6;
        printf("Hello World %d %.2f", i, f);
        return 0;
}
```

output:
Hello World 5 2.60

格式化输出转换函数 - printf

```
#include <stdio.h>
int printf(char *fmt, …);
```

% 转换说明符及其含义是:

1. d, i　　int 参数转换为有符号十进制表示法。
2. o, u, x　　unsigned int 参数转换为无符号八进制 (o)、无符号十进制 (u) 或无符号十六进制 (x 和 X) 。
3. f　　double 参数被四舍五入并转换为十进制。
4. c　　int 参数被转换到一个无符号字符。
5. %　　写入"%"。没有参数被转换。
6. s　　char * 参数是指向字符类型数组的指针（指向字符串的指针）。

变量初始化 - initialization

声明变量的同时给变量赋初值。

```
#include <stdio.h>
int main()
{
        int     i = 3, j = 5, k;            /* '=': initialization */
        float   f = 2.6;

        k = 10;                             /* '=': assignment */
        printf("i = %d\nj = %d\nk = %d\nf = %.2f\n", i, j, k, f);
        return 0;
}
```

字符 '\n' 换行符 newline, 键盘"enter"键, '\' 转义字符。

3.5 运算符和表达式 - operator and expression

'=' 赋值 - assignment

```
double pi;
int    count;

pi = 3.1415926;
```

count = 10;

注意:变量赋值之前必须先声明/定义。赋值表达式具有值并可以用在普通表达式中。

右结合: 相同优先级运算符的结合性。
y = x = 10; ⇔ y = (x = 10);

算术 **- arithmetic (+, -, *, /, %)**
3 + 4; 5 * 8;

i = 3;
i = 3 + 4 * 5;
i = i + 8;
i = i - 3;
i = i * 2;
i = i / 5;
i = i % 8;

i += 8;
i *= 2;
i /= 5;
i %= 8;
i *= 3 + 2; ⇔ i = i * (3 + 2);
i += j; ⇔ i = i + j;

优先级:(高 -> 低)	结合性
*, /, %	left
+, -	left
=	right

op= 赋值运算符
大多数二元运算符(即有左、右两个操作数的运算符, 比如+)都有一个相应的赋值运算符 op=, 其中, op 可以是下面这些运算符之一:

+ - * / % << >> & ^ | -> +=, -=, *=, /=, <<=, >>=, &=, ^=, |=

如果 expr1 和 expr2 是表达式, 那么

expr1 op= expr2 ⇔ expr1 = (expr1) op (expr2)

i += 6; vs i = i + 6;
除了简洁外, 赋值运算符还有一个优点:表示方式与人们的思维习惯比较接近。我们通常会说"把 6 加到 i 上"或"把 i 增加 6", 而不会说"取 i 的值, 加上 6, 再把结果放回到 i 中", 因此, 表达式 i += 6 比 i = i +6 更自然.

++, -- 运算符 **- unary**

prefix			postfix		
++i;	⇔	(i += 1);	i++;	⇔	(i +=1, i - 1);
--i;	⇔	(i -= 1;)	i--;	⇔	(i -= 1, i + 1);

优先级:(高 -> 低)	结合性
++, --	right
*, /, %	left
+, -	left
=, op=	right

example:

```
#include <stdio.h>
int main()
{
   int i, j;

   i = 10;
   j = 5 + i++;
   printf("post increment:\ni = %d\tj = %d\n", i, j);

   i = 10;
   j = 5 + ++i;
   printf("pre increment:\ni = %d\tj = %d\n", i, j);

   return 0;
}
```

output:

```
post increment:
i = 11  j = 15
pre increment:
i = 11  j = 16
```

, 运算符 **- comma**

由逗号分隔的两个表达式的求值次序为从左到右，并且左表达式的值被丢弃。右操作数的类型和值就是结果的类型和值。在开始计算右操作数以前，将完成左操作数涉及到的副作用的计算。

优先级:(高 -> 低)	结合性
++, --	right
*, /, %	left
+, -	left

	=, op=	right
	,	left

example:

```
#include <stdio.h>
int main()
{
        int     i, j;

        j = (i = 1, 2, 3);
        printf("i = %d\n", i);
        printf("j = %d\n", j);
        return 0;
}
```

```
f(a, (t = 3, t + 2), c);    =>      f(a, 5, c);
f(a, (t = 3, 6), c);        =>      f(a, 6, c);
```

==, != 相等运算符 - equality

x == y	1 if x equals y, otherwise 0.
x != y	1 if x does not equal y, otherwise 0.

```
#include <stdio.h>
int main()
{
        int x, y;

        x = y = 10;
        printf ("x: %d, y: %d, x==y ? %d\n", x, y, x == y);
        printf ("x: %d, y: %d, x!=y ? %d\n", x, y, x != y);

        y = 20;
        printf ("x: %d, y: %d, x==y ? %d\n", x, y, x == y);
        printf ("x: %d, y: %d, x!=y ? %d\n", x, y, x != y);
        return 0;
}
```

>, >=, <, <= 关系运算符 - relational

x > y	1 if x greater than y, otherwise 0.
x >= y	1 if x greater or equal y, otherwise 0.
x < y	1 if x is less than y, otherwise 0.
x <= y	1 if x less or equal y, otherwise 0.

优先级:(高 -> 低)	结合性
++, --	right
*, /, %	left
+, -	left
>, >=, <, <=	left
==, !=	left
=, op=	right
,	left

&&, || 逻辑运算符 **- logical**

x && y	1 if x != 0 and y != 0, otherwise 0.
x || y	1 if x != 0 or y != 0, otherwise 0.

逻辑运算符将任何非零操作数视为 1。

```
#include <stdio.h>
int main()
{
   int x, y;
   double d1, d2;

   x = -1;
   y = 10;
   printf ("x: %d, y: %d, x && y ? %d\n", x, y, x && y);
   printf ("x: %d, y: %d, x || y ? %d\n", x, y, x || y);
   printf("\n");

   x = 0;
   printf ("x: %d, y: %d, x && y ? %d\n", x, y, x && y);
   printf ("x: %d, y: %d, x || y ? %d\n", x, y, x || y);
   printf("\n");

   d1 = 0.217;
   d2 = 0;
   printf("d1: %.2f, d2: %.2f, d1 && d2? %d\n", d1, d2, d1 && d2);
   printf("d1: %.2f, d2: %.2f, d1 || d2? %d\n", d1, d2, d1 || d2);
   return 0;
}
```

逻辑运算符&&与||有一些较为特殊的属性，由&&与||连接的表达式按从左到右的顺序进行求值，并且，在知道结果值为真或假后立即停止计算。绝大多数 C 语言程序运用了这些属性。

优先级:(高 -> 低)	结合性
++, --	right
*, /, %	left
+, -	left
>, >=, <, <=	left
==, !=	left
&&	left
||	left
=, op=	right
,	left

x > 0 && x < 5 => 0 < x < 5
x > 0 && x != 3
x > 0 && x > y || x < 0 && x != -1

!, & 一元运算符 - unary

!x 1 if x == 0, otherwise 0. ((Logical NOT)
&x address of x.

优先级:(高 -> 低)	结合性
++, --, !, &	right
*, /, %	left
+, -	left
>, >=, <, <=	left
==, !=	left
&&	left
||	left
=, op=	right
,	

```
#include <stdio.h>
int scanf(char *fmt, …);

example:
#include <stdio.h>

int main()
{
	int i;

	printf("Enter Integer: ");
	scanf("%d", &i);
```

```
        printf("i = %d\tand !i = %d", i, !i);
        return 0;
}
```

小结

	operator	associativity
	(), [], ->, .	left
sizeof, +, -, ~,	++, --, !, &, *	right
	*, /, %	left
	+, -	left
	<<, >>	left
	>, >=, <, <=	left
	==, !=	left
	&	left
	^	left
	|	left
	&&	left
	||	left
	?:	right
	=, op=	right
	,	left

编译器解析最长有效结合性。

---x; (wrong)　　- --x;　　-++x;　　- ++x;

y += x;　　y + = x; (wrong);

op 优先级：

1. 算术运算符为基线；
2. 一元运算符最高，赋值运算符最低；
3. 关系运算符在中间；
4. 逻辑或 (||) 分两边。
5. 一元赋值右结合。

注意：一元右结合意味着后缀运算符具有更高的优先级。

```
#include <stdio.h>

int main()
{
        int x, y;

        x = 1;
        y = 2;
        y += x += 3;
```

```
	printf("x = %d\ty = %d\n", x, y);
	return 0;
}
```

3.6 编程练习

数据类型大小

```
#include <stdio.h>
#define PI  3.1415926

int main()
{
	printf("integer type size:\n");
	printf("char: %2lu\tshort: %2lu\tint: %2lu\tlong: %2lu\n",
		sizeof(char), sizeof(short), sizeof(int), sizeof(long));

	printf("\nfloating type size:\n");
	printf("float:%2lu\tdouble:%2lu\n", sizeof(float), sizeof(double));
	return 0;
}
```

两整数之和

```
#include <stdio.h>

int main()
{
	int	i1, i2, sum;

	printf("Enter two Integer: ");
	scanf("%d %d", &i1, &i2);

	sum = i1 + i2;
	printf("sum is: %d", sum);
	return 0;
}
```

圆的面积

```
#include <stdio.h>
#define PI  3.1415926
```

```
int main()
{
        double          r, s;

        printf("Enter radius: ");
        scanf("%lf", &r);

        s = PI * r * r;
        printf("The area of circle: %lf", s);
        return 0;
}
```

4 控制流 - flow control

普通语句以；结尾，包括声明 (declaration)语句和表达式 (expression) 语句。
复合/块 (component/block) 语句，以 ‘{‘ 开头，以 ‘}’ 结尾，最后不需要分号。块语句中包括一条或多条声明语句和表达式语句。

一条语句 statement 既可以是普通语句（以；结尾），也可以是块语句 ({ }).

example:

```
{
        int     x, y;

        x = 3;
        y = x + 2;
}
```

4.1 if-else

if-else 语句用于条件判断，语法格式为

```
if (expr)
        statement1
else                    /* optional */
        statement2
```

注意：statement 或者为；结尾的普通语句，或者为块语句。

先计算表达式的值，如果其值为真（即表达式的值为非 0），则执行语句 1；如果其值为假（即表达式的值为 0），并且该语句包含 else 部分，则执行语句 2。

example:

```
#include <stdio.h>

int main()
{
	int x;

	printf("input x: ");
	scanf("%d", &x);

	if (x < 0)
		x = -x;
	printf("x = %d\n", x);
	return 0;
}
```

```
#include <stdio.h>

int main()
{
	int x1, x2, max;

	printf("input x1 and x2: ");
	scanf("%d %d", &x1, &x2);

	if (x1 > x2)
		max = x1;
	else
		max = x2;
	printf("x1 = %d\tx2 = %d\n", x1, x2);
	printf("The max value is: %d\n", max);
	return 0;
}
```

注意：

```
if (expr)	⇔	if (expr != 0)
if (0.2)	⇔	if (0.2 != 0)
```

example:

```
if (i && j)	⇔	if (i != 0 && j != 0)
```

```
if (i && j) { i++; j++ }    equal to
if (i && j)
        { i++; j++}        equal to
if (i && j)
{
        i++;
        j++;
}
if (i && j) {
        i++;
        j++;
}
```

```
#include <stdio.h>

int main()
{
        int       i, j;

        printf("input integer i and j: ");
        scanf("%d %d", &i, &j);

        if (i > 0 && j > 0) {                /* not equal i && j */
                i--;
                j--;
        } else {
                i++;
                j++;
        }
        printf("i = %d\tj = %d\n", i, j);
        return 0;
}
```

4.2 else-if

语法格式

```
if (expr)
        statement
else if (expr)
        statement
else if (expr)
        statement
```

```
else
        statement

if (i > 0)
        …
else if (i == 0)
        …
else                    /* optional, i < 0 */
        …
```

example:

```
#include <stdio.h>

int main()
{
   int marks;

   printf("enter you marks: ");
   scanf("%d", &marks);
   if (marks > 85 && marks <= 100)
      printf("scored grade A\n");
   else if (marks >= 60)
      printf("scored grade B");
   else if (marks > 40)
      printf("scored grade C");
   else
      printf("scored grade C");
   return 0;
}
```

4.3 while loop and for loop

while 循环语法格式为：

```
while (expr)
        statement
```

首先求表达式的值。如果其值非 0，则执行语句，并再次求该表达式的值。这一循环过程一直进行下去，直到该表达式的值为 0 为止，随后继续执行语句后面的部分。

for 循环语法格式为

```
for (expr1; expr2; expr3)
        statement
```

等价于

```
expr1;
while(expr2) {
        statement
        expr3;
}
```

在给定条件为真时重复一条语句或块语句。在执行循环体之前测试条件。

example:

```
#include <stdio.h>

int main()
{
        int i;

        i = 5;
        while (i > 0) {
        printf("%d: hello, world\n", i);
        i--;
        }
        return 0;
}
```

i	i>0 ?	print	i--
5	y	y	y
4	y	y	y
3	y	y	y
2	y	y	y
1	y	y	y
0	N	-	-

当 i = n时

loop n times; while(expr) n+1 times

```
#include <stdio.h>

int main()
{
        int i;

        for (i = 0; i < 5; i++)
                printf("%d: hello, world\n", i);
```

```
        return 0;
}
```

i=0	i<5	printf	i++	i
y	y	y	y	1
-	y	y	y	2
-	y	y	y	3
-	y	y	y	4
-	y	y	y	5
-	N	-	-	

当 i < n 时

expr1 1 time; expr2 n+1 times; expr3 n times; loop n times.

4.4 编程练习

两个库函数：fprintf, exit

```
#include <stdio.h>
int fprintf(FILE *stream, char *fmt, ...);
```

指向文件流的指针 stream: stdin, stdout, stderr

fprintf(stdin, fmt, …); ⇔ printf(fmt, …);

stderr: 标准错误输出；unix shell 经常将标准输出 (stdout) 重定向到文件或通过管道导向另一程序的输入，导致输出在终端不可见。我们通常希望错误输出到终端可见而不是消失在管道中。

the Sum of Natural Numbers

sum = 1 + 2 + …+ n

```
#include <stdlib.h>
void exit(int status);
```

exit() 函数会导致正常的进程终止，并且 status 返回调用者。

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

int main()
{
        int     n, i, sum;
```

```
	printf("enter a positive integer: ");
	scanf("%d", &n);
	if (n <= 0) {
		fprintf(stderr, "error, n must greater than 0\n");
		exit(1);
	}
	sum = 0;
	for (i = 1; i <= n; i++)
		sum += i;
	printf("sum = %d\n", sum);
	return 0;
}
```

reverse integer

```
#include <stdio.h>

int main()
{
	int	n, rev = 0, rem;	/* rev: reversed, rem: remainder */

	printf("Enter an integer: ");
	scanf("%d", &n);
	while (n != 0) {
		rem = n % 10;
		rev = rev * 10 + rem;
		n /= 10;
	}
	printf("Reversed number = %d", rev);
	return 0;
}
```

n	n != 0	k
n	y	1
n/10	y	2
n/10^2	y	3
…	y	…
n/10^(k-1)	y	k
n/10^k	N	-

n/10^k = 0 => 1 < n/10^(k-1) < 10 => n/10 < 10^(k-1) < n

lg(n/10) < k-1 < lgn => lgn-1 < k-1 < lgn

lgn < k < lgn + 1　　=>　　k = lgn + 1　　(向下取整）

palindrome

回文数（也称为数字回文）是一个数字（例如 16461），当其数字反转时保持不变。

```
#include <stdio.h>

int main()
{
   int n, rev = 0, rem, orig;        /* reversed, remainder, original */

   printf("Enter an integer: ");
   scanf("%d", &n);
   orig = n;
   while (n != 0) {
         rem = n % 10;
         rev = rev * 10 + rem;
         n /= 10;
   }
   if (orig == rev)
         printf("%d is a palindrome.", orig);
   else
         printf("%d is not a palindrome.", orig);

   return 0;
}
```

perfect number

完全数，又称完美数或完备数，是一些特殊的自然数：它所有的真因子（即除了自身以外的约数）的和，恰好等于它本身。
第一个完全数是6，它有约数1、2、3、6，除去它本身6外，其余3个数相加，恰好等于本身。
第二个完全数是28，它有约数1、2、4、7、14、28，除去它本身28外，其余5个数相加，也恰好等于本身。
后面的数是496、8128。

```
#include<stdio.h>

int main()
{
         int       num, rem, sum = 0, i;

         printf("Enter a number: ");
```

```
	scanf("%d", &num);

	for(i = 1; i < num; i++) {  /* find all divisors and add them  */
		rem = num % i;
		if (rem == 0)
			sum += i;
	}
	if (sum == num)
		printf(" %d is a Perfect Number\n", num);
	else
		printf("%d is not a Perfect Number\n", num);
	return 0;
}
```

print stars

```
#include <stdio.h>

int main()
{
	int i, j, rows;

	printf("Enter the number of rows: ");
	scanf("%d", &rows);
	for (i = 1; i <= rows; ++i) {
		for (j = 1; j <= i; ++j)
			printf("* ");
		printf("\n");
	}
	return 0;
}
```

i	j loop	j <= i
1	1	1 + 1
2	2	2 + 1
3	3	3 + 1
…	…	…
n	n	n + 1

total j loop = 1 + 2 + 3 + … + n = n(n+1)/2

5 函数 - function

5.1 函数概念

函数是执行特定任务或计算的代码块 ({ })。
函数接受一组输入、执行特定计算并产生输出(返回值)。

C 函数包含三个方面:

1. 函数声明　　函数原型, 告知编译器函数名称、函数参数列表和返回类型。
2. 函数定义　　函数体 (由 { } 界定) 包含要执行的实际语句。
3. 函数调用　　可以从程序中的任何位置调用函数。必须传递与函数声明一致的参数。

函数的一般形式为 :

```
return_type function_name( parameter list )
{
  body of the function
}
```

example:

```
#include <stdio.h>                      /* fun decl: int printf(char *fmt, …);    */

int max(int x, int y)                   /* function declaration and definition */
{
        if (x > y)
                return x;
        else
                return y;
}
int main()
{
        int     a = 10, b = 20;

        int m = max(a, b);              /* function call */
        printf("m is: %d\n", m);
        return 0;
}
```

?: 三元运算符 - 条件表达式

```
expr1 ? expr2 : expr3
```

首先计算 expr1, 如果其值不等于 0(为真), 则计算 expr2 的值, 并以该值作为条件 表达式的值, 否则计算 expr3 的值, 并以该值作为条件表达式的值。expr2 与 expr3 中只能有一个表达式被计算。

```
#include <stdio.h>

int max(int, int);                              /* function declaration */
int main()
{
        int     a = 10, b = 20;

        printf("max is: %d\n", max(a, b));      /* function call */
        return 0;
}
int max(int x, int y)                           /* function definition */
{
        return (x > y) ? x : y;
}
```

函数没有返回值或没有参数的函数声明

```
void fun(void);         !=      void fun();     /* old decl */
```

example:

```
#include <stdio.h>

int evenodd(int);       /* declaration */

int main()
{
        int     n, flag;

        printf("enter a number: ");
        scanf("%d", &n);

        printf("number %d is ", n);
        evenodd(n) ? printf("even\n") : printf("odd\n");
        return 0;
}

/* evenodd: check whether a number is even or odd */
int evenodd(int n)
{
        if (n % 2 == 0)
                return 1;
        return 0;
}
```

5.2 参数传递

寄存器："%rdi", "%rsi", "%rdx", "%rcx", "%r8", "%r9"，进行参数传递。

5.3 递归函数 - recursion

example: 自然数之和

```
#include <stdio.h>

int sum(int n)
{
        int i, s;

        s = 0;
        for (i = 1; i <= n; i++)
                s += i;
        return s;
}

int main()
{
        int n, s;

        printf("Enter number: ");
        scanf("%d", &n);
        s = sum(n);
        printf("sum of 1 to %d: %d\n", n, s);
        return 0;
}
```

对于 sum(n) = 1 + 2 + 3 + … + (n-1) + n　　(n >= 1)

sum(n) = sum(n-1) + n　　(n >= 1)
sum(1) = sum(0) + 1　　=>　　sum(0) = 0;　　(定义终止条件）

```
int sum(int n)                          /* T(n) */
{
        if (n == 0)
                return 0;
        return sum(n-1) + n;            /* T(n-1) */
```

```
}
```

T(n) = T(n-1) + 2 (n = 0; T(0) = 1)
T(n-1) = T(n-2) + 2
T(n-2) = T(n-3) + 2
… …
T(1) = T(0) + 2

=> T(n) = T(0) + 2 * n => T(n) = 2 * n + 1 => O(n)
function call (space) O(n)

example: 正序和倒序打印 n 个自然数

```
#include <stdio.h>

void prtd(int n)
{
	for (int i = 1; i <= n; i++)
		printf("%d\t", i);
	printf("\n");
}

void rprtd(int n)		/* T(n) = O(n), fun call O(n)	*/
{
	if (n == 0)
		return;
	printf("%d\t", n);
	rprtd(n - 1);
}

int main()
{
   int n;

   printf("Enter number: ");
   scanf("%d", &n);

   prtd(n);
   rprtd(n);
   return 0;
}
```

example: factorial - 阶乘

fac(n) = 1 * 2 * 3 * …* (n-1) * n

fac(0) = fac(1) = 1

fac(n) = fac(n - 1) * n;
fac(1) = fac(0) * 1;
fac(0) = 1;

```
#include <stdio.h>

int fac(int n)                          /* T(n) = O(n), fun call O(n)    */
{
        if (n == 0)
                return 1;
         return fac(n -1) * n;
}
int main()
{
        int       n;

        printf("Enter number: ");
        scanf("%d", &n);
        printf("%d", fac(n));
        return 0;
}
```

6 指针与数组 - pointer and array

6.1 指针 - pointer

指针是一种保存变量地址的变量。指针本身的数据类型是 (unsigned long)，声明指针需要表明指针指向的变量类型。

打印变量地址：

```
#include <stdio.h>
int main()
{
        int x;

        printf("%p", &x);
        return 0;
}
```

声明指针与赋值：

```
#include <stdio.h>
int main()
{
	int x = 10;
	int *ptr;

	ptr = &x;
	printf("%d\n", *ptr);
	*ptr = 20;
	printf("%d\n", x);
	return 0;
}
```

int *ptr;　　表达式 *ptr 的结果是 int 类型。(ptr is a pointer to int)
ptr = &x;　　*ptr 同 x 的使用效果相同。
x = x + 1;　　⇔　　*ptr = *ptr + 1; (ptr 指向的对象 + 1)
x++;　　⇔　　(*ptr)++;
++x;　　⇔　　++*ptr;

指针本身也是变量，可以进行赋值：

```
int *p, *q;
p = &x;
q = &y;
```

p = NULL;　　p 的值 (address) 为 0，常用作判断是否是一个有效指针。
p = q;　　p 指向 y。

example: swap

```
#include <stdio.h>

void swap(int x, int y)      /* wrong */
{
	int temp;

	temp = x;
	x = y;
	y = temp;
}
int main()
{
	int a, b;

	a = 10;
	b = 20;
```

```
	swap(a, b);
	printf("a = %d\tb = %d\n", a, b);
	return 0;
}

swap - 正确版本
#include <stdio.h>

void swap(int *px, int *py)
{
	int temp;

	temp = *px;
	*px = *py;
	*py = temp;
}
int main()
{
	int a, b;

	a = 10;
	b = 20;
	swap(&a, &b);
	printf("a = %d\tb = %d\n", a, b);
	return 0;
}
```

6.2 数组 - array

数组是存储在连续内存位置的相同数据类型的数据项的集合，可以使用数组的索引随机访问元素。

数组声明和初始化

可以通过多种方式声明数组。可以通过指定它的类型和大小，通过初始化它或两者来完成。

```
int arr[10];

int n = 10;
int arr[n];

int arr[] = { 10, 20, 30, 40 };		/* initialization */
```

```
int arr[4] = {10, 20, 30, 40};              /* initialization */

int arr[6] = { 10, 20, 30, 40 }             =>      int arr[] = {10, 20, 30, 40, 0, 0};
```

数组访问

使用整数索引访问数组元素。
数组索引从 0 开始，一直到数组大小减 1。数组元素与变量等同。

```
#include <stdio.h>

int main()
{
        int       arr[5], x, y = 18;

        arr[0] = 5;
        arr[2] = -10;
        arr[3 / 2] = 2;                     // this is same as arr[1] = 2
        arr[3] = arr[0];

        x = arr[3];
        arr[4] = y;

        printf("%d %d %d %d", arr[0], arr[1], arr[2], arr[3]);
        return 0;
}
```

数组索引没有越界检查。

```
#include <stdio.h>

int main()
{
   int arr[2];

   arr[0] = 12;
   arr[1] = 5;
   printf("%d ", arr[3]);
   printf("%d ", arr[-2]);
   return 0;
}
```

数组具有地址和大小两个属性。其地址为首元素（索引0）地址，大小为数组所有元素大小之和（sizeof运算符，编译时属性）。

```
#include <stdio.h>
int main()
{
	int arr[3], i;

	printf("int size: %lu\n\n", sizeof(int));
	printf("size of arr: %lu\n", sizeof(arr));
	printf("number of elems: %lu\n", sizeof arr / sizeof arr[0]);
	printf("arr address: %p\n\n", arr);
	for (i = 0; i < 5; i++)                    /* overflow */
		printf("address of arr[%d] is %p\n", i, &arr[i]);
	return 0;
}
```

数组的遍历 - 打印数组

```
#include <stdio.h>

int main()
{
	int n, arr[] = {1, 2, 3, 4};

	n = sizeof(arr) / sizeof(arr[0]);
	for (int i = 0; i < n; i++)
		printf("%d\t", arr[i]);
	return 0;
}
```

6.3 指针与数组

在 C 语言中，指针和数组之间的关系十分密切，指针与数组适合放在一起讨论。

```
int a[5], *p, x;
p = a;
```

注意：在表达式中，数组名称自动转换为指向数组首地址的指针。

```
p = a;          ⇔    p = &a[0];
a[0]            ⇔    p[0]          ⇔    *p/*a          ⇔    *(p + 0)/*(a+0)
a[i]            ⇔    p[i]          ⇔    *(p + i)
x = a[i]        ⇔    x = p[i]      ⇔    x = *(p + i)
p = &a[i]       ⇔    p = a + i
```

指针算术运算

“指针加 1”就意味着，p+1 指向 p 所指向的对象的下一个对象。相应地，p+i 指向 p 所指向的对象之后的第 i 个对象。

如果 p, q 是指向相同数据类型的指针，则有
p++, p--, ++p, --p => address of p +/- sizeof(*p)
p += index p -= index offset = p - q
if (p == NULL), if (p != NULL), if (p), if(!p), if (p == q), if (p != q)

指针是变量，而数组名不是变量，因此，类似于 a = p 和 a++ 形式的语句是非法的。

6.4 编程练习

printarray - 打印数组

```
#include <stdio.h>
#define NELEMS(a) sizeof(a)/sizeof(a[0])

void printarray(int a[], int n)
{
	int i;

	for (i = 0; i < n; i++)
		printf("%3d%c", a[i], (i%5 == 4 || i == n-1) ? '\n' : ' ');
}
int main()
{
	int arr[] = {1, 2, 3, 4, 5, 6};

	printarray(arr, NELEMS(arr));
	return 0;
}

void printarray(int a[], int n)
{
	int *p;

	for (p = a; p < &a[n]; p++)
		printf("%d\t", *p);
}
```

findelem - 查找数组元素

```
#include <stdio.h>

int findelem(int key, int a[], int n)
{
	int i;

	for (i = 0; i < n; i++)
		if (a[i] == key)
			return i;
	return -1;			/* not found */
}

int main()
{
	int idx, arr[5] = { 10, 30, 15, 40, 20 };

	idx = findelem(15, arr, 5);
	if (idx != -1)
		printf("key 15 found at index: %d\n", idx);
	else
		printf("key 15 does not found\n");
	return 0;
}
```

maxofa - 数组的最大值

```
#include <stdio.h>
#define NELEMS(a) sizeof(a)/sizeof(a[0])

int maxofa(int a[], int n)
{
	int i, max;

	for (max = a[0], i = 0; i < n; i++)
		if (a[i] > max)
			max = a[i];
	return max;
}
int main()
{
	int arr[] = {12, 2, 8, 7, 24, 11};

	printf("max of array is: %d\n", maxofa(arr, NELEMS(arr)));
```

```
	return 0;
}
```

average - 数组的平均值

```
#include <stdio.h>
#define NELEMS(a) sizeof(a)/sizeof(a[0])

double average(int a[], int n)
{
	int i;
	double sum;

	for (sum = 0, i = 0; i < n; i++)
		sum += a[i];
	return sum/n;
}
int main()
{
	int arr[] = {88, 76, 92, 83, 80, 95};

	printf("average of array is: %.2f\n", average(arr, NELEMS(arr)));
	return 0;
}
```

6.4.4 select sort - 选择排序

```
#include <stdio.h>
#define NELEMS(a) sizeof(a)/sizeof(a[0])

void swap(int *, int *);
void printarray(int *, int);

void sort(int v[], int n)
{
	int i, j;

	for (; n > 1; n--) {
		j = 0;
		for (i = 0; i < n; i++)
			if (v[i] > v[j])
				j = i;
		swap(v+n-1, v+j);
	}
}
```

```
int main()
{
	int arr[] = {88, 76, 92, 83, 80, 95, 68, 70};
	int n = NELEMS(arr);

	sort(arr, n);
	printarray(arr, n);
	return 0;
}

void swap (int *x, int *y)
{
	int t = *x;
	*x = *y;
	*y = t;

}
void printarray(int a[], int n)
{
	int i;

	for (i = 0; i < n; i++)
		printf("%3d%c", a[i], (i%10 == 9 || i == n-1) ? '\n' : ' ');
}
```

time complexity O(n^2), space complexity O(n)

insert sort - 插入排序

```
void isort(int *a, int n)
{
	int i, j;

	for (i = 1; i < n; i++)
		for (j = i; j > 0 && a[j-1] > a[j]; j--)
			swap(a+j, a+j-1);
}
```

time complexity O(n^2), space complexity O(n)

binsearch - 二分查找 (sorted array)

```
int binsearch(int x, int v[], int n)
{
	int low, high, mid;

	low = 0;
	high = n - 1;
	while (low <= high) {
		mid = (low + high) / 2;
		if (x < v[mid])
			high = mid - 1;
		else if (x > v[mid])
			low = mid + 1;
		else	/* found match */
			return mid;
	}
	return -1;	/* no match */
}
```

时间复杂度分析

```
while (low <= high)	=>	while (n)

while (n) {
	if (x == v[mid])
		return mid;
	n /= 2;
}
```

time complexity O(logn), space complexity O(n)

7 常量，字符串，字符数组

const, string, character array

7.1 数字常量，字符常量

integer const:

1. int 23, 150
2. unsigned 23U, 150u
3. long 23L, 150l

4. unsigned long 23UL 150ul
5. octal 037 (31d), 0177 (127d)
6. hex 0x1f 0X7F

floating const:
1. double 2.38, 12.5, 1e-2(1 x 10 ^-2)
2. float 2.38f, 12.5f 1e-2f

字符常量 - character const
一个字符常量是一个整数，书写时将一个字符括在单引号中，如，'x'。字符在机器字符集中的数值就是字符常量的值。

ASCII code 字符集
'0' ~ '9' 48 ~ 57
'A' ~ 'Z' 65 ~ 90
'a' ~ 'z' 97 ~ 122

'\0' 0
'\t' 9 /* tab */
'\n' 10 /* newline */
' ' 32 /* space */

字符常量'\0'表示值为 0 的字符，也就是空字符（null）。我们通常用'\0'的形式代替 0，以强调某些表达式的字符属性，但其数字值为 0。

\a	响铃符	\\	反斜杠
\b	回退符	**\?**	问号
\f	换页符	**\'**	单引号
\n	换行符 (newline)	**\"**	双引号
\r	回车符	\ooo	八进制数
\t	横向制表符 (tab)	\xhh	十六进制数
\v	纵向制表符		

字符常量一般用来与其它字符进行比较，但 也可以像其它整数一样参与数值运算

```
int     i;
char    c;
i = '8' - '0';      c = 'a' + i;
```

常量表达式是仅仅只包含常量的表达式。这种表达式在编译时求值，而不在运行时求值。

```
int     i, j;
i = 3 + 8 * 6;    j = 5 * i;
```

example:

```
#include <stdio.h>

int main()
{
	int	i;
	char	c;

	i = 'h' - 'a';
	c = 'a' + i;

	printf("a = %d\n", 'a');
	printf("i = %c - %c = %d\n", 'h', 'a', i);
	printf("\'%c\' = \'%c\' + %d = %d\n", c, 'a', i, c);
	printf("\n");

	printf("\\0\t%d\n", '\0');
	printf("\\n\t%d\n", '\n');
	printf("\\t\t%d\n", '\t');
	printf("space\t%d\n", ' ');

	return 0;
}
```

7.2 字符数组

数据类型为 char 的数组，或者说字节 (bytes) 数组。

```
	char name[7] = { 'R', 'i', 't', 'c', 'h', 'i', 'e' };
	for (int i = 0; i < 7; i++)
		printf("%c", name[i]);
```

当一个字符数组以 '\0' (0) 结尾时，这样的字符数组称为字符串。

```
	char name[] = { 'R', 'i', 't', 'c', 'h', 'i', 'e', '\0' };

	for (int i = 0; name[i] != '\0'; i++)
		printf("%c", name[i]);
	printf("\n");
	printf("%s\n", name);
```

字符数组与字符串的区别：

1. 字符数组通常用来声明一个字节缓冲区，数据类型由程序员根据需要设定和转换。

2. 字符串通常用来表示连续存放的 char 类型，通过结尾 0 对连续字符进行打包。因此字符串占用的内存空间比字符串本身的长度多一位以容纳 '\0'。
3. 当操作指向字符数组的指针时我们关心的是数组的长度以避免指针越界；对于指向字符串的指针我们通常关系的是结尾 '\0' 以指示是否到达字符串的末尾。终止 '\0' 隐含了字符串的长度信息。

同数组一样，字符数组或字符串不能进行整体赋值操作。

字符串数组有更简单的声明方式：

```
char name[] = "Dennis Ritchie";
```

以 " " 包围的字符串称为 字符串字面值 - string literal - 书写格式不是存储格式.

```
#include <stdio.h>

int main()
{
	char	name[] = "Brian Kernighan";
	int	l;

	l = 0;
	while (name[l] != '\0')
		l++;
	printf("name: %s\n", name);
	printf("string length: %d\n", l);
	printf("name size: %lu", sizeof name);
	return 0;
}
```

C 语言提供了许多字符串操作的常规库函数，上面的字符串长度可通过库函数计算：

```
#include <string.h>
int l = strlen(name);
```

几个常用字符串函数 <string.h>

1. char *strcpy(s, ct)　　将字符串 ct（包括'\0'）复制到字符串 s 中，并返 回 s
2. char *strcat(s, ct)　　将字符串 ct 连接到 s 的尾部，并返回 s
3. int strcmp(cs,ct)　　比较字符串 cs 和 ct；当 cs<ct 时，返回一个负数；当 cs==ct 时，返回 0；当 cs>ct 时，返回一个正数
4. size_t strlen(cs)　　返回字符串 cs 的长度

7.3 字符串 - string literal

字符串常量是一个字符数组，例如：

"I am a student"

在字符串的内部表示中，字符数组以空字符'\0'结尾，所以，程序可以通过检查空字符找到字符数组的结尾。字符串常量占据的存储单元数也因此比双引号内的字符数大 1。

对于函数参数，如

printf("hello, world\n");　　=>　　char *s = "hello, world\n"; printf("%s", s);

当类似于这样的一个字符串出现在程序中时，实际上是通过字符指针访问该字符串的。在上述语句中，printf 接受的是一个指向字符数组第一个字符的指针。也就是说，字符串常量可通过一个指向其第一个元素的指针访问。

```
char    *s;
s = "my test";
```

将把一个指向该字符数组的指针赋值给 s; 该过程并没有进行字符串的复制，而只是涉及到指针的操作。C 语言没有提供将整个字符串作为一个整体进行处理的运算符。

注意下面两个定义的差别：

```
char name[] = "Rob Pike";          /* name: string array */
char *pname = "Rob Pike";          /* pname: pointer to const string */
```

注意字符串中的空格 ' ' (space)，其内存值为 32，而不是 0 ('\0')

8 动态内存分配 - malloc and free

动态内存分配，在运行时分配内存。

void * - 通用指针
void * 指针是一个不与任何数据类型关联的指针。它指向存储中的某个数据位置，即指向变量的地址。它也被称为通用指针。

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

int main() {
```

```
        int     i = 5;
        float   f = 8.3;
        void    *p;
        int     *ip;
        float   *fp;

        p = &i;
        ip = (int *) p;
        printf("Integer variable is = %d\n", *((int*) p));
        printf("*ip value: %d\n\n", *ip);

        p = &f;
        fp = (float *) p;
        printf("Float variable is = %f\n", *((float*) p));
        printf("*fp value: %f\n", *fp);
        return 0;
}
```

函数原型 - malloc, free

```
#include <stdlib.h>

void *malloc(size_t size);      /* 分配size字节并返回一个指针到分配的内存。内存未初始化。*/
void free(void *ptr);           /* 释放ptr指向的内存空间 (由 malloc 分配) */


int     *p;
p = (int *) malloc(sizeof(int));
if (p == NULL) {
        fprintf(stderr, "malloc error");
        exit(1);
}
…
free(p);
```

example: sdup - 字符串复制

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

char *sdup(const char *s)
{
        char    *t;
        int     len;
```

```
        len = strlen(s);
        if ((t = malloc(len + 1)) == NULL) {
                fprintf(stderr, "malloc error");
                exit(1);
        }
        for (int i = 0; i < len; i++)
                t[i] = s[i];
        t[len] = '\0';
        return t;
}

int main()
{
        char    *str;

        str = sdup("Hellow, world");
        printf("%s\n", str);
        free(str);
        return 0;
}
```

C 语言提供了库函数 strdup

```
#include <string.h>
char *strdup(const char *s);
```

strdup() 函数返回一个指向新字符串的指针，该字符串是字符串 s 的副本。新字符串的内存使用 malloc(3) 获得，并且可以使用 free(3) 释放。

9 局部变量，全局变量，符号表

local, global, symbol table

C 程序由全局变量和函数组成。

9.1 局部变量 - local variable

在函数体内部或块内声明/定义的变量称为局部变量，局部变量由编译器在栈 (stack) 中自动分配内存，因此也叫做自动变量。局部变量具有 auto (storage class) 关键字属性。

局部变量只能由该函数或代码块内的语句使用。局部变量对于它们自身之外的函数是未知的。

9.2 全局变量 - global variable

在函数体之外声明的变量称为全局变量。

全局变量对于其后定义的所有函数均可见。

```
#include <stdio.h>

void f();

int	x = 10;
int	y;

int main()
{
	int	x;

	x = 20;
	y = 5;
	printf("main():\tx = %d\ty = %d\n\n", x, y);
	f();
	return 0;
}

void f()
{
	printf("f():\tx = %d\ty = %d\n", x, y);
}
```

9.3 符号表，作用域 - symbol table, scope

符号表是由编译器创建和维护的重要数据结构，用于跟踪变量的语义，即它存储有关名称的作用域 (scope)和绑定信息的信息，有关各种实体实例的信息，例如变量和函数名称，数组，结构等。

作用域
简单来说，变量的作用域就是它在程序中的生命周期。
这意味着变量的作用域是整个程序中声明、使用和修改变量的代码块。

10 结构和联合 - struct and union

10.1 结构 - struct

结构是 C 中一个用户定义的数据类型，它允许组合不同类型的数据项。

创建结构，结构变量

'struct' 关键字用于创建结构。下面是例子。

```
struct point {
        int     x;
        int     y;
};

struct book {
        char    title[50];
        char    author[50];
        int     id;
};

example: define and access structure
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct book {
        char    title[50];
        char    author[50];
        int     id;
};

int main()
{
        struct book book = {}, book1;

        strcpy(book1.title, "The C programming language");
        strcpy(book1.author, "Brain W. Kernighan & Dennis M. Ritchie");
        book1.id = 12806;

        struct book book2 = {"The UNIX programming environment",
                             "Kernighan & Rob Pike", 937699};
```

```
        book = book1;           /* struct assignment, memory copy */
        printf("title: %s\n", book.title);
        printf("author: %s\n", book.author);
        printf("book id: %d\n\n", book.id);

        book = book2;
        printf("title: %s\n", book.title);
        printf("author: %s\n", book.author);
        printf("book id: %d\n", book.id);
        return 0;
}
```

结构作为函数的参数

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

struct book {
        char    title[50];
        char    author[50];
        int     id;
};

void printbook(struct book);

int main()
{
        struct book book = {}, book1;

        strcpy(book1.title, "The C programming language");
        strcpy(book1.author, "Brain W. Kernighan & Dennis M. Ritchie");
        book1.id = 12806;

        struct book book2 = {"The UNIX programming environment",
                             "Kernighan & Rob Pike", 937699};
        book = book2;
        printbook(book);
        printbook(book1);
        printbook(book2);
        return 0;
}
```

```
void printbook(struct book book)
{
	printf("title: %s\n", book.title);
	printf("author: %s\n", book.author);
	printf("book id: %d\n", book.id);
}
```

指向结构的指针

example: 引用结构的地址

```
int main()
{
	struct book book1, *p;

	strcpy(book1.title, "The C programming language");
	strcpy(book1.author, "Brain W. Kernighan & Dennis M. Ritchie");
	book1.id = 12806;

	struct book book2 = {"The UNIX programming environment",
				"Kernighan & Rob Pike", 937699};
	p = &book1;
	printbook(*p);
	p = &book2;
	printbook(*p);

	return 0;
}
```

example: operate structure using pointer

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

struct book {
	char	title[50];
	char	author[50];
	int	id;
};

void printbook(struct book *);

int main()
{
	struct book *book1, *book2;
```

```
	book1 = (struct book *) malloc(sizeof(struct book));
	if (book1 == NULL) {
		fprintf(stderr, "malloc error");
		exit(1);
	}
	strcpy(book1->title, "The C programming language");
	strcpy(book1->author, "Brain W. Kernighan & Dennis M. Ritchie");
	book1->id = 12806;
	printbook(book1);

	if ((book2 = malloc(sizeof(*book2))) == NULL) {
		fprintf(stderr, "malloc error");
		exit(1);
	}
	*book2 = *book1;
	strcat(book2->title, " (second edition)");
	book2->id = 18307;
	printbook(book2);
	return 0;
}

void printbook(struct book *book)
{
	printf("title: %s\n", book->title);
	printf("author: %s\n", book->author);
	printf("book id: %d\n", book->id);
}
```

10.2 联合 - union

union 允许在同一内存位置存储不同的数据类型。可以定义具有许多成员的联合，但在任何给定时间只有一个成员可以包含值。联合提供了一种将同一内存位置用于多种用途的有效方式。

typedef

用于为另一种数据类型创建附加名称（别名），但不创建新类型，通常用于简化声明复杂的类型组成的结构。

```
typedef struct point Point;
struct point {
	int	x;
```

```
        int     y;
};
Point   p = {3, 5};

typedef struct book {
        char    title[50];
        char    author[50];
        int     id;
} Book;
```

switch 语句

switch 语句是一种多路判定语句，它测试表达式是否与一些常量整数值中的某一个值匹配，并执行相应的分支动作。

```
switch(expr) {
case const-expr: statements
cast const-expr: statements
default:
        statements
}
```

break 语句将导致程序的执行立即从 switch 语句中退出。在 switch 语句中，case的作用只是一个标号。

union 示例

union 的语法同 struct 相同。我们看一个加法的例子。

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

typedef union datum Datum;
union datum {
        int ival;
        float fval;
};

Datum add(Datum, Datum, int);

int main()
```

```
{
	Datum d1, d2, d3;

	d1.ival = 3;
	d2.ival = 10;
	d3 = add(d1, d2, 0);
	printf("3 + 10 = %d\n", d3.ival);

	d1.fval = 2.5;
	d2.fval = 3.1;
	d3 = add(d1, d2, 1);
	printf("2.5 + 3.1 = %.2f\n", d3.fval);

	return 0;
}

Datum add(Datum d1, Datum d2, int flag)
{
	Datum   d;

	switch(flag) {		/* 0 integer val, 1 float val */
	case 0:
		d.ival = d1.ival + d2.ival;
		break;
	case 1:
		d.fval = d1.fval + d2.fval;
		break;
	default:
		fprintf(stderr, "add: unsupport datum type");
		exit(1);
	}
	return d;
}
```

数据封装

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

typedef struct {
	int	utype;		/* 0 - int; 1 - float; 2 - double; 3 - long */
	union u_tag {
		int	i;
```

```
                float     f;
                double d;
                long     l;
        } u;
} constant;

int main()
{
        constant         cons;

        cons.utype = 2;
        cons.u.d = 3.1415926;

        printf("utype = %d\n", cons.utype);
        switch(cons.utype) {
        case 0:
                printf("value = %d\n", cons.u.i);
                break;
        case 2:
                printf("value = %f\n", cons.u.d);
                break;
        default:
                fprintf(stderr, "constant, unsupported utype");
                exit(1);
        }
        return 0;
}
```

11 链表 - link list

11.1 自引用结构 - self referential structure

具有一个或多个指针成员的结构，这些成员指针指向结构本身类型的结构。

```
struct node {
        int       data;
        struct node *next;
};
```

需要考虑的重要一点是指针应该在访问之前正确初始化，因为默认情况下它包含垃圾值。

创建一个新的 node 节点

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

struct node {
        int data;
        struct node *next;
};

struct node *newnode(int x)
{
        struct node *n;

        if ((n = malloc(sizeof(*n))) == NULL) {
                fprintf(stderr, "newnode: malloc error");
                exit(1);
        }
        n->data = x;
        n->next = NULL;
        return n;
}
int main()
{
        struct node *p;

        p = newnode(10);
        p->next = newnode(20);
        printf("node data: %d\n", p->data);
        printf("next node data: %d\n", p->next->data);

        return 0;
}
```

注意：为了简化问题，在本示例中，我们最后没有释放 (free) malloc 分配的 node 节点内存。

11.2 链表 - link list

链表 (link list) 是包含一系列连接节点 (node)的线性数据结构。在这里，每个节点 (node) 存储下一个节点的数据和地址。

每个节点都是相同的自引用结构，包括：

1. 一个数据项 (data item)
2. 另一个节点的地址 (pointer to another node, or NULL)

创建，打印链表

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

struct node {
	int data;
	struct node *next;
};

struct node *newnode(int);
void printlink(struct node *);

int main()
{
	struct node *head, *one, *two, *three;

	one = newnode(10);
	two = newnode(20);
	three = newnode(30);

	one->next = two;
	two->next = three;
	head = one;
	printlink(head);

	return 0;
}

void printlink(struct node *p)
{
	while (p != NULL) {
		printf("%d ", p->data);
		p = p->next;
	}

}
```

```
struct node *newnode(int x)
{
	struct node *n;

	if ((n = malloc(sizeof(*n))) == NULL) {
		fprintf(stderr, "newnode: malloc error");
		exit(1);
	}
	n->data = x;
	n->next = NULL;
	return n;
}
```

atolink - 数组转换为链表

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

struct node {
	int data;
	struct node *next;
};

void printlink(struct node *);
struct node *newnode(int);
struct node *atolink(int *, int);

int main()
{
	int arr[] = { 10, 20, 30, 40, 50 };
	struct node *head;

	head = atolink(arr, sizeof(arr)/sizeof(arr[0]));
	printlink(head);
	return 0;
}

struct node *atolink(int a[], int n)		/* convert array to link list */
{
	struct node head = {}, *cur;

	cur = &head;
```

```
	for (int i = 0; i < n; i++)
		cur = cur->next = newnode(a[i]);

	return head.next;
}

void printlink(struct node *p)
{
	while (p != NULL) {
		printf("%d ", p->data);
		p = p->next;
	}

}

struct node *newnode(int x)
{
	struct node *n;

	if ((n = malloc(sizeof(*n))) == NULL) {
		fprintf(stderr, "newnode: malloc error");
		exit(1);
	}
	n->data = x;
	n->next = NULL;
	return n;
}
```

12 预处理器 - preprocessor

预处理器执行宏替换、条件编译以及包含指定的文件，以#开头的命令行就是预处理器处理的对象。

预处理器有单独的语法，它不是 C 语言的一部分。

本章不会学习新的预处理器指令，而是对学过的部分进行简单的总结。

#include 指令
包含预定义的头文件名，并以 < > 包围，如

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

#define 指令
#define 指令允许在源代码中定义宏。这些宏定义允许声明常量值以在整个代码中使用。

宏定义不是变量，不能像变量一样被程序代码更改。在创建表示数字、字符串或表达式的常量时，通常会使用此语法。

```
#include <stdio.h>

#define NAME "Tom"
#define AGE 10

int main()
{
	printf("%s is over %d years old.\n", NAME, AGE);
	return 0;
}
```

宏参数
类函数宏可以接受参数，就像真正的函数一样。参数必须是有效的 C 标识符，以逗号和可选的空格分隔。

```
#include <stdio.h>

#define MAX(a,b)	(a) > (b) ? (a) : (b)

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

	x = 10;
	y = 20;
	z = MAX(x, y);
	printf("max of x and y: %d\n", z);
	return 0;
}
```

```
z = MAX(x, y)  -> z = (x) > (y) ? (x) : (y)
```

注意：宏参数 a, b 可以是表达式，为了保证正确的优先级，将 a, b 分别用 () 括起来是必要的。

与处理器单行解析，行尾不需要 ; 终结，如果一条指令需要跨越多行，需要在行尾添加 \ 进行连接：

```
#define MACROS	long line ….	\
```

```
continue line   \
end line
```

我们聚焦 C 语法的学习，不宜过早学习复杂的预处理指令，够用即可。同样的包括 makefile, 文件操作等。

13 标准输入, 标准输出, 错误输出 - stdin, stdout, stderr

printf();
scanf();
fprintf();

14 课程回顾

C 语言关键字：

auto	break	case	char	const	continue
default	do	double	else	enum	extern
float	for	goto	if	int	long
register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void
volatile	while				