**1.愉快的开始-hello world**

* 1. **include 头文件包含**

include 告诉编译器，包含一个头文件。在 C 语言中，任何库函数调用都需要包含头文件。

<头文件>，是让编译器去系统目录下寻找相关的头文件“头文件”，是让编译器去当前目录下寻找相关的头文件

* 1. **main 函数**

main 函数是 C 语言中的主函数，一个 C 语言的程序有且只有一个主函数。

* 1. **注释**

// 单行注释，单行注释是 C++语言的注释方法

/\* \*/ 多行注释，多行注释是标准 C 语言的注释方法

* 1. **{}括号**

C 语言所有函数的代码都包含在{}中。

* 1. **声明**

int a;

声明一个变量，名称为 a，类型为 int。变量的名称是可以自定义的。

* 1. **C 语言自定义名字的要求**
     1. 可以使用字母、数字、下划线，但不能以数字开头
     2. 字母区分大小写
     3. 不能以系统关键字作为名称
     4. 每一行代码必须以英文状态下的分号结尾
  2. **printf 函数**

printf 是向标准输出设备输出字符串

如果要输出一个字符串，如：printf(“hello world”);

如果要输出一个整数，如：printf(“%d”,整数);

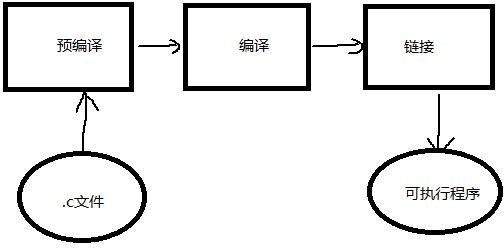
* 1. **return 语句**

一个函数遇到 return 语句就会终止。return 是 C 语言的关键字。

* 1. **system 系统调用**

system 库函数的功能是执行操作系统的命令或者运行指定的程序。system 库函数的调用需要使用 include <stdlib.h> 。

* + 1. **system 返回值在 Windows 和 Unix 下不同**
    2. **POSIX**
  1. **C 语言编译过程gcc 参数简介**

**1.10.1**

* + 1. **–E 预编译**

gcc –E –o a.e a.c

预编译 a.c 文件，生成的目标文件名为 a.e。

预编译是展示 include 包含的头文件内容，同时隐藏代码中的注释。

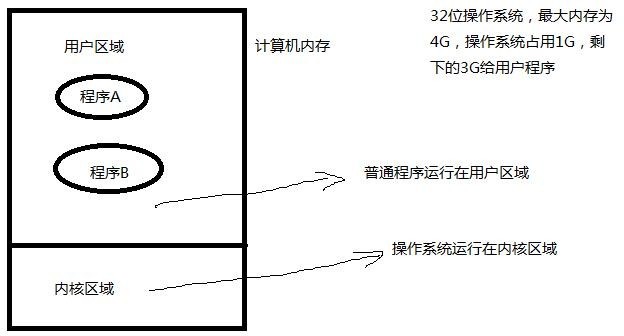
* + 1. **–S 汇编**

-S 是将 C 语言转化为汇编语言

* + 1. **–c 编译**

将代码编译为二进制的机器指令

* 1. **操作系统结构**



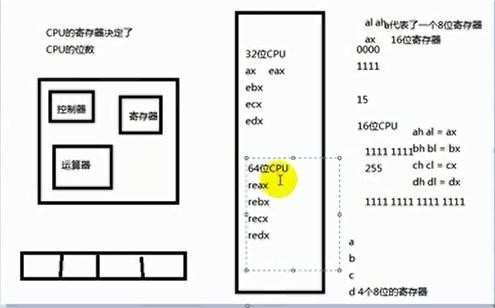
* + 1. **用户模式**

应用程序都是运行在用户区域。

* + 1. **内核模式**

操作系统的内核，设备驱动程序，都是运行在内核区域。

* 1. **64 位、32 位系统区别**
     1. **CPU 内部结构与寄存器**



* + 1. **RISC 与 CISC CPU 架构**

RISC 精简指令集

CISC 复杂指令集，一般来讲，x86 架构的 CPU 都是复杂指令的，AMD，Intel 就是 x86 架构的，Linux 就是基于 x86 的操作系统。

* + 1. **SPARC，x86 与 ARM**

Sun 有自己的 CPU，是基于 SPARC，其实就是一款 RISC 的 CPU。

* 1. **汇编语言**
     1. **I386 汇编简介**

mov eax, 10

add eax, 10

sub eax, 20 call printf

* + 1. **VS 反汇编**

先按 f9 设置一个断点

F5，用调式运行代码

* 1. **IDE 工具**

**2.C 语言中的数据类型**

* 1. **常量**

常量就是在程序中不可变化的量。

* + 1. **#define**

定义一个宏常量。

* + 1. **const**

定义一个 const 常量。

* 1. **字符串常量**

对于#define 类型定义的常量，常量名习惯为大写；对于 const 类型定义的常量及变量，名称习惯为小写结合大写。

* 1. **二进制数、位、字节与字**

一个位只能表示 0 或 1 两种状态，简称 bit，1 位=1bit

一个字节为 8 个二进制数，称为 8 位，简称 Byte，1 字节=8 位一个字为 2 个字节，简称 WORD

两个字为双字，简称 DWORD

* 1. **八进制**

以 8 为基数的数制系统就是 8 进制，C 语言中用 0 表示 8 进制，如 0666。

|  |  |  |
| --- | --- | --- |
| 十进制 | 八进制 | 二进制 |
| 0 | 0 | 0 |
| 1 | 1 | 1 |
| 2 | 2 | 10 |
| 3 | 3 | 11 |
| 4 | 4 | 100 |
| 5 | 5 | 101 |
| 6 | 6 | 110 |
| 7 | 7 | 111 |

* 1. **十六进制**

以 16 位基数的数制系统就是 16 进制，C 语言中用 0x 表示 16 进制。

|  |  |  |
| --- | --- | --- |
| 十进制 | 十六进制 | 二进制 |
| 0 | 0 | 0 |
| 1 | 1 | 1 |
| 2 | 2 | 10 |
| 3 | 3 | 11 |
| 4 | 4 | 100 |
| 5 | 5 | 101 |
| 6 | 6 | 110 |

|  |  |  |
| --- | --- | --- |
| 7 | 7 | 111 |
| 8 | 8 | 1000 |
| 9 | 9 | 1001 |
| 10 | A | 1010 |
| 11 | B | 1011 |
| 12 | C | 1100 |
| 13 | D | 1101 |
| 14 | E | 1110 |
| 15 | f | 1111 |

十进制转化为 8 进制，将十进制数作为被除数，8 作为除数，取得商数和余数，直到商数为 0 时，将余数倒过来就是所求的 8 进制数。

十进制转化为 16 进制，将十进制数作为被除数，16 作为除数，取得商数和余数，直到商数为 0 时，将余数倒过来就是所求的 16 进制数。

* 1. **原码**

将最高位作为符号位（0 代表正，1 代表负），其余各位取数值本身的绝对值。

* 1. **反码**

一个数如果为正，那么反码和原码相同。如果为负，那么符号位为 1，其余位取

反。

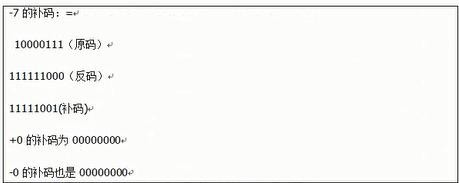
* 1. **补码**

号位。

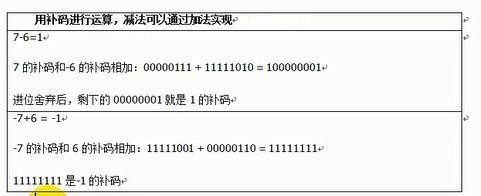
原码、反码不利于计算机的运算，如：原码表示的 7 和-7 的相加，还需要判断符

正数：原码、反码和补码相同。

负数：最高位为 1，对其余位取反，最后对整体+1。



补码符号位不动，其他位取反，最后整个数+1，得到原码。



计算机内部运算正数使用原码，负数使用补码。

* 1. **sizeof 关键字**

sizeof 是 C 语言关键字，用于求指定数据类型在内存中的大小，单位是字节。

sizeof 与 size\_t 类型

* 1. **int 类型**
     1. **int 常量、变量**

**i**nt 就是 32 位的一个二进制整数，在内存中占据 4 个字节的空间。

* + 1. **printf 输出 int 值**

%d，输出一个有符号的 10 进制数，

%u，输出一个无符号的 10 进制数。

* + 1. **printf 输出 8 进制和 16 进制**

%x，代表输出 16 进制数，

%X，用大写字母输出 16 进制数，

%o，代表输出 8 进制数。

* + 1. **short,long,long long,unsigned int**

short 意思为短整数，在 32 位操作系统下为 2 个字节，即 16bit=16 位。

long 意思为长整数，在 32 位操作系统下为 4 个字节，在 64 位 Windows 系统下是 4 个字节，Unix 系统下为 8 个字节。

int 不管在 32 位系统下还是 64 位系统下，不管是 Windows 系统还是 Unix 系统都是 4 个字节。

long long 是 64 位，即 8 个字节大小的整数，对于 32 位系统，CPU 寄存器是 32

位的，所以计算 long long 类型的数据，效率很低。

* + 1. **整数溢出**

计算一个整数时，超过整数能够容纳的最大单位后，整数会溢出，溢出的结果是高位舍弃。

当一个小的整数给大的整数，符号位不会丢失，会继承。

* + 1. **大端对齐与小端对齐**

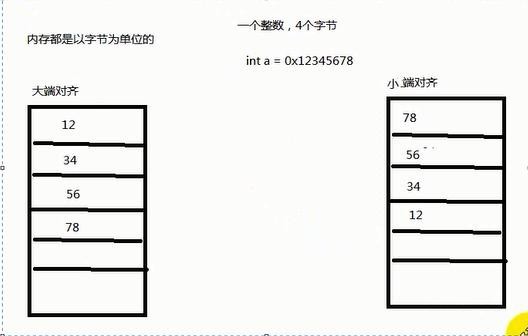
对于 arm，Intel 这种 x86 架构的复杂指令 CPU，整数在内存中是倒着存放的，即低地址放高位，高地址放低位，这就是“小端对齐”。但对于 Unix 服务器的 CPU，更多的是采用“大端对齐”的方式存放整数。

* 1. **char 类型**
     1. **char 常量，变量**

char c; 定义一个 char 变量。

‘a’，char 的常量。

char 的本质就是一个整数，一个只有 1 个字节的整数。



* + 1. **printf 输出 char**

%c 意思是输出一个字符。

* + 1. **不可打印的字符使用转义符**

\a 警报

\b 退格

\n 换行

\r 空格

\t 制表符

\\ 斜杠

\’单引号

\”双引号

\? 问号

* 1. **浮点float,double,long double 类型**
     1. **浮点常量，变量**

float 在 32 位系统下是 4 个字节，double 在 32 位系统下是 8 个字节。小数的效率很低，避免使用，除非明确的计算小数。

* + 1. **printf 输出浮点数**

%f，输出一个 double，

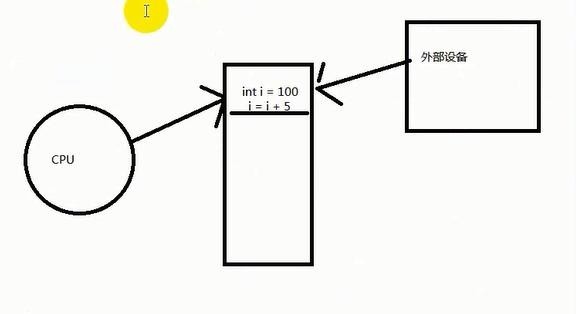
%lf，输出一个 long double。

* 1. **类型限定**
     1. **const**

const 代表一个不能改变值的常量。

* + 1. **volatile**

代表变量是一个可能被 CPU 指令之外的地方改变的。编译器就不会针对这个变量去优化代码。

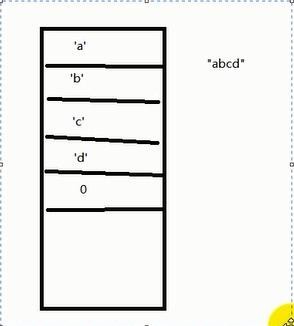


* + 1. **register**

变量在 CPU 寄存器里面，而不是在内存里面，但是 register 是建议型的指令，而不是命令型的指令。

1. **字符串格式化输出和输入**
   1. **字符串在计算机内部的存储方式**

字符串是内存中一段连续的 char 空间，以’\0’结尾。

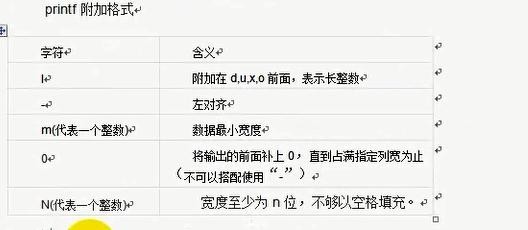


“”是 C 语言表达字符串的方式。

* 1. **printf 函数，putchar 函数**





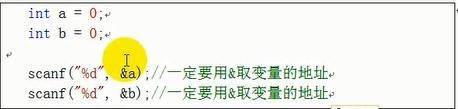


putchar 是显示一个字符的函数。

* 1. **scanf 函数和getchar 函数**

scanf 获取用户通过键盘输入的值，放入变量中，注意参数一定是变量的地址

(&)。



getchar 获取用户通过键盘输入的字符。

1. **运算符表达式和语句**
   1. **基本运算符**
      1. **基本概念**

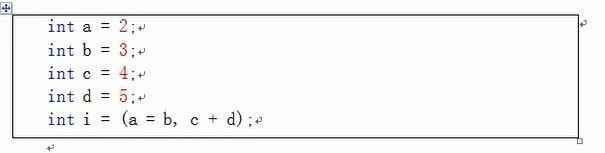
数据对象：泛指数据在内存中的存储区域。

左值：表示可以被更改的数据对象。右值：能赋给左值的量。

**4.1.2 加（+）、减（-）、乘（\*）、除（/）、取余或取模（%）、+=、-=、\*=、/=、%=、**

**++、--**

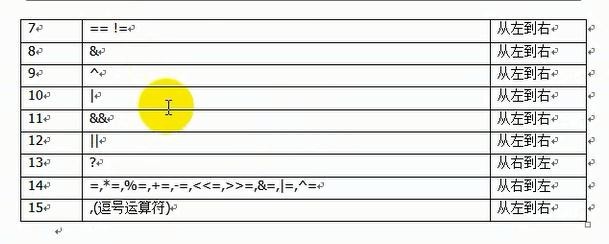
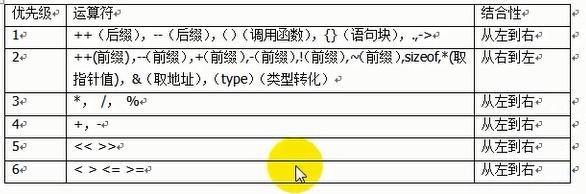
* + 1. **逗号运算符**



逗号表达式先求逗号左边的值，再求逗号右边的值，整个语句的值是逗号右边的

值。

* + 1. **运算符优先级**



* 1. **复合语句**

{}代码块

* 1. **空语句**

只有一个分号的语句就是空语句，空语句在 C 语言中是合法的，并且在某些场合是必用的。

* 1. **类型转化**

图片

（）为强制类型转化运算符

1. **分支语句**
   1. **比较运算符**

在 C 语言中，0 代表 false，非 0 代表真。

**5.1.1 < <= > >= == !=**

* 1. **比较运算符优先级**

前四种相同，后两种相同，前四种优先级高于后两种。

* 1. **逻辑运算符**
     1. **逻辑与（&&）、逻辑或（||）、逻辑非（！）**
  2. **if**

单分支。

if(条件){

//复合语句

}

当条件为真的时候，执行复合语句；当条件为假的时候，不执行复合语句。

* 1. **if else**

双分支。

if(条件){

//复合语句 1

} else{

//复合语句 2

}

如果条件为真，执行复合语句 1；否则执行复合语句 2。

* 1. **if else if**

多重分支。

if(条件 1)

{

复合语句 1

}

else if(条件 2)

{

复合语句 2

}

else if(条件 3)

{

复合语句 3

}

else

{

复合语句 4

}

当有多个 else 的时候，else 总是和上方最近的那个 if 语句配对。

* 1. **switch 与break,default**

多重选择。

什么时候用 if，什么时候用 switch?

当条件很复杂，一个条件中含有&&,||,!时，适合使用 if；如果条件很简单，但是分支很多且不是范围值，适合使用 switch。

* 1. **三目运算符**

一个求绝对值的例子

int i = -8;

int x = (i <0) ? –i : i;

先求？左边的条件。如果条件为真，那么等于：左边的值；否则等于：右边的值

* 1. **goto 语句与标号**

无条件跳转。

不建议使用 goto 语句，goto 语句会使得程序的可读性变差。

1. **循环语句**
   1. **while**

while（条件），如果条件为真，循环继续；如果条件为假，循环结束。 while(1)是死循环。

* 1. **continue**

循环遇到 continue 语句，不再执行 continue 下面的代码，而是直接返回到循环起始语句处继续执行。

* 1. **break**

循环遇到 break 语句，立刻终止循环，执行后续代码。

* 1. **do while**

do

{

//复合语句；

}while（条件）；

对于 do while 来讲，循环的复合语句至少被执行一次； 对于 while 来讲，有可能循环的复合语句一次也不执行。

* 1. **for**

1. **数组**
   1. **一维数组的定义和使用**
   2. **数组在内存中的存储方式**

数组在内存中就是一段连续的空间，并且每个元素的类型是一样的。

* 1. **一维数组初始化**

|  |  |  |  |
| --- | --- | --- | --- |
| int | array[10] = | | {10, 1, 2, 3, 4, 5, 6, 7, 8, 19};//定义数组的同时为数组的成员初始化 |
| int | array[10] = | | {3 ,7 ,9};//将数组的前 3 个元素赋值，后面的元素置为 0 |
| int | array[10] = | | { 0 };//将数组所有的元素都置为 0 |
| int | i | = 0; |  |
| for | (i | = 0; i < | 10; i++) |
| { |  |  |  |
|  |  | array[i] | = 0;//通过循环遍历数组的每个元素，将元素的值置为0 |
| } |  |  |  |

* 1. **二维数组定义与使用**
  2. **二维数组初始化**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| int | array[2][3] | = | { | { | 1, 2, 3 }, { 1, 2, 3 } };//定义一个二维数组并初始化成员值 |
| int | array[2][3] | = | { | 0 | };//定义一个二维数组并初始化元素值为 0 |

1. **字符串与字符数组**
   1. **字符数组定义**
   2. **字符数组初始化**

|  |  |
| --- | --- |
| char | array[100] = {‘a’, ‘b’, ‘c’, ‘d’}; |
| char | array[100] = “abcd”; |
| char | array[100] = {0}; |
| char | array[] = “abcd”; |

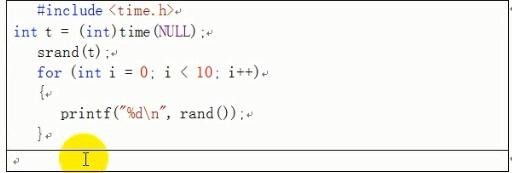
* 1. **字符数组使用**
  2. **随机数产生函数rand 与 srand**

头文件 stdlib.h

rand 是伪随机数产生器，每一次调用 rand 产生的随机数是一样的。如果调用

rand 之前先调用 srand，就可以出现任意的随机数。

只要保证每次调用 srand 时，参数值不同，rand 函数就会产生不同的随机数



* 1. **使用scanf 输入字符串**
  2. **字符串的结束标志**

scanf 将回车，空格都认为是字符串输入结束标志

* 1. **字符串处理函数**
     1. **gets**

gets 不能用类似“%s”或者“%d”之类的字符转义，只能接受字符串的输入

* + 1. **fgets 函数**

gets 函数不检查预留缓冲区是否能够容纳用户实际输入的数据，多出来的字符会导致内存溢出，fgets 函数改进了这个问题。

由于 fgets 函数是为读取文件设计的，所以读取键盘时没有 gets 那么方便

* + 1. **puts 函数**

puts 函数打印字符串，与 printf 不同，puts 会在最后自动添加一个‘\n’

* + 1. **fputs 函数**

fputs 函数是 puts 的文件操作版本

* + 1. **strlen,字符串长度**
    2. **strcat,字符串追加**
    3. **strncat 字符串有限追加**
    4. **strcmp,字符串比较**
    5. **strncmp 字符串有限比较**
    6. **strcpy 字符串拷贝**
    7. **strncpy 字符串有限拷贝**
    8. **sprintf 格式化字符串**
    9. **sscanf 函数**
    10. **strchr 查找字符**
    11. **strstr 查找子字符串**
    12. **strtok 分割字符串**
    13. **atoi 转换为 int**

需要包含头文件 stdlib.h

* + 1. **atof 转化为float**
    2. **atol 转换为 long**

1. **函数**
   1. **函数的原型和调用**
   2. **函数的形参和实参**
   3. **函数的返回类型和返回值**
   4. **main 函数与 exit 函数**
   5. **多个源代码文件程序的编译**
      1. **头文件的使用**

如果把 main 函数放在第一个文件中，而把自定义函数放在第二个文件中，就需要在第一个文件中声明函数原型

如果把函数原型包含在一个头文件里，就不必每次使用函数的时候都声明其原型 了，把函数声明放入头文件是很好的习惯。

* + 1. **#include 和#define 的意义**#include 就是简单的文件内容替换#define 就是简单的文本替换
    2. **#ifndef 与#endif**

#ifndef 意思是条件预编译，如果#ifndef 后面的条件成立，就预编译#ifndef 开始到#endif 之间的代码，否则就不会预编译这段代码

* 1. **函数的递归**

1. **指针**
   1. **指针**
      1. **指针的概念**

指针变量也是一个变量

指针存放的内容是一个地址，该地址指向一块内存空间。



* + 1. **指针变量的定义**

专门用于存放另一个变量的地址的变量

* + 1. **&取地址运算符**

&可以获取一个变量在内存当中的地址

* + 1. **无类型指针**

void\* p

* + 1. **NULL**

当一个指针不指向任何一个有效内存地址的时候，此时把指针设置为 NULL

空指针也称为野指针

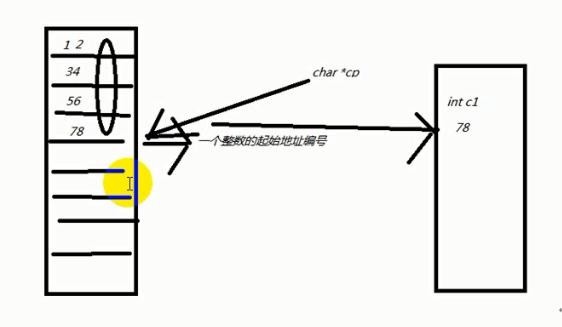
值为 NULL 的指针为空指针，没有指向任何变量地址的指针叫做野指针

* + 1. **指针的兼容性**

指针之间赋值比普通数据类型赋值更为严格，例如：不可以把一个 double\*赋值给

int\*

原则上是相同类型的指针指向相同类型的变量地址。



* + 1. **常量指针和指向常量的指针**

const char\* p;//定义一个指向常量的指针char\* const p;//定义一个常量指针，

* + 1. **指针与数组的关系**
    2. **指针运算**

指针运算不是简单的整数加减法，而是指针**指向的数据类型在内存中占用的字节数作为倍数的运算**

char \*p;

p++移动了 sizeof(char)个字节数int \*p;

p++移动了 sizeof(int)个字节数

求差值，p1-p2，通常用于同一个数组内求两个元素之间的距离 比较 p1==p2，通常用来比较两个指针是否指向同一个位置

* + 1. **通过指针使用数组元素**
    2. **指针数组**
    3. **指向指针的指针（二级指针）**

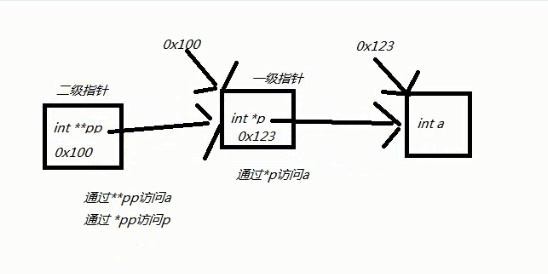
int \*p[5];

指针就是一个变量，存储内存地址，可以定义一个指向指针的指针。

int i = 10; int \*p1 = &i;

int \*\*p2 = &p1; printf(“%d\n”, \*\*p2);

以此类推可以定义 3 级甚至多级指针



* + 1. **指向二维数组的指针**

|  |  |
| --- | --- |
| int buf[3][5] | 二维数组名称，buf 代表数组首元素地址 |
| int (\*a)[5] | 定义指向 int[5]类型的指针变量 |
| a[0],\*(a+0),\*a | 0 行，0 列的元素地址 |
| a+1 | 第 1 行首地址 |
| a[1], \*(a+1) | 第 1 行，0 列元素地址 |

|  |  |
| --- | --- |
| a[1]+2,\*(a+1)+2,&a[1][2] | 第 1 行，2 列的元素地址 |
| \*(a[1]+2), ,a[1][2] | 第 1 行，2 列元素的值 |

* + 1. **指针变量作为函数的参数**

函数的参数可以是指针类型，它的作用是将一个变量的地址传递给另一个函数。 通过函数的指针参数可以间接地实现形参修改实参的值。

* + 1. **一维数组名作为函数参数**

当数组名作为函数参数时，C 语言将数组名解释为指针

int func(int array[10]);

* + 1. **二维数组名作为函数参数**

二维数组做函数参数时可以不指定第一个下标

int func(int array[][10]);

将二维数组作为函数参数的情况不是很多

* + 1. **const 关键字保护数组内容**

如果将一个数组作为函数的形参，那么数组内容可能在被调用函数的内部修改，有时候不希望数组内容被修改，可以使用 const 参数。

func(const int array[]);

* + 1. **指针作为函数的参数**
    2. **指针作为函数的返回值**
    3. **指向函数的指针**

指针可以指向变量，数组，也可以指向一个函数。

一个函数在编译的时候会分配一个入口地址，这个入口地址就是函数的指针，函数名称就代表函数的入口地址。

函数指针的定义方式：int (\*p)(int n);//定义了一个指向 int func(int n)类型的函数地址的指针。

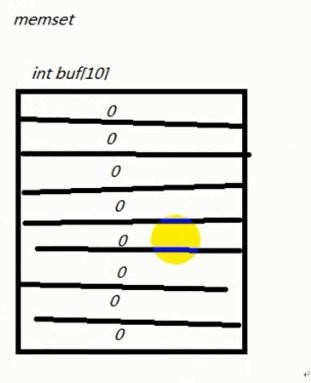
1. 定义函数指针变量的形式为：函数返回类型（\*指针变量名称）（参数列表）
2. 函数可以通过函数指针调用
3. int(\*p)()代表指向一个函数，但不是固定哪个函数
   * 1. **把指向函数的指针作为函数的参数**

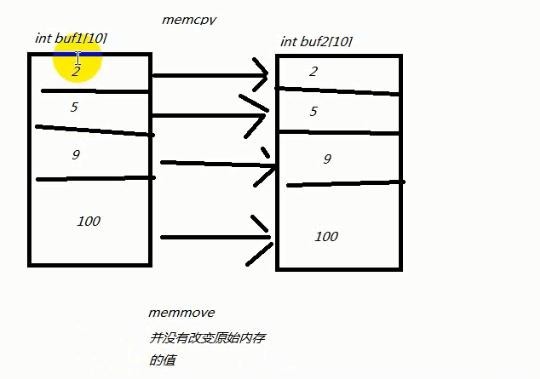
**把函数指针作为另一个函数的参数称为回调函数**

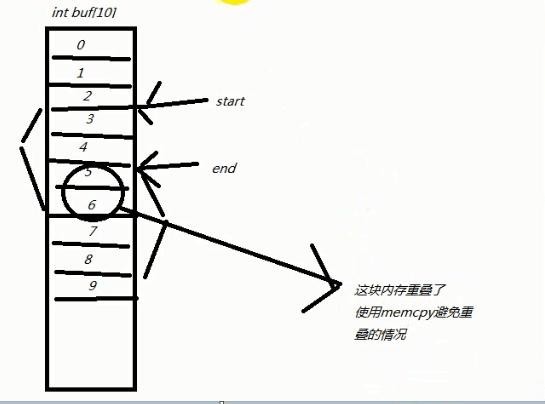
* + 1. **memset,memcpy,memmove 函数**

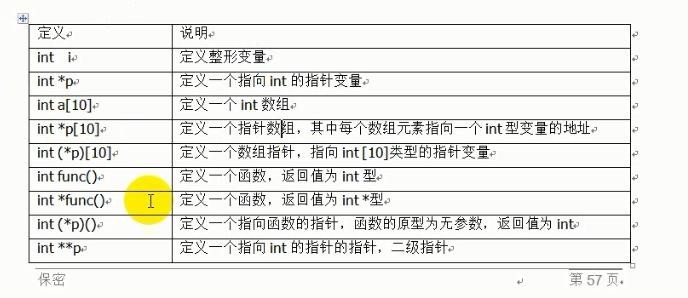
这三个函数分别实现内存设置，内存拷贝和内存移动 使用 memcpy 时，一定要确保内存没有重叠区域

#include <string.h>







**指针总结**

1. **字符指针与字符串**
   1. **指针与字符串**

在 C 语言中，大多数字符串操作就是指针操作

* 1. **通过指针访问字符串数组**
  2. **函数的参数为char**

1. **内存管理**
   1. **作用域**

一个 C 语言变量的作用域可以使代码块作用域，函数作用域或者文件作用域代码块{}是之间的一段代码

出现在{}之外的变量就是全局变量

* + 1. **auto 自动变量**

一般情况下代码块内部定义的变量都是自动变量，当然也可以显式地使用 auto 关

键字

* + 1. **register 寄存器变量**

通常变量在内存中，如果能把变量放到 CPU 的寄存器里面，代码执行效率会更高

Register int i;

* + 1. **代码块作用域的静态变量**

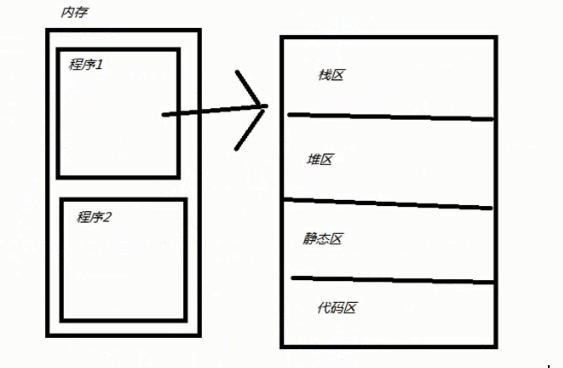
**静态变量是指内存位置在程序执行期间一直不改变的变量**，一个代码块内部的静态变量只能被这个代码块内部访问

* + 1. **代码块作用域之外的静态变量**

代码块之外的静态变量在程序执行期间一直存在，但只能被定义这个变量的文件访

问

* + 1. **全局变量**
    2. **外部变量和 extern 关键字**
    3. **全局函数和静态函数**
  1. **内存四区**



* + 1. **代码区**

代码区 code，程序被操作系统加载到内存的时候，所有的可执行代码都加载到代码区，也叫代码段，这块内存是不可以在运行期间修改的。

* + 1. **静态区**

所有的全局变量以及程序中的静态变量都存储到静态区，比较一下两段代码的区

别：

|  |  |
| --- | --- |
| Int a=0; | Int a=0; |

|  |  |
| --- | --- |
| Int main()  {  Static int b=0; Printf(“%p,%p\n”,&a,&b); Return 0;  } | Static int b=0; Int main()  {  Printf(“%p,%p\n”,&a,&b); Return 0;  } |

* + 1. **栈区**

栈 stack 是一种先进后出的内存结构，所有的自动变量，函数的形参都是由编译器自动放入栈中，当一个自动变量超出其作用域时，自动从栈中弹出。

对于自动变量，何时入栈，何时出栈不需要程序控制，由 C 语言编译器实现。栈不会很大，一般是以 k 为单位的

* + 1. **栈溢出**

当栈空间已满，却还在往栈中放入变量

对于一个 32 位操作系统，最大管理 4G 内存，其中 1G 是给操作系统自己使用的，剩下的 3G 都是给用户程序，一个用户程序理论上可以使用 3G 的内存空间。

* + 1. **堆区**

堆 heap 和栈一样，也是一种在程序运行过程中可以随时修改的内存区域，但没有栈那样先进后出的顺序。

堆是一个大容器，它的容量要远远大于栈，但是在 C 语言中，堆内存空间的申请和释放需要手动通过代码来完成

* 1. **堆的分配和释放**

操作系统在管理内存的时候，最小单位不是字节，是内存页。

* + 1. **malloc**

void \* malloc(size\_t \_Size)

指针。

malloc 函数在堆中分配参数\_Size 指定大小的内容，单位：字节，函数返回 void \*

* + 1. **free**

void free(void \*p);

free 负责在堆中释放 malloc 分配的内存，参数 p 为 malloc 返回的堆中的内存地

址

* + 1. **calloc**

void \* calloc(size\_t \_Count, size\_t \_Size); calloc 与 malloc 类似，负责在堆中分配内存。

第一个参数是所需内存的单元数量，第二个参数是所需内存的单元大小（单位：字节），calloc 自动将分配的内存置 0

* + 1. **realloc**

成功返回新分配的堆内存地址，失败返回 NULL

1. **结构体，联合体，枚举和 typedef**
   1. **结构体**
      1. **定义结构体 struct 和初始化**
      2. **访问结构体成员**

.操作符

**13.1.5 结构数组**

1. **文件操作**
   1. **fopen**

r 以只读方式打开文件，该文件必须存在

r+以可读写方式打开文件，该文件必须存在

rb+读写打开一个二进制文件，允许读写数据，文件必须存在rw+读写打开一个文本文件，允许读和写

w 打开只写文件，若文件存在则文件长度清为 0，即该文件内容会消失。若文件不存在，则建立该文件

w+打开可读写文件，若文件存在则文件长度清为 0，即该文件内容会消失。若文件不存在，则建立该文件

a 以追加的方式打开只写文件。若文件不存在，则会建立该文件，如果文件存在， 写入的数据会被加到文件尾部，即文本原先的内容会被保留。（EOF 符保留）

a+以追加方式打开可读写的文件。若文件不存在，则会建立该文件，如果文件存 在，写入的数据会被加到文件尾部，即文件原先的内容会被保留。（原来的 EOF 符不保留）

* 1. **二进制模式和文本模式的区别**

1. 在 windows 系统中，文本模式下，文件以”\r\n”代表换行。如果以文本模式打开文件， 并使用 fputs 等函数写入换行符”\n” 时， 函数会自动在”\n” 前面加上”\r”。即实际写入文件的是”\r\n”。
2. 在类 Unix/Linux 系统中文本模式下，文件以”\n”代表换行。所以 Linux 系统

中在文本模式和二进制下并无区别。

* 1. **fclose**

fclose 关闭 fopen 打开的文件

* 1. **getc 和putc 函数**
  2. **EOF 与feof 函数文件结尾**

程序怎么才能知道是否已经到达文件结尾了，EOF 代表文件结尾如果文件是文件尾部，feof 函数返回 true

* 1. **fprintf,fscanf,fgets,fputs 函数**

这些函数都是通过 FILE \*对文件进行读写。

* 1. **stat 函数**

#include <sys/stat.h>

int stat(const char \* \_Filename, struct stat \*\_Stat) stat.st\_size;//文件大小，单位：字节

函数的第一个参数代表文件名，第二个参数是 struct stat 结构。

* 1. **fread 和 fwrite 函数**

size\_t fread(void \*buffer, size\_t size, size\_t count, FILE \*stream);

size\_t fwrite(const void \*buffer, size\_t size, size\_t count, FILE \*stream);

注意：这两个函数以二进制形式对文件进行操作，不局限于文本文件 返回值：返回实际写入的数据块数目

* 1. **fread 和 feof**

1. **基础数据结构与算法**
   1. **什么是数据结构**

数据是对客观事物的符号表示，在计算机中是指所有能输入到计算机中并被计算机处理的数据总称。

数据元素是数据的基本单位，在计算机中通常作为一个整体进行处理。 数据对象是性质相同的数据元素的集合，是数据的一个子集。

数据结构是相互之间存在一种或多种特定关系的数据元素的集合。

数据类型是和数据结构密切相关的一个概念，在计算机语言中，每个变量、常量或者表达式都有一个所属的数据类型。

抽象数据类型（abstract data type ADT）是指一个数据模型以及定义在该模型上的一组操作，抽象数据类型的定义仅仅取决于它的一组逻辑性，与其在计算机内部如何表示以及实现无关。

* 1. **什么是算法**
  2. **排序**
  3. **查找**
  4. **链表**
     1. **单向链表定义**

对于数组，逻辑关系上相邻的两个元素的物理位置也是相邻的，这种结构的优点是可以随机存储任意位置的元素，但缺点是如果从数组中间删除或插入元素时，需要大量移 动元素，效率低下。

链式存储结构的特点，元素的存储单元可以使连续的，也可以是不连续的。因此为了表示每个元素 a,与其后面的元素 a+1 之间的关系，对于元素 a，除了存储其本身信息之外，还需要存储一个指向其后元素的地址。这两部分数据称为结点（node）。

一个结点中存储的数据元素被称为数据域，存储指向后面元素的域叫做指针域。n 个结点的存储映像链接成一个链表。

整个链表必须从头结点开始进行，头结点的指针指向下一个结点的位置，最后一个结点的指针指向 NULL。

在链表中，通过指向其后结点位置的指针将链表中的每个结点链接在一起。