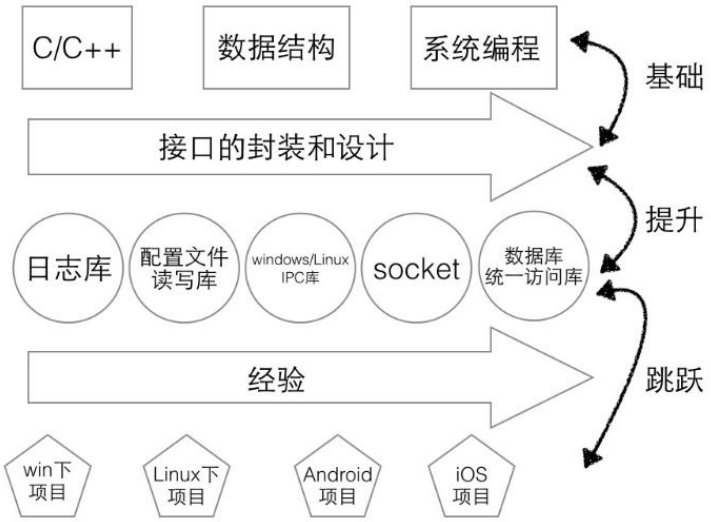
1. 前言

企业需要能干活的人，需要能上战场的兵。

1.1 技术层次

对于解决问题的解决方案有清晰的架构图，那么对于技术学习也要分清层次:



1.2 接口的封装设计

```
//初始化网络连接句柄 socket,也叫环境初始化
int socketclient_init(void** handle);

//发送报文接口
int socketclient_send(void* handle, unsigned char* buf, int buflen);

//接收报文接口
int socketclient_recv(void* handle, unsigned char* buf, int* buflen);

//socket 环境释放
int socketclient_destroy(void** handle);
```

1.3 过程的封装设计

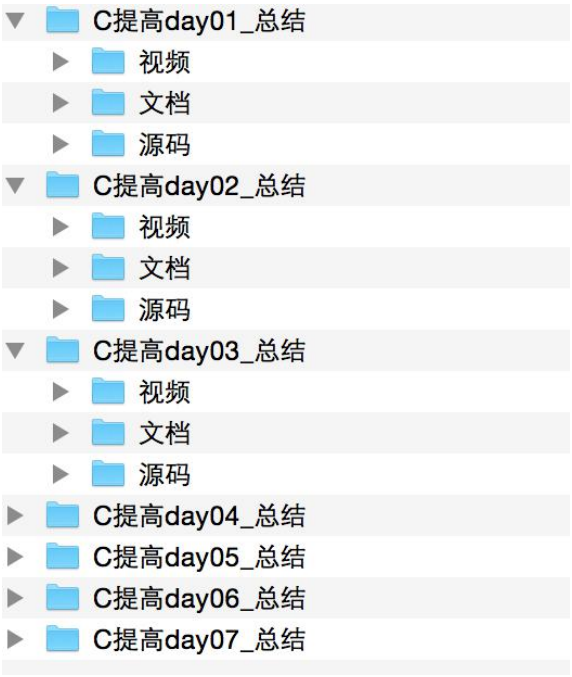
```
//打印函数
void PrintArray(int arr[],int len){

    //打印
    for (int i = 0; i < len; i++){
        printf("%d ", arr[i]);
    }
    printf("\n");

}

void MySort(int arr[],int len){
    //排序
    for (int i = 0; i < len; i++){

        for (int j = len - 1; j > i; j--){

            if (arr[j] < arr[j - 1]){
                int temp = arr[j];
                arr[j] = arr[j - 1];
                arr[j - 1] = temp;
            }
        }
    }
}

void test(){

    int arr[] = { 10, 50, 20, 90, 30 };
    int len = sizeof(arr) / sizeof(int);

    PrintArray(arr, len);
    MySort(arr, len);
    PrintArray(arr, len);
}
```

1.4 战前准备

1.4.1 听课要求

- 专心听讲、积极思考；
- 遇到不懂的暂时先记下，课后再问；
- 建议准备一个笔记本(记录重点、走神的时间)；
- 当堂动手运行，不动手，永远学不会；
- 杜绝边听边敲(如果老师讲的知识点很熟，你可以边听边敲)、杜绝犯困听课。
- 如果时间允许，请课前做好预习；
- 从笔记、代码等资料中复习上课讲过的知识点。尽量少回看视频，别对视频产生依赖，可以用 2 倍速度回看视频；
- 按时完成老师布置的练习，记录练习中遇到的 BUG 和解决方案，根据自己的理解总结学到的知识点；
- 初学者 应该抓住重点，不要钻牛角尖遇到问题了，优先自己尝试解决，其次谷歌百度，最后再问老师；
- 如果时间允许，可以多去网上找对应阶段的学习资料面试题，注意作息，积极锻炼。

1.4.2 资料管理



2. C 语言概述

欢迎大家来到 c 语言的世界，c 语言是一种强大的专业化的编程语言。

1.1 C 语言的起源

贝尔实验室的 Dennis Ritchie 在 1972 年开发了 C,当时他正与 ken Thompson 一起设计 UNIX 操作系统，然而，C 并不是完全由 Ritchie 构想出来的。它来自 Thompson 的 B 语言。

1.2 使用 C 语言的理由

在过去的几十年中，c 语言已成为最流行和最重要的编程语言之一。它之所以得到发展，是因为人们尝试使用它后都喜欢它。过去很多年中，许多人从 c 语言转而使用更强大的 c++语言，但 c 有其自身的优势，仍然是一种重要的语言，而且它还是学习 c++的必经之路。

- 高效性。c 语言是一种高效的语言。c 表现出通常只有汇编语言才具有的精细的控制能力(汇编语言是特定 cpu 设计所采用的一组内部制定的助记符。不同的 cpu 类型使用不同的汇编语言)。如果愿意，您可以细调程序以获得最大的速度或最大的内存使用率。
- 可移植性。c 语言是一种可移植的语言。意味着，在一个系统上编写的 c 程序经过很少改动或不经过修改就可以在其他的系统上运行。
- 强大的功能和灵活性。c 强大而又灵活。比如强大灵活的 UNIX 操作系统便是用 c 编写的。其他的语言(Perl、Python、BASIC、Pascal)的许多编译器和解释器也都是用 c 编写的。结果是当你在一台 Unix 机器上使用 Python 时，最终由一个 c 程序负责生成最后的可执行程序。

1.3 C 语言标准

1.3.1 K&R C

起初，C 语言没有官方标准。1978 年由美国电话电报公司(AT&T）贝尔实验室正式发表了 C 语言。布莱恩·柯林汉（Brian Kernighan） 和 丹尼斯·里奇（Dennis Ritchie） 出版了一本书，名叫《The C Programming Language》。这本书被 C 语言开发者们称为 **K&R**，很多年来被当作 C 语言的非正式的标准说明。人们称这个版本的 C 语言为 **K&R C**。

K&R C 主要介绍了以下特色：结构体（struct）类型；长整数（long int）类型；无符号整数（unsigned int）类型；把运算符=+和=-改为+=和-=。因为=+和=-会使得编译器不知道使用者要处理 i = -10 还是 i =- 10，使得处理上产生混淆。

即使在后来 ANSI C 标准被提出的许多年后，K&R C 仍然是许多编译器的最准要求，许多老旧的编译器仍然运行 K&R C 的标准。

1.3.2 ANSI C/C89 标准

1970 到 80 年代，C 语言被广泛应用，从大型主机到小型微机，也衍生了 C 语言的很多不同版本。1983 年，美国国家标准协会（ANSI）成立了一个委员会 X3J11，来制定 C 语言标准。

1989 年，美国国家标准协会（ANSI）通过了 C 语言标准，被称为 **ANSI X3.159-1989 "Programming Language C"**。因为这个标准是 1989 年通过的，所以一般简称 **C89 标准**。有些人也简称 **ANSI C**，因为这个标准是美国国家标准协会（ANSI）发布的。

1990 年，国际标准化组织（ISO）和国际电工委员会（IEC）把 C89 标准定为 C 语言的国际标准，命名为 **ISO/IEC 9899:1990 - Programming languages -- C**[5] 。因为此标准是在 1990 年发布的，所以有些人把简称作 **C90 标准**。不过大多数人依然称之为 **C89 标准**，因为此标准与 ANSI C89 标准完全等同。

1994 年，国际标准化组织（ISO）和国际电工委员会（IEC）发布了 C89 标准修订版，名叫 ISO/IEC 9899:1990/Cor 1:1994[6] ，有些人简称为 **C94 标准**。

1995 年，国际标准化组织（ISO）和国际电工委员会（IEC）再次发布了 C89 标准修订版，名叫 ISO/IEC 9899:1990/Amd 1:1995 - C Integrity[7] ，有些人简称为 **C95 标准**。

1.3.3 C99 标准

1999 年 1 月，国际标准化组织（ISO）和国际电工委员会（IEC）发布了 C 语言的新标准，名叫 **ISO/IEC 9899:1999 - Programming languages -- C** ，简称 **C99 标准。**这是 C 语言的第二个官方标准。

例如：

增加了新关键字 restrict，inline，_Complex，_Imaginary，_Bool

支持 long long，long double _Complex，float _Complex 这样的类型

支持了不定长的数组。数组的长度就可以用变量了。声明类型的时候呢,就用 int a[*] 这样的写法。不过考虑到效率和实现，这玩意并不是一个新类型。

3. 内存分区

3.1 数据类型

3.1.1 数据类型概念

> 什么是数据类型？为什么需要数据类型?
>
> 数据类型是为了更好进行内存的管理,让编译器能确定分配多少内存。

我们现实生活中，狗是狗，鸟是鸟等等，每一种事物都有自己的类型，那么程序中使用数据类型也是来源于生活。

当我们给狗分配内存的时候，也就相当于给狗建造狗窝，给鸟分配内存的时候，也就是给鸟建造一个鸟窝，我们可以给他们各自建造一个别墅，但是会造成内存的浪费，不能很好的利用内存空间。

我们在想，如果给鸟分配内存，只需要鸟窝大小的空间就够了，如果给狗分配内存，那么也只需要狗窝大小的内存，而不是给鸟和狗都分配一座别墅，造成内存的浪费。

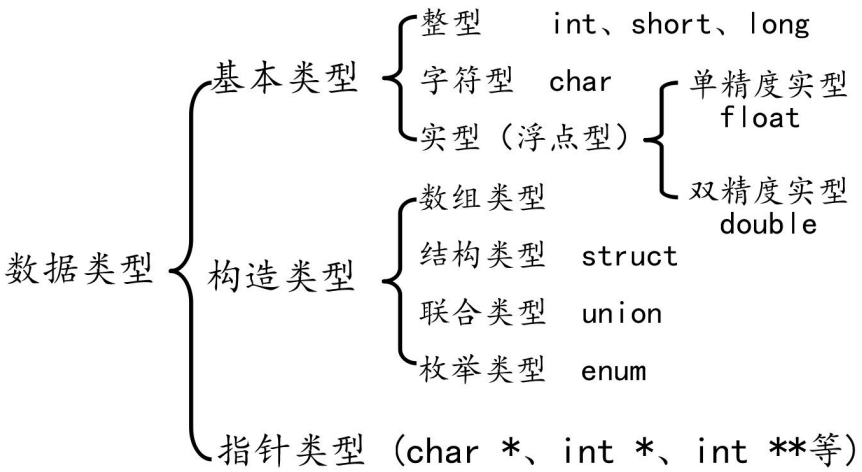
当我们定义一个变量，a = 10,编译器如何分配内存？计算机只是一个机器，它怎么知道用多少内存可以放得下 10？

所以说，数据类型非常重要，它可以告诉编译器分配多少内存可以放得下我们的数据。

狗窝里面是狗，鸟窝里面是鸟，如果没有数据类型，你怎么知道冰箱里放得是一头大象!

数据类型基本概念：

- 类型是对数据的抽象;
- 类型相同的数据具有相同的表示形式、存储格式以及相关操作;
- 程序中所有的数据都必定属于某种数据类型;
- 数据类型可以理解为创建变量的模具: **固定大小内存的别名**;



3.1.2 数据类型别名

示例代码：

```
typedef unsigned int u32;
typedef struct _PERSON{
    char name[64];
    int age;
}Person;

void test(){
    u32 val; //相当于 unsigned int val;
    Person person; //相当于 struct PERSON person;
}
```

3.1.3 void 数据类型

void 字面意思是”无类型”,void* 无类型指针，无类型指针可以指向任何类型的数据。

void 定义变量是没有任何意义的，当你定义 void a，编译器会报错。

void 真正用在以下两个方面：

- 对函数返回的限定；
- 对函数参数的限定；

示例代码：

```
//1. void 修饰函数参数和函数返回
void test01(void){
    printf("hello world");
}

//2. 不能定义 void 类型变量
void test02(){
    void val; //报错
}

//3. void* 可以指向任何类型的数据，被称为万能指针
void test03(){
```

```
    int a = 10;
    void* p = NULL;
    p = &a;
    printf("a:%d\n",*(int*)p);

    char c = 'a';
    p = &c;
    printf("c:%c\n",*(char*)p);
}

//4. void* 常用于数据类型的封装
void test04(){
    //void * memcpy(void * _Dst, const void * _Src, size_t _Size);
}
```

3.1.4 sizeof 操作符

sizeof 是 c 语言中的一个操作符，类似于++、--等等。sizeof 能够告诉我们编译器为某一特定数据或者某一个类型的数据在内存中分配空间时分配的大小，大小以字节为单位。

基本语法：

```
sizeof(变量);

sizeof 变量；

sizeof(类型);
```

sizeof 注意点：

- sizeof 返回的占用空间大小是为这个变量开辟的大小，而不只是它用到的空间。和现今住房的建筑面积和实用面积的概念差不多。所以对结构体用的时候，大多情况下就得考虑字节对齐的问题了；
- sizeof 返回的数据结果类型是 unsigned int；
- 要注意数组名和指针变量的区别。通常情况下，我们总觉得数组名和指针变量差不多，但是在用 sizeof 的时候差别很大，对数组名用 sizeof 返回的是整个数组的大

小，而对指针变量进行操作的时候返回的则是指针变量本身所占得空间，在 32 位机的条件下一般都是 4。而且当数组名作为函数参数时，在函数内部，形参也就是个指针，所以不再返回数组的大小；

示例代码：

```
//1. sizeof 基本用法
void test01(){
    int a = 10;
    printf("len:%d\n", sizeof(a));
    printf("len:%d\n", sizeof(int));
    printf("len:%d\n", sizeof a);
}

//2. sizeof 结果类型
void test02(){
    unsigned int a = 10;
    if (a - 11 < 0){
        printf("结果小于 0\n");
    }
    else{
        printf("结果大于 0\n");
    }
    int b = 5;
    if (sizeof(b) - 10 < 0){
        printf("结果小于 0\n");
    }
    else{
        printf("结果大于 0\n");
    }
}

//3. sizeof 碰到数组
void TestArray(int arr[]){
    printf("TestArray arr size:%d\n",sizeof(arr));
}
void test03(){
    int arr[] = { 10, 20, 30, 40, 50 };
    printf("array size: %d\n",sizeof(arr));

    //数组名在某些情况下等价于指针
    int* pArr = arr;
```

```
    printf("arr[2]:%d\n",pArr[2]);
    printf("array size: %d\n", sizeof(pArr));

    //数组做函数函数参数，将退化为指针,在函数内部不再返回数组大小
    TestArray(arr);
}
```

3.1.5 数据类型总结

- 数据类型本质是固定内存大小的别名，是个模具，C 语言规定：通过数据类型定义变量；
- 数据类型大小计算（sizeof）；
- 可以给已存在的数据类型起别名 typedef；
- 数据类型的封装（void 万能类型）；

3.2 变量

3.1.1 变量的概念

既能读又能写的内存对象，称为变量；

若一旦初始化后不能修改的对象则称为常量。

变量定义形式： 类型 标识符, 标识符, … , 标识符

3.1.2 变量名的本质

- 变量名的本质：一段连续内存空间的别名；
- 程序通过变量来申请和命名内存空间 int a = 0；
- 通过变量名访问内存空间；

- 不是向变量名读写数据，而是向变量所代表的内存空间中读写数据；

修改变量的两种方式：

```
    void test(){

    int a = 10;

    //1. 直接修改
    a = 20;
    printf("直接修改,a:%d\n",a);

    //2. 间接修改
    int* p = &a;
    *p = 30;

    printf("间接修改,a:%d\n", a);
}
```

3.3 程序的内存分区模型

3.3.1 内存分区

3.3.1.1 运行之前

我们要想执行我们编写的 c 程序，那么第一步需要对这个程序进行编译。

gcc –E index.c –o index.i

gcc –S index.i –o index.s //编译成汇编文件

gcc –c index.s –o index.o .obj

gcc index.o –o index //编译成可执行文件

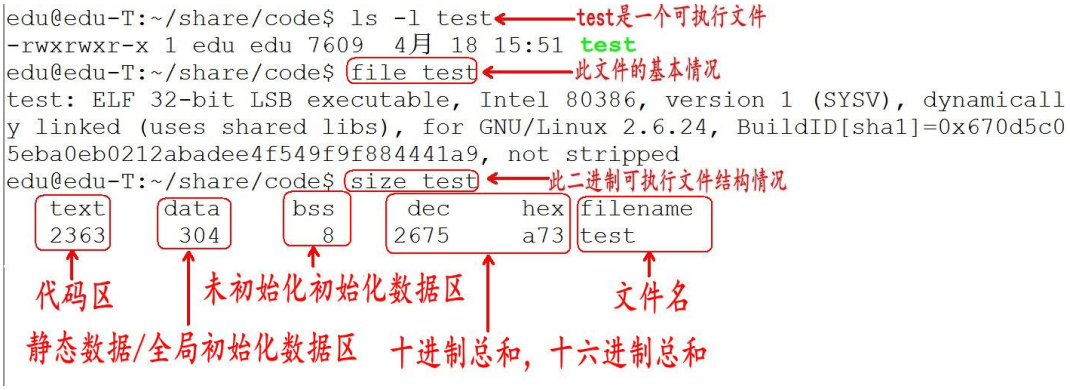
1）预处理：宏定义展开、头文件展开、条件编译，这里并不会检查语法

2）编译：检查语法，将预处理后文件编译生成汇编文件

3）汇编：将汇编文件生成目标文件(二进制文件)

4）链接：将目标文件链接为可执行程序

当我们编译完成生成可执行文件之后，我们通过在 linux 下 size 命令可以查看一个可执行二进制文件基本情况：



通过上图可以得知，在没有运行程序前，也就是说程序没有加载到内存前，可执行程序内部已经分好 3 段信息，分别为代码区（text）、数据区（data）和未初始化数据区（bss）3 个部分（有些人直接把 data 和 bss 合起来叫做静态区或全局区）。

- 代码区

存放 CPU 执行的机器指令。通常代码区是可共享的（即另外的执行程序可以调用它），使其可共享的目的是对于频繁被执行的程序，只需要在内存中有一份代码即可。代码区通常是只读的，使其只读的原因是防止程序意外地修改了它的指令。另外，代码区还规划了局部变量的相关信息。

- 全局初始化数据区/静态数据区（data 段）

该区包含了在程序中明确被初始化的全局变量、已经初始化的静态变量（包括全局静态变量

和局部静态变量）和常量数据（如字符串常量）。

- 未初始化数据区（又叫 bss 区）

存入的是全局未初始化变量和未初始化静态变量。未初始化数据区的数据在程序开始执行之前被内核初始化为 0 或者空（NULL）。

总体来讲说，程序源代码被编译之后主要分成两种段：程序指令和程序数据。代码段属于程序指令，而数据域段和.bss 段属于程序数据。

那为什么把程序的指令和程序数据分开呢？

- 程序被 load 到内存中之后，可以将数据和代码分别映射到两个内存区域。由于数据区域对进程来说是可读可写的，而指令区域对程序来讲说是只读的，所以分区之后呢，可以将程序指令区域和数据区域分别设置成可读可写或只读。这样可以防止程序的指令有意或者无意被修改；
- 当系统中运行着多个同样的程序的时候，这些程序执行的指令都是一样的，所以只需要内存中保存一份程序的指令就可以了，只是每一个程序运行中数据不一样而已，这样可以节省大量的内存。比如说之前的 Windows Internet Explorer 7.0 运行起来之后， 它需要占用 112 844KB 的内存，它的私有部分数据有大概 15 944KB，也就是说有 96 900KB 空间是共享的，如果程序中运行了几百个这样的进程，可以想象共享的方法可以节省大量的内存。

3.3.1.1 运行之后

程序在加载到内存前，代码区和全局区(data 和 bss)的大小就是固定的，程序运行期间不能改变。然后，运行可执行程序，操作系统把物理硬盘程序 load(加载)到内存，除了根据可执行程序的信息分出代码区（text）、数据区（data）和未初始化数据区（bss）之外，还额外增加了栈区、堆区。

- 代码区（text segment）

加载的是可执行文件代码段，所有的可执行代码都加载到代码区，这块内存是不可以在运行期间修改的。

- 未初始化数据区（BSS）

加载的是可执行文件 BSS 段，位置可以分开亦可以紧靠数据段，存储于数据段的数据（全局未初始化，静态未初始化数据）的生存周期为整个程序运行过程。

- 全局初始化数据区/静态数据区（data segment）

加载的是可执行文件数据段，存储于数据段（全局初始化，静态初始化数据，文字常量(只读)）的数据的生存周期为整个程序运行过程。

- 栈区（stack）

栈是一种先进后出的内存结构，由编译器自动分配释放，存放函数的参数值、返回值、局部变量等。在程序运行过程中实时加载和释放，因此，局部变量的生存周期为申请到释放该段栈空间。

- 堆区（heap）

堆是一个大容器，它的容量要远远大于栈，但没有栈那样先进后出的顺序。用于动态内存分配。堆在内存中位于 BSS 区和栈区之间。一般由程序员分配和释放，若程序员不释放，程序结束时由操作系统回收。

类型	作用域	生命周期	存储位置
auto 变量	一对{}内	当前函数	栈区
static 局部变量	一对{}内	整个程序运行期	初始化在 data 段，未初始化在 BSS 段
extern 变量	整个程序	整个程序运行期	初始化在 data 段，未初始化在 BSS 段
static 全局变量	当前文件	整个程序运行期	初始化在 data 段，未初始化在 BSS 段
extern 函数	整个程序	整个程序运行期	代码区
static 函数	当前文件	整个程序运行期	代码区
register 变量	一对{}内	当前函数	运行时存储在 CPU 寄存器
字符串常量	当前文件	整个程序运行期	data 段

注意：建立正确程序运行内存布局图是学好 C 的关键！！

3.3.2 分区模型

3.3.2.1 栈区

由系统进行内存的管理。主要存放函数的参数以及局部变量。在函数完成执行，系统自

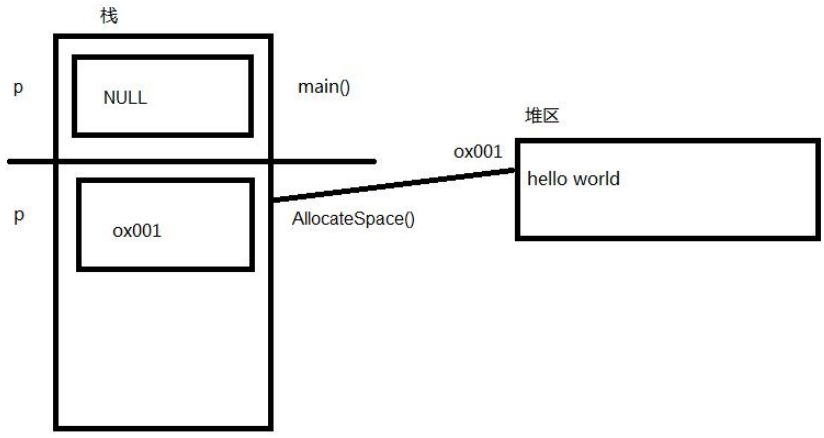
行释放栈区内存，不需要用户管理。

```
#char* func(){
    char p[] = "hello world!"; //在栈区存储 乱码
    printf("%s\n", p);
    return p;
}
void test(){
    char* p = NULL;
    p = func();
    printf("%s\n",p);
}
```

3.3.2.2 堆区

由编程人员手动申请，手动释放，若不手动释放，程序结束后由系统回收，生命周期是整个程序运行期间。使用 malloc 或者 new 进行堆的申请。

```c
char* func(){
    char* str = malloc(100);
    strcpy(str, "hello world!");
    printf("%s\n",str);
    return str;
}

void test01(){
    char* p = NULL;
    p = func();
    printf("%s\n",p);
}

void allocateSpace(char* p){
    p = malloc(100);
    strcpy(p, "hello world!");
    printf("%s\n", p);
}

void test02(){

    char* p = NULL;
    allocateSpace(p);

    printf("%s\n", p);
}
```



堆分配内存 API：

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

```
void *calloc(size_t nmemb, size_t size);
```

功能：

在内存动态存储区中分配 nmemb 块长度为 size 字节的连续区域。calloc 自动将分配的内存置 0。

参数：

nmemb：所需内存单元数量

size：每个内存单元的大小（单位：字节）

返回值：

成功：分配空间的起始地址

失败：NULL

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

功能：

重新分配用 malloc 或者 calloc 函数在堆中分配内存空间的大小。

realloc 不会自动清理增加的内存，需要手动清理，如果指定的地址后面有连续的空间，那么就会在已有地址基础上增加内存，如果指定的地址后面没有空间，那么 realloc 会重新分配新的连续内存，把旧内存的值拷贝到新内存，同时释放旧内存。

参数：

ptr：为之前用 malloc 或者 calloc 分配的内存地址，如果此参数等于 NULL，那么和 realloc 与 malloc 功能一致

size：为重新分配内存的大小，单位：字节

返回值：

成功：新分配的堆内存地址

失败：NULL

示例代码：

```
void test01(){

    int* p1 = calloc(10,sizeof(int));
    if (p1 == NULL){
        return;
    }
    for (int i = 0; i < 10; i ++){
        p1[i] = i + 1;
    }
    for (int i = 0; i < 10; i++){
        printf("%d ",p1[i]);
    }
    printf("\n");
    free(p1);
}
```

```c
void test02(){
    int* p1 = calloc(10, sizeof(int));
    if (p1 == NULL){
        return;
    }
    for (int i = 0; i < 10; i++){
        p1[i] = i + 1;
    }

    int* p2 = realloc(p1, 15 * sizeof(int));
    if (p2 == NULL){
        return;
    }

    printf("%d\n", p1);
    printf("%d\n", p2);

    //打印
    for (int i = 0; i < 15; i++){
        printf("%d ", p2[i]);
    }
    printf("\n");

    //重新赋值
    for (int i = 0; i < 15; i++){
        p2[i] = i + 1;
    }

    //再次打印
    for (int i = 0; i < 15; i++){
        printf("%d ", p2[i]);
    }
    printf("\n");

    free(p2);
}
```

3.3.2.3 全局/静态区

全局静态区内的变量在编译阶段已经分配好内存空间并初始化。这块内存在程序运行期间一直存在,它主要存储**全局变量**、**静态变量**和**常量**。

注意：

（1）这里不区分初始化和未初始化的数据区，是因为静态存储区内的变量若不显示初始化，则编译器会自动以默认的方式进行初始化，即静态存储区内不存在未初始化的变量。

（2）全局静态存储区内的常量分为常变量和字符串常量，一经初始化，不可修改。静态存储内的常变量是全局变量，与局部常变量不同，区别在于局部常变量存放于栈，实际可间接通过指针或者引用进行修改，而全局常变量存放于静态常量区则不可以间接修改。

（3）字符串常量存储在全局/静态存储区的常量区。

示例代码：

```
int v1 = 10;//全局/静态区
const int v2 = 20; //常量，一旦初始化，不可修改
static int v3 = 20; //全局/静态区
char *p1; //全局/静态区，编译器默认初始化为NULL

//那么全局static int 和 全局int变量有什么区别?

void test(){
    static int v4 = 20; //全局/静态区
}
```

加深理解：

```
char* func(){
    static char arr[] = "hello world!"; //在静态区存储 可读可写
    arr[2] = 'c';
    char* p = "hello world!"; //全局/静态区-字符串常量区
    //p[2] = 'c'; //只读，不可修改
    printf("%d\n",arr);
    printf("%d\n",p);
    printf("%s\n", arr);
    return arr;
}
void test(){
    char* p = func();
    printf("%s\n",p);
}
```

字符串常量是否可修改？字符串常量优化：

ANSI C 中规定：修改字符串常量，结果是未定义的。

ANSI C 并没有规定编译器的实现者对字符串的处理，例如：

1.有些编译器可修改字符串常量，有些编译器则不可修改字符串常量。

2.有些编译器把多个相同的字符串常量看成一个（这种优化可能出现在字符串常量中，节省空间），有些则不进行此优化。如果进行优化，则可能导致修改一个字符串常量导致另外的字符串常量也发生变化，结果不可知。

所以尽量不要去修改字符串常量！

C99 标准：
char *p = "abc"; defines p with type ‘‘pointer to char’’ and initializes it to point to an object with type ‘‘array of char’’ with length 4 whose elements are initialized with a character string literal. **If an attempt is made to use p to modify the contents of the array, the behavior is undefined.**

字符串常量地址是否相同？

tc2.0，同文件字符串常量地址不同。

Vs2013,字符串常量地址同文件和不同文件都相同。

Dev c++、QT 同文件相同，不同文件不同。

3.3.2.4 总结

在理解 C/C++内存分区时，常会碰到如下术语：数据区，堆，栈，静态区，常量区，全局区，字符串常量区，文字常量区，代码区等等，初学者被搞得云里雾里。在这里，尝试捋清楚以上分区的关系。

数据区包括：堆，栈，全局/静态存储区。

全局/静态存储区包括：常量区，全局区、静态区。

常量区包括：字符串常量区、常变量区。

代码区：存放程序编译后的二进制代码，不可寻址区。

可以说，C/C++内存分区其实只有两个，即代码区和数据区。

3.3.3 函数调用模型

3.3.3.1 函数调用流程

栈(stack)是现代计算机程序里最为重要的概念之一，几乎每一个程序都使用了栈，没有栈就没有函数，没有局部变量，也就没有我们如今能见到的所有计算机的语言。在解释为什么栈如此重要之前，我们先了解一下传统的栈的定义：

在经典的计算机科学中，栈被定义为一个特殊的容器，用户可以将数据压入栈中(入栈，push)，也可以将压入栈中的数据弹出(出栈，pop),但是栈容器必须遵循一条规则：先入栈的数据最后出栈(First In First Out,FIFO).

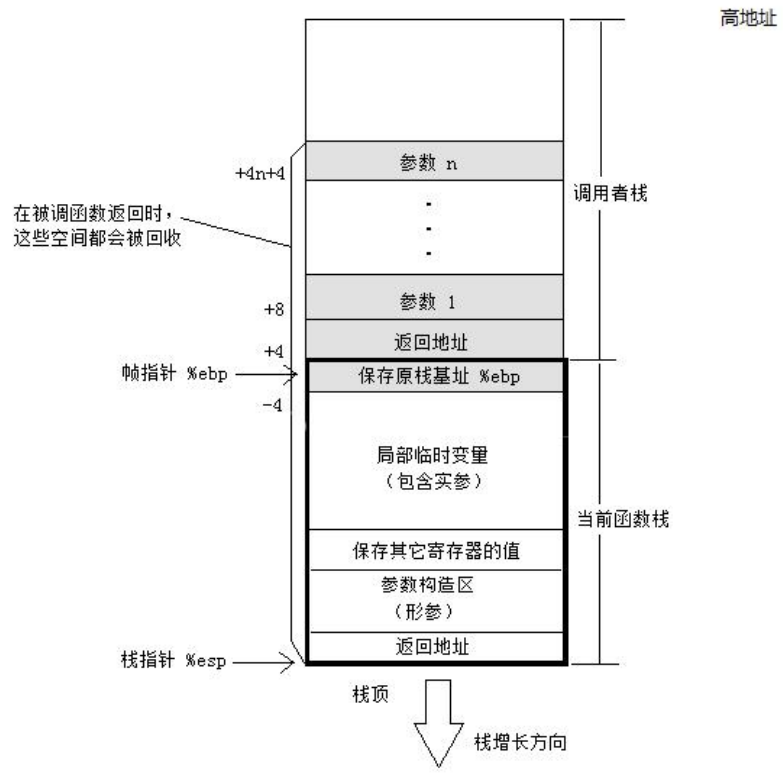
在经典的操作系统中，栈总是向下增长的。压栈的操作使得栈顶的地址减小，弹出操作使得栈顶地址增大。

栈在程序运行中具有极其重要的地位。最重要的，栈保存一个函数调用所需要维护的信息，这通常被称为堆栈帧(Stack Frame)或者活动记录(Activate Record).一个函数调用过程所需要的信息一般包括以下几个方面：

- 函数的返回地址；
- 函数的参数；
- 临时变量；
- 保存的上下文：包括在函数调用前后需要保持不变的寄存器。

我们从下面的代码，分析以下函数的调用过程：

```
int func(int a,int b){
    int t_a = a;
    int t_b = b;
    return t_a + t_b;
}

int main(){
    int a = 10;
    int b = 20;
    int ret = 0;
    ret = func(a, b);
    int a = 20;
    return EXIT_SUCCESS;
}
```



3.3.3.2 调用惯例

现在，我们大致了解了函数调用的过程，这期间有一个现象，那就是函数的调用者和被调用者对函数调用有着一致的理解，例如，它们双方都一致的认为函数的参数是按照某个

固定的方式压入栈中。如果不这样的话，函数将无法正确运行。

如果函数调用方在传递参数的时候先压入 a 参数，再压入 b 参数，而被调用函数则认为先压入的是 b,后压入的是 a,那么被调用函数在使用 a,b 值时候，就会颠倒。

因此，函数的调用方和被调用方对于函数是如何调用的必须有一个明确的约定，只有双方都遵循同样的约定，函数才能够被正确的调用，这样的约定被称为”调用惯例(Calling Convention)”.一个调用惯例一般包含以下几个方面：

函数参数的传递顺序和方式

函数的传递有很多种方式，最常见的是通过栈传递。函数的调用方将参数压入栈中，函数自己再从栈中将参数取出。对于有多个参数的函数，调用惯例要规定函数调用方将参数压栈的顺序：从左向右，还是从右向左。有些调用惯例还允许使用寄存器传递参数，以提高性能。

栈的维护方式

在函数将参数压入栈中之后，函数体会被调用，此后需要将被压入栈中的参数全部弹出，以使得栈在函数调用前后保持一致。这个弹出的工作可以由函数的调用方来完成，也可以由函数本身来完成。

为了在链接的时候对调用惯例进行区分，调用惯例要对函数本身的名字进行修饰。不同的调用惯例有不同的名字修饰策略。

事实上，在 c 语言里，存在着多个调用惯例，而默认的是 cdecl.任何一个没有显示指定调用惯例的函数都是默认是 cdecl 惯例。比如我们上面对于 func 函数的声明，它的完整写法应该是：

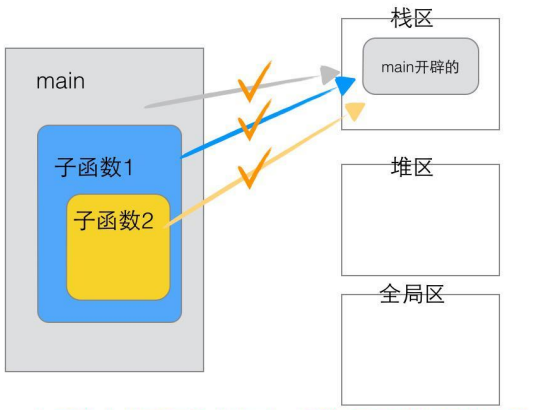
```
int _cdecl func(int a,int b);
```

注意: _cdecl 不是标准的关键字，在不同的编译器里可能有不同的写法，例如 gcc 里就

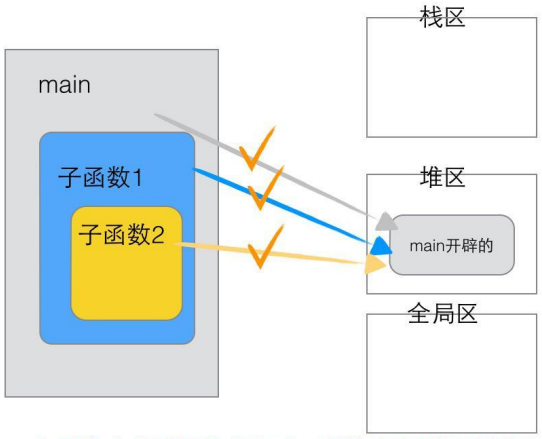
不存在_cdecl 这样的关键字，而是使用__attribute__((cdecl)).

调用惯例	出栈方	参数传递	名字修饰
cdecl	函数调用方	从右至左参数入栈	下划线+函数名
stdcall	函数本身	从右至左参数入栈	下划线+函数名+@+参数字节数
fastcall	函数本身	前两个参数由寄存器传递，其余参数通过堆栈传递。	@+函数名+@+参数的字节数
pascal	函数本身	从左至右参数入栈	较为复杂，参见相关文档

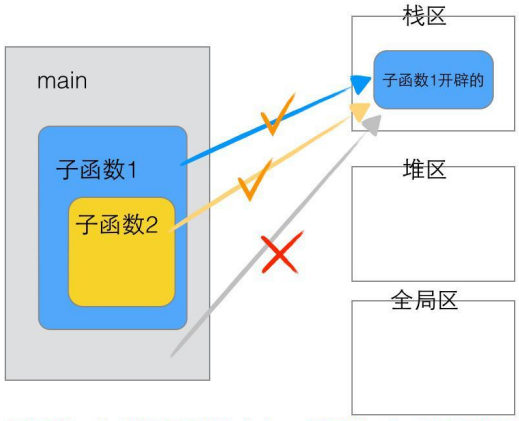
3.3.3.2 函数变量传递分析



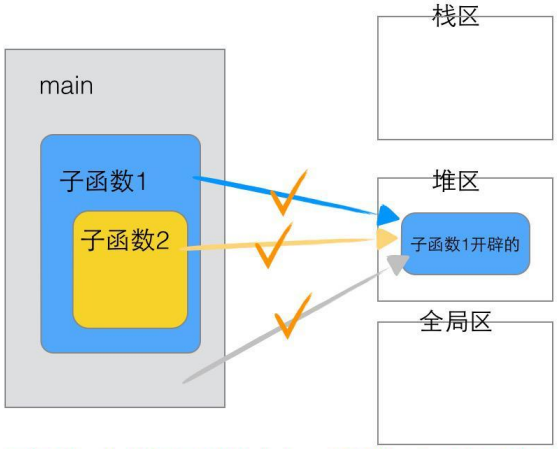
main函数在栈区开辟的内存，所有子函数均可以使用



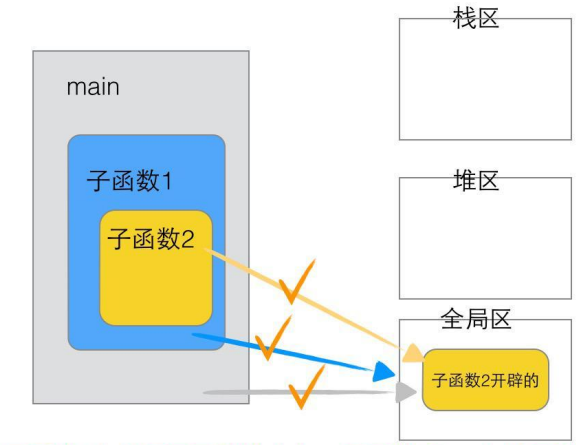
main函数在堆区开辟的内存，所有子函数均可以使用



子函数1在栈区开辟的内存，子函数1和2均可以使用

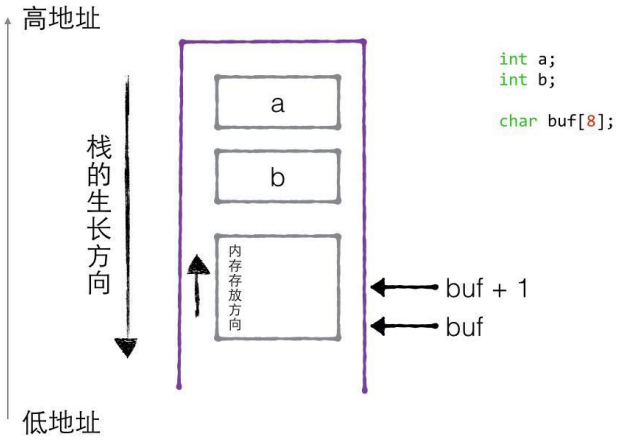


子函数1在栈区开辟的内存，子函数1和2均可以使用



子函数2在全局区开辟的内存，子函数1和main均可以使用

3.3.4 栈的生长方向和内存存放方向




```c
//1. 栈的生长方向
void test01(){

    int a = 10;
    int b = 20;
    int c = 30;
    int d = 40;

    printf("a = %d\n", &a);
    printf("b = %d\n", &b);
```

```
    printf("c = %d\n", &c);
    printf("d = %d\n", &d);

    //a的地址大于b的地址，故而生长方向向下
}

//2. 内存生长方向(小端模式)
void test02(){

    //高位字节 -> 地位字节
    int num = 0xaabbccdd;
    unsigned char* p = &num;

    //从首地址开始的第一个字节
    printf("%x\n",*p);
    printf("%x\n", *(p + 1));
    printf("%x\n", *(p + 2));
    printf("%x\n", *(p + 3));
}
```

4. 指针强化

4.1 指针是一种数据类型

4.1.1 指针变量

指针是一种数据类型，占用内存空间，用来保存内存地址。

```
void test01(){

    int* p1 = 0x1234;
    int*** p2 = 0x1111;

    printf("p1 size:%d\n",sizeof(p1));
    printf("p2 size:%d\n",sizeof(p2));


    //指针是变量，指针本身也占内存空间，指针也可以被赋值
    int a = 10;
    p1 = &a;
```

```
    printf("p1 address:%p\n", &p1);
    printf("p1 address:%p\n", p1);
    printf("a address:%p\n", &a);

}
```

4.1.2 野指针和空指针

4.1.2.1 空指针

标准定义了 NULL 指针，它作为一个特殊的指针变量，表示不指向任何东西。要使一个指针为 NULL,可以给它赋值一个零值。为了测试一个指针百年来那个是否为 NULL,你可以将它与零值进行比较。

对指针解引用操作可以获得它所指向的值。但从定义上看，NULL 指针并未执行任何东西，因为对一个 NULL 指针因引用是一个非法的操作，在解引用之前，必须确保它不是一个 NULL 指针。

如果对一个 NULL 指针间接访问会发生什么呢？结果因编译器而异。

不允许向 NULL 和非法地址拷贝内存：

```
void test(){
    char *p = NULL;
    //给 p 指向的内存区域拷贝内容
    strcpy(p, "1111"); //err

    char *q = 0x1122;
    //给 q 指向的内存区域拷贝内容
    strcpy(q, "2222"); //err
}
```

4.1.2.2 野指针

在使用指针时，要避免野指针的出现：

野指针指向一个已删除的对象或未申请访问受限内存区域的指针。与空指针不同，野指针无法通过简单地判断是否为 NULL 避免，而只能通过***养成良好的编程习惯***来尽力减少。对野指针进行操作很容易造成程序错误。

什么情况下回导致野指针？

- **指针变量未初始化**

任何指针变量刚被创建时不会自动成为 NULL 指针，它的缺省值是随机的，它会乱指一气。所以，指针变量在创建的同时应当被初始化，要么将指针设置为 NULL，要么让它指向合法的内存。

- **指针释放后未置空**

有时指针在 free 或 delete 后未赋值 NULL，便会使人以为是合法的。别看 free 和 delete 的名字（尤其是 delete），它们只是把指针所指的内存给释放掉，但并没有把指针本身干掉。此时指针指向的就是“垃圾”内存。释放后的指针应立即将指针置为 NULL，防止产生“野指针”。

- **指针操作超越变量作用域**

不要返回指向栈内存的指针或引用，因为栈内存在函数结束时会被释放。

```
void test(){
    int* p = 0x001; //未初始化
    printf("%p\n",p);
    *p = 100;
}
```

操作野指针是非常危险的操作，应该规避野指针的出现：

- **初始化时置 NULL**

指针变量一定要初始化为 NULL，因为任何指针变量刚被创建时不会自动成为 NULL 指针，它的缺省值是随机的。

- **释放时置 NULL**

当指针 p 指向的内存空间释放时，没有设置指针 p 的值为 NULL。delete 和 free 只是把内存空间释放了，但是并没有将指针 p 的值赋为 NULL。通常判断一个指针是否合法，都是使用 if 语句测试该指针是否为 NULL。

4.1.3 间接访问操作符

通过一个指针访问它所指向的地址的过程叫做间接访问，或者叫解引用指针，这个用于执行间接访问的操作符是*。

注意：对一个 int*类型指针解引用会产生一个整型值，类似地，对一个 float*指针解引用会产生了一个 float 类型的值。

int arr[5] arr = int* (&arr)

int arr1[5][3] arr1 = int(*)[3]

&arr1

- 在指针声明时，* 号表示所声明的变量为指针
- 在指针使用时，* 号表示操作指针所指向的内存空间

1）* 相当通过地址(指针变量的值)找到指针指向的内存，再操作内存

2）* 放在等号的左边赋值（给内存赋值，写内存）

3）* 放在等号的右边取值（从内存中取值，读内存）

```
//解引用
void test01(){

    //定义指针
    int* p = NULL;
    //指针指向谁，就把谁的地址赋给指针
    int a = 10;
    p = &a;
    *p = 20;//*在左边当左值，必须确保内存可写
    //*号放右面，从内存中读值
    int b = *p;
    //必须确保内存可写
    const char* str = "hello world!";
    *str = 'm';

    printf("a:%d\n", a);
    printf("*p:%d\n", *p);
    printf("b:%d\n", b);
}
```

4.1.4 指针的步长

指针是一种数据类型，是指它指向的内存空间的数据类型。指针所指向的内存空间决定了指针的步长。指针的步长指的是，当指针+1 时候，移动多少字节单位。

思考如下问题：

```
int a = 0xaabbccdd;
unsigned int *p1 = &a;
unsigned char *p2 = &a;

//为什么*p1打印出来正确结果？
printf("%x\n", *p1);
//为什么*p2没有打印出来正确结果？
printf("%x\n", *p2);
```

```
//为什么p1指针+1加了4字节？
printf("p1  =%d\n", p1);
printf("p1+1=%d\n", p1 + 1);
//为什么p2指针+1加了1字节？
printf("p2  =%d\n", p2);
printf("p2+1=%d\n", p2 + 1);
```

4.2 指针的意义_间接赋值

4.3.1 间接赋值的三大条件

通过指针间接赋值成立的三大条件：

1）2 个变量（一个普通变量一个指针变量、或者一个实参一个形参）

2）建立关系

3）通过 * 操作指针指向的内存

```
void test(){
    int a = 100; //两个变量
    int *p = NULL;
    //建立关系
    //指针指向谁，就把谁的地址赋值给指针
    p = &a;
    //通过*操作内存
    *p = 22;
}
```

4.3.2 如何定义合适的指针变量

```
void test(){
    int b;
    int *q = &b; //0 级指针
    int **t = &q;
    int ***m = &t;
    *m = 2 级指针
    *（2 级指针） = 1 级指针
    *（1 级指针） = 0 级指针
}
```

4.3.3 间接赋值：从 0 级指针到 1 级指针

```
int func1(){ return 10; }

void func2(int a){
    a = 100;
}
//指针的意义_间接赋值
void test02(){
    int a = 0;
    a = func1();
    printf("a = %d\n", a);

    //为什么没有修改?
    func2(a);
    printf("a = %d\n", a);
}

//指针的间接赋值
void func3(int* a){
    *a = 100;
}

void test03(){
    int a = 0;
    a = func1();
    printf("a = %d\n", a);

    //修改
    func3(&a);
    printf("a = %d\n", a);
}
```

4.3.4 间接赋值：从 1 级指针到 2 级指针

```
void AllocateSpace(char** p){
    *p = (char*)malloc(100);
    strcpy(*p, "hello world!");
}

void FreeSpace(char** p){
```

```
    if (p == NULL){
        return;
    }
    if (*p != NULL){
        free(*p);
        *p = NULL;
    }

}

void test(){

    char* p = NULL;

    AllocateSpace(&p);
    printf("%s\n",p);
    FreeSpace(&p);

    if (p == NULL){
        printf("p 内存释放!\n");
    }
}
```

4.3.4 间接赋值的推论

- 用 1 级指针形参，去间接修改了 0 级指针(实参)的值。
- 用 2 级指针形参，去间接修改了 1 级指针(实参)的值。
- 用 3 级指针形参，去间接修改了 2 级指针(实参)的值。
- 用 n 级指针形参，去间接修改了 n-1 级指针(实参)的值。

4.3 指针做函数参数

指针做函数参数，具备输入和输出特性：

- 输入：主调函数分配内存

- 输出：被调用函数分配内存

4.3.1 输入特性

```
void fun(char *p /* in */)
{
    //给 p 指向的内存区域拷贝内容
    strcpy(p, "abcddsgsd");
}

void test(void)
{
    //输入，主调函数分配内存
    char buf[100] = { 0 };
    fun(buf);
    printf("buf  = %s\n", buf);
}
```

4.3.2 输出特性

```
void fun(char **p /* out */, int *len)
{
    char *tmp = (char *)malloc(100);
    if (tmp == NULL)
    {
        return;
    }
    strcpy(tmp, "adlsgjldsk");

    //间接赋值
    *p = tmp;
    *len = strlen(tmp);
}

void test(void)
{
    //输出，被调用函数分配内存，地址传递
    char *p = NULL;
    int len = 0;
    fun(&p, &len);
```

```
    if (p != NULL)
    {
        printf("p = %s, len = %d\n", p, len);
    }
```

4.4 字符串指针强化

4.4.1 字符串指针做函数参数

4.4.1.1 字符串基本操作

```
//字符串基本操作
//字符串是以 0 或者'\0' 0 NULL 结尾的字符数组，(数字 0 和字符'\0'等价)
void test01(){

    //字符数组只能初始化 5 个字符，当输出的时候，从开始位置直到找到 0 结束
    char str1[] = { 'h', 'e', 'l', 'l', 'o' };
    printf("%s\n",str1);

    //字符数组部分初始化，剩余填 0
    char str2[100] = { 'h', 'e', 'l', 'l', 'o' };
    printf("%s\n", str2);

    //如果以字符串初始化，那么编译器默认会在字符串尾部添加'\0'
    char str3[] = "hello";
    printf("%s\n",str3);
    printf("sizeof str:%d\n",sizeof(str3));
    printf("strlen str:%d\n",strlen(str3));

    //sizeof 计算数组大小，数组包含'\0'字符
    //strlen 计算字符串的长度，到'\0'结束

    //那么如果我这么写,结果是多少呢?
    char str4[100] = "hello";
    printf("sizeof str:%d\n", sizeof(str4));
    printf("strlen str:%d\n", strlen(str4));

    //请问下面输入结果是多少? sizeof 结果是多少? strlen 结果是多少?
    char str5[] = "hello\0world";
    printf("%s\n",str5);
```

```
    printf("sizeof str5:%d\n",sizeof(str5));
    printf("strlen str5:%d\n",strlen(str5));

    //再请问下面输入结果是多少？sizeof 结果是多少？strlen 结果是多少？
    char str6[] = "hello\012world";
    printf("%s\n", str6);
    printf("sizeof str6:%d\n", sizeof(str6));
    printf("strlen str6:%d\n", strlen(str6));
}
```

八进制和十六进制转义字符：

在 C 中有两种特殊的字符，八进制转义字符和十六进制转义字符，八进制字符的一般形式是'\ddd'，d 是 0-7 的数字。十六进制字符的一般形式是'\xhh'，h 是 0-9 或 A-F 内的一个。八进制字符和十六进制字符表示的是字符的 ASCII 码对应的数值。

比如 ：

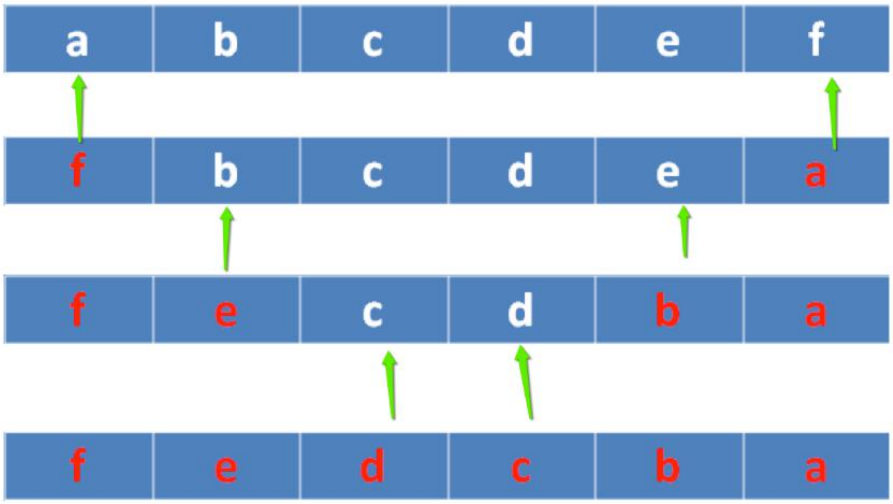
- '\063'表示的是字符'3'，因为'3'的 ASCII 码是 30（十六进制），48（十进制），63（八进制）。
- '\x41'表示的是字符'A'，因为'A'的 ASCII 码是 41（十六进制），65（十进制），101（八进制）。

4.4.1.2 字符串拷贝功能实现

```
//拷贝方法 1
void copy_string01(char* dest, char* source ){

    for (int i = 0; source[i] != '\0';i++){
        dest[i] = source[i];
    }

}

//拷贝方法 2
```

```c
void copy_string02(char* dest, char* source){
    while (*source != '\0' /* *source != 0 */){
        *dest = *source;
        source++;
        dest++;
    }
}

//拷贝方法 3
void copy_string03(char* dest, char* source){
    //判断*dest 是否为 0，0 则退出循环
    while (*dest++ = *source++){}
}

//拷贝方法 4
//1）应该判断下传入的参数是否为 NULL
//2）最好不要直接使用形参
int copy_string04(char* dest, char* source){
    if (dest == NULL){
        return -1;
    }
    if (source == NULL){
        return -2;
    }

    char* src = source;
    char* tar = dest;

    while (*tar++ = *src++){}

    return 0;
}
```

4.4.1.3 字符串反转模型




```
void reverse_string(char* str){

    if (str == NULL){
        return;
    }

    int begin = 0;
    int end = strlen(str) - 1;

    while (begin < end){

        //交换两个字符元素
        char temp = str[begin];
        str[begin] = str[end];
        str[end] = temp;

        begin++;
        end--;
    }

}

void test(){
    char str[] = "abcdefghijklmn";
    printf("str:%s\n", str);
    reverse_string(str);
```

```
    printf("str:%s\n", str);
}
```

4.4.2 字符串的格式化

4.4.2.1 sprintf

```
#include <stdio.h>
int sprintf(char *str, const char *format, ...);
功能：
    根据参数 format 字符串来转换并格式化数据，然后将结果输出到 str 指定的空间中，直到出现字符串结束符 '\0' 为止。
参数：
    str：字符串首地址
    format：字符串格式，用法和 printf()一样
返回值：
    成功：实际格式化的字符个数
    失败： - 1
```

```
void test(){

    //1. 格式化字符串
    char buf[1024] = { 0 };
    sprintf(buf, "你好,%s,欢迎加入我们!", "John");
    printf("buf:%s\n",buf);

    memset(buf, 0, 1024);
    sprintf(buf, "我今年%d 岁了!", 20);
    printf("buf:%s\n", buf);

    //2. 拼接字符串
    memset(buf, 0, 1024);
    char str1[] = "hello";
    char str2[] = "world";
    int len = sprintf(buf,"%s %s",str1,str2);
    printf("buf:%s len:%d\n", buf,len);

    //3. 数字转字符串
    memset(buf, 0, 1024);
    int num = 100;
```

```
    sprintf(buf, "%d", num);
    printf("buf:%s\n", buf);
    //设置宽度 右对齐
    memset(buf, 0, 1024);
    sprintf(buf, "%8d", num);
    printf("buf:%s\n", buf);
    //设置宽度 左对齐
    memset(buf, 0, 1024);
    sprintf(buf, "%-8d", num);
    printf("buf:%s\n", buf);
    //转成 16 进制字符串 小写
    memset(buf, 0, 1024);
    sprintf(buf, "0x%x", num);
    printf("buf:%s\n", buf);

    //转成 8 进制字符串
    memset(buf, 0, 1024);
    sprintf(buf, "0%o", num);
    printf("buf:%s\n", buf);
}
```

4.4.2.2 sscanf

```
#include <stdio.h>
int sscanf(const char *str, const char *format, ...);
功能:
    从 str 指定的字符串读取数据，并根据参数 format 字符串来转换并格式化数据。
参数:
    str: 指定的字符串首地址
    format: 字符串格式，用法和 scanf()一样
返回值:
    成功: 实际读取的字符个数
    失败: - 1
```

格式	作用
%*s 或%*d	跳过数据
%[width]s	读指定宽度的数据
%[a-z]	匹配 a 到 z 中任意字符(尽可能多的匹配)
%[aBc]	匹配 a、B、c 中一员，贪婪性

格式	作用
%[^a]	匹配非 a 的任意字符，贪婪性
%[^a-z]	表示读取除 a-z 以外的所有字符

```
//1. 跳过数据
void test01(){
    char buf[1024] = { 0 };
    //跳过前面的数字
    //匹配第一个字符是否是数字，如果是，则跳过
    //如果不是则停止匹配
    sscanf("123456aaaa", "%*d%s", buf);
    printf("buf:%s\n",buf);
}

//2. 读取指定宽度数据
void test02(){
    char buf[1024] = { 0 };
    //跳过前面的数字
    sscanf("123456aaaa", "%7s", buf);
    printf("buf:%s\n", buf);
}

//3. 匹配 a-z 中任意字符
void test03(){
    char buf[1024] = { 0 };
    //跳过前面的数字
    //先匹配第一个字符，判断字符是否是 a-z 中的字符，如果是匹配
    //如果不是停止匹配
    sscanf("abcdefg123456", "%[a-z]", buf);
    printf("buf:%s\n", buf);
}

//4. 匹配 aBc 中的任何一个
void test04(){
    char buf[1024] = { 0 };
    //跳过前面的数字
    //先匹配第一个字符是否是 aBc 中的一个，如果是，则匹配，如果不是则停止匹配
    sscanf("abcdefg123456", "%[aBc]", buf);
    printf("buf:%s\n", buf);
}

//5. 匹配非 a 的任意字符
```

```
void test05(){
    char buf[1024] = { 0 };
    //跳过前面的数字
    //先匹配第一个字符是否是 aBc 中的一个，如果是，则匹配，如果不是则停止匹配
    sscanf("bcdefag123456", "%[^a]", buf);
    printf("buf:%s\n", buf);
}

//6. 匹配非 a-z 中的任意字符
void test06(){
    char buf[1024] = { 0 };
    //跳过前面的数字
    //先匹配第一个字符是否是 aBc 中的一个，如果是，则匹配，如果不是则停止匹配
    sscanf("123456ABCDbcdefag", "%[^a-z]", buf);
    printf("buf:%s\n", buf);
}
```

课堂小练习：

1. 已给定字符串为: helloworld@itcast.cn,请编码实现 helloworld 输出和 itcast.cn 输出。.
2. 已给定字符串为:123abcd$myname@000qwe.请编码实现匹配出 myname 字符串，并输出.

4.5 一级指针易错点

4.5.1 越界

```
void test(){

    char buf[3] = "abc";

    printf("buf:%s\n",buf);

}
```

4.5.2 指针叠加会不断改变指针指向

```
void test(){

    char *p = (char *)malloc(50);

    char buf[] = "abcdef";

    int n = strlen(buf);

    int i = 0;



    for (i = 0; i < n; i++)

    {

        *p = buf[i];

        p++; //修改原指针指向

    }



    free(p);

}
```

4.5.3 返回局部变量地址

```
char *get_str()

{

    char str[] = "abcdedsgads"; //栈区,

    printf("[get_str]str = %s\n", str);

    return str;

}
```

4.5.4 同一块内存释放多次

```
void test(){

    char *p = NULL;



    p = (char *)malloc(50);

    strcpy(p, "abcdef");



    if (p != NULL)

    {

        //free()函数的功能只是告诉系统 p 指向的内存可以回收了

        // 就是说，p 指向的内存使用权交还给系统

        //但是，p 的值还是原来的值(野指针)，p 还是指向原来的内存

        free(p);

    }



    if (p != NULL)

    {

        free(p);

    }

}
```

4.6 const 使用

```
//const修饰变量
void test01(){
    //1. const基本概念
    const int i = 0;
    //i = 100; //错误，只读变量初始化之后不能修改

    //2. 定义const变量最好初始化
    const int j;
    //j = 100; //错误，不能再次赋值

    //3. c语言的const是一个只读变量，并不是一个常量，可通过指针间接修改
    const int k = 10;
    //k = 100; //错误，不可直接修改，我们可通过指针间接修改
    printf("k:%d\n", k);
    int* p = &k;
    *p = 100;
    printf("k:%d\n", k);
}

//const 修饰指针
void test02(){

    int a = 10;
    int b = 20;
    //const放在*号左侧 修饰p_a指针指向的内存空间不能修改,但可修改指针的指向
    const int* p_a = &a;
    //*p_a = 100; //不可修改指针指向的内存空间
    p_a = &b; //可修改指针的指向

    //const放在*号的右侧， 修饰指针的指向不能修改，但是可修改指针指向的内存空间
    int* const p_b = &a;
    //p_b = &b; //不可修改指针的指向
    *p_b = 100; //可修改指针指向的内存空间

    //指针的指向和指针指向的内存空间都不能修改
    const int* const p_c = &a;
}
//const指针用法
struct Person{
    char name[64];
    int id;
```

```
    int age;
    int score;
};

//每次都对对象进行拷贝，效率低，应该用指针
void printPersonByValue(struct Person person){
    printf("Name:%s\n", person.name);
    printf("Name:%d\n", person.id);
    printf("Name:%d\n", person.age);
    printf("Name:%d\n", person.score);
}

//但是用指针会有副作用，可能会不小心修改原数据
void printPersonByPointer(const struct Person *person){
    printf("Name:%s\n", person->name);
    printf("Name:%d\n", person->id);
    printf("Name:%d\n", person->age);
    printf("Name:%d\n", person->score);
}
void test03(){
    struct Person p = { "Obama", 1101, 23, 87 };
    //printPersonByValue(p);
    printPersonByPointer(&p);
}
```

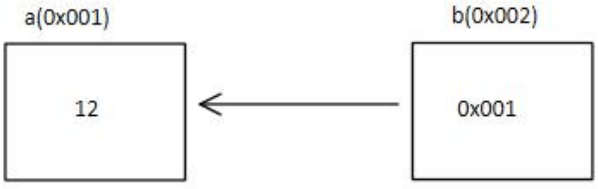
5. 指针的指针(二级指针)

5.1 二级指针基本概念

这里让我们花点时间来看一个例子，揭开这个即将开始的序幕。考虑下面这些声明：

```
int a = 12;
int *b = &a;
```

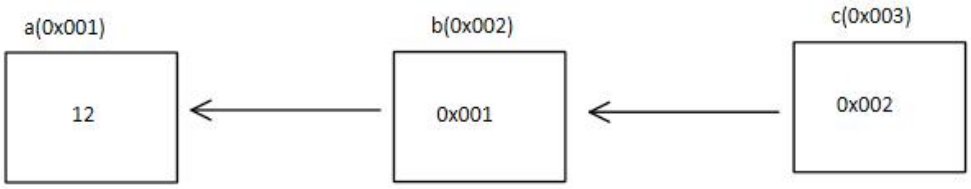
它们如下图进行内存分配：



假定我们又有了第 3 个变量，名叫 c,并用下面这条语句对它进行初始化：

```
c = &b;
```

它在内存中的大概模样大致如下：



问题是 :c 的类型是什么？显然它是一个指针 ,但它所指向的是什么？变量 b 是一个“指向整型的指针”，所以任何指向 b 的类型必须是指向“指向整型的指针”的指针，更通俗地说，是一个指针的指针。

它合法吗？是的！指针变量和其他变量一样，占据内存中某个特定的位置，所以用&操作符取得它的地址是合法的。

那么这个变量的声明是怎样的声明的呢？

```
int **c = &b;
```

那么这个**c 如何理解呢？*操作符具有从右想做的结合性，所以这个表达式相当于*(*c),我们从里向外逐层求职。*c 访问 c 所指向的位置，我们知道这是变量 b.第二个间接访问操作符访问这个位置所指向的地址，也就是变量 a.指针的指针并不难懂，只需要留心所有的箭头，如果表达式中出现了间接访问操作符，你就要随箭头访问它所指向的位置。

5.2 二级指针做形参输出特性

二级指针做参数的输出特性是指由被调函数分配内存。

```
//被调函数,由参数n确定分配多少个元素内存
void allocate_space(int **arr,int n){
    //堆上分配n个int类型元素内存
    int *temp = (int *)malloc(sizeof(int)* n);
    if (NULL == temp){
```

```
        return;
    }
    //给内存初始化值
    int *pTemp = temp;
    for (int i = 0; i < n;i ++){
        //temp[i] = i + 100;
        *pTemp = i + 100;
        pTemp++;
    }
    //指针间接赋值
    *arr = temp;
}
//打印数组
void print_array(int *arr,int n){
    for (int i = 0; i < n;i ++){
        printf("%d ",arr[i]);
    }
    printf("\n");
}
//二级指针输出特性(由被调函数分配内存)
void test(){
    int *arr = NULL;
    int n = 10;
    //给arr指针间接赋值
    allocate_space(&arr,n);
    //输出arr指向数组的内存
    print_array(arr, n);
    //释放arr所指向内存空间的值
    if (arr != NULL){
        free(arr);
        arr = NULL;
    }
}
```

5.3 二级指针做形参输入特性

二级指针做形参输入特性是指由主调函数分配内存。

```
//打印数组
void print_array(int **arr,int n){
    for (int i = 0; i < n;i ++){
        printf("%d ",*(arr[i]));
    }
```

```
    printf("\n");
}
//二级指针输入特性(由主调函数分配内存)
void test(){

    int a1 = 10;
    int a2 = 20;
    int a3 = 30;
    int a4 = 40;
    int a5 = 50;

    int n = 5;

    int** arr = (int **)malloc(sizeof(int *) * n);
    arr[0] = &a1;
    arr[1] = &a2;
    arr[2] = &a3;
    arr[3] = &a4;
    arr[4] = &a5;

    print_array(arr,n);

    free(arr);
    arr = NULL;
}
```

5.4 强化训练_画出内存模型图

```
void mian()
{
    //栈区指针数组
    char *p1[] = { "aaaaa", "bbbbb", "ccccc" };

    //堆区指针数组
    char **p3 = (char **)malloc(3 * sizeof(char *)); //char *array[3];

    int i = 0;
    for (i = 0; i < 3; i++)
    {
        p3[i] = (char *)malloc(10 * sizeof(char)); //char buf[10]
        sprintf(p3[i], "%d%d%d", i, i, i);
```

```
    }
}
```

5.4 多级指针

将堆区数组指针案例改为三级指针案例：

```
//分配内存
void allocate_memory(char*** p, int n){

    if (n < 0){
        return;
    }

    char** temp = (char**)malloc(sizeof(char*)* n);
    if (temp == NULL){
        return;
    }

    //分别给每一个指针 malloc 分配内存
    for (int i = 0; i < n; i++){
        temp[i] = malloc(sizeof(char)* 30);
        sprintf(temp[i], "%2d_hello world!", i + 1);
    }

    *p = temp;
}

//打印数组
void array_print(char** arr, int len){
    for (int i = 0; i < len; i++){
        printf("%s\n", arr[i]);
    }
    printf("----------------------\n");
}

//释放内存
void free_memory(char*** buf, int len){
    if (buf == NULL){
        return;
    }
```

```
    char** temp = *buf;

    for (int i = 0; i < len; i++){
        free(temp[i]);
        temp[i] = NULL;
    }

    free(temp);
}

void test(){

    int n = 10;
    char** p = NULL;
    allocate_memory(&p, n);
    //打印数组
    array_print(p, n);
    //释放内存
    free_memory(&p, n);
}
```

6.位运算

可以使用 C 对变量中的个别位进行操作。您可能对人们想这样做的原因感到奇怪。这种能力有时确实是必须的，或者至少是有用的。C 提供位的**逻辑运算符**和**移位运算符**。在以下例子中，我们将使用二进制计数法写出值，以便您可以了解对位发生的操作。在一个实际程序中，您可以使用一般的形式的整数变量或常量。例如不适用 00011001 的形式，而写为 25 或者 031 或者 0x19.在我们的例子中，我们将使用 8 位数字，从左到右，每位的编号是 7 到 0。

6.1 位逻辑运算符

4 个位运算符用于整型数据，包括 char.将这些位运算符成为位运算的原因是它们对每

位进行操作，而不影响左右两侧的位。请不要将这些运算符与常规的逻辑运算符(&& 、|| 和!)相混淆，常规的位的逻辑运算符对整个值进行操作。

6.1.1 按位取反~

一元运算符~将每个 1 变为 0，将每个 0 变为 1，如下面的例子：

```
~(10011010)
01100101
```

假设 a 是一个 unsigned char，已赋值为 2.在二进制中，2 是 00000010.于是-a 的值为 11111101 或者 253。请注意该运算符不会改变 a 的值，a 仍为 2。

```
unsigned char a = 2;   //00000010
unsigned char b = ~a;  //11111101
printf("ret = %d\n", a); //ret = 2
printf("ret = %d\n", b); //ret = 253
```

6.1.2 位与（AND）：&

二进制运算符&通过对两个操作数逐位进行比较产生一个新值。对于每个位，只有两个操作数的对应位都是 1 时结果才为 1。

```
  (10010011)
& (00111101)
= (00010001)
```

C 也有一个组合的位与-赋值运算符：&=。下面两个将产生相同的结果：

```
val &= 0377
val = val & 0377
```

例如：一个数 &1 的结果就是取二进制的最末位。这可以用来判断一个整数的奇偶，二进制的最末位为 0 表示该数为偶数，最末位为 1 表示该数为奇数。

6.1.3 位或（OR）：|

二进制运算符|通过对两个操作数逐位进行比较产生一个新值。对于每个位，如果其中任意操作数中对应的位为 1，那么结果位就为 1.

```
   (10010011)
 | (00111101)
 = (10111111)
```

C 也有组合位或-赋值运算符： |=

```
val |= 0377
val = val | 0377
```

or 运算通常用于二进制特定位上的无条件赋值，例如一个数 or 1 的结果就是把二进制最末位强行变成 1。如果需要把二进制最末位变成 0，对这个数 or 1 之后再减一就可以了。

6.1.4 位异或:

二进制运算符^对两个操作数逐位进行比较。对于每个位，如果操作数中的对应位有一个是 1(但不是都是 1)，那么结果是 1.如果都是 0 或者都是 1，则结果位 0.

```
   (10010011)
 ^ (00111101)
 = (10101110)
```

C 也有一个组合的位异或-赋值运算符： ^=

```
val ^= 0377
val = val ^ 0377
```

6.1.5 用法

6.1.5.1 打开位

已知：10011010：

1. 将位 2 打开

   flag | 10011010

```
(10011010)
|(00000100)
=(10011110)
```

2. 将所有位打开。

   flag | ~flag

```
(10011010)
|(01100101)
=(11111111)
```

6.1.5.2 关闭位

flag & ~flag

```
(10011010)
&(01100101)
=(00000000)
```

6.1.5.3 转置位

转置(toggling)一个位表示如果该位打开，则关闭该位；如果该位关闭，则打开。您可以使用位异或运算符来转置。其思想是如果 b 是一个位(1 或 0)，那么如果 b 为 1 则 b^1 为 0，如果 b 为 0，则 1^b 为 1。无论 b 的值是 0 还是 1,0^b 为 b.

flag ^ 0xff

```
(10010011)
^(11111111)
```

```
=(01101100)
```

6.1.5.4 交换两个数不需要临时变量

```
//a ^ b = temp;
//a ^ temp = b;
//b ^ temp = a
 (10010011)
^(00100110)
=(10110101)

 (10110101)
^(00100110)
  10010011

  int a = 10;
  int b = 30;
```

6.2 移位运算符

现在让我们了解一下 C 的移位运算符。移位运算符将位向左或向右移动。同样，我们仍将明确地使用二进制形式来说明该机制的工作原理。

6.2.1 左移 <<

左移运算符<<将其左侧操作数的值的每位向左移动 ,移动的位数由其右侧操作数指定。空出来的位用 0 填充，并且丢弃移出左侧操作数末端的位。在下面例子中，每位向左移动两个位置。

```
(10001010) << 2
(00101000)
```

该操作将产生一个新位置，但是不改变其操作数。

```
1 << 1 = 2;
2 << 1 = 4;
4 << 1 = 8;
8 << 2 = 32
```

左移一位相当于原值*2.

6.2.2 右移 >>

右移运算符>>将其左侧的操作数的值每位向右移动，移动的位数由其右侧的操作数指定。丢弃移出左侧操作数有段的位。对于 unsigned 类型，使用 0 填充左端空出的位。**对于有符号类型，结果依赖于机器。空出的位可能用 0 填充，或者使用符号(最左端)位的副本填充。**

```
//有符号值
(10001010) >> 2
(00100010)     //在某些系统上的结果值

(10001010) >> 2
(11100010)     //在另一些系统上的解雇

//无符号值
(10001010) >> 2
(00100010)    //所有系统上的结果值
```

6.2.3 用法：移位运算符

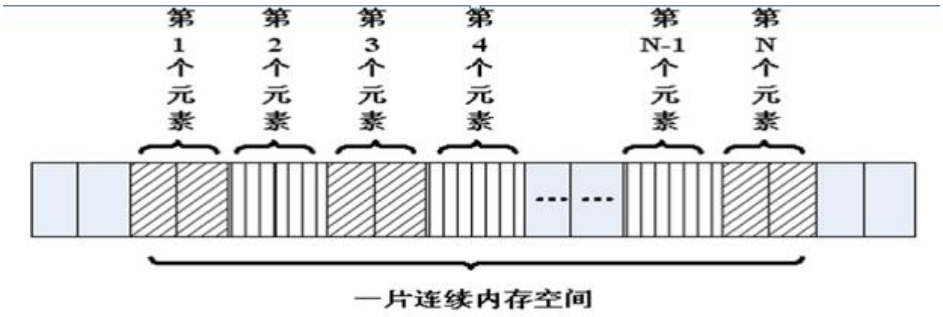
移位运算符能够提供快捷、高效（依赖于硬件）对 2 的幂的乘法和除法。

number << n	number 乘以 2 的 n 次幂
number >> n	如果 number 非负，则用 number 除以 2 的 n 次幂

6. 多维数组

6.1 一维数组

- 元素类型角度：数组是相同类型的变量的有序集合
- 内存角度：连续的一大片内存空间



在讨论多维数组之前，我们还需要学习很多关于一维数组的知识。首先让我们学习一个概念。

6.1.1 数组名

考虑下面这些声明：

```
int a;
int b[10];
```

我们把 a 称作标量，因为它是个单一的值，这个变量是的类型是一个整数。我们把 b 称作数组，因为它是一些值的集合。下标和数名一起使用，用于标识该集合中某个特定的值。例如，b[0]表示数组 b 的第 1 个值，b[4]表示第 5 个值。每个值都是一个特定的标量。

那么问题是 b 的类型是什么？它所表示的又是什么？一个合乎逻辑的答案是它表示整个数组，但事实并非如此。在 C 中，在几乎所有数组名的表达式中，数组名的值是一个**指针常量**，也就是数组第一个元素的地址。它的类型取决于数组元素的类型：如果他们是 int

类型，那么数组名的类型就是“指向 int 的常量指针”；如果它们是其他类型，那么数组名的类型也就是“指向**其他类型**的常量指针”。

请问：指针和数组是等价的吗？

答案是**否定**的。数组名在表达式中使用的时候，编译器才会产生一个指针常量。那么数组在什么情况下不能作为指针常量呢？在以下两种场景下：

- 当数组名作为 sizeof 操作符的操作数的时候，此时 sizeof 返回的是整个数组的长度，而不是指针数组指针的长度。
- 当数组名作为&操作符的操作数的时候，此时返回的是一个指向数组的指针，而不是指向某个数组元素的指针常量。

```
int arr[10];
//arr = NULL; //arr作为指针常量，不可修改
int *p = arr; //此时arr作为指针常量来使用
printf("sizeof(arr):%d\n", sizeof(arr)); //此时sizeof结果为整个数组的长度
printf("&arr type is %s\n", typeid(&arr).name()); //int(*)[10]而不是 int*
```

6.1.2 下标引用

```
int arr[] = { 1, 2, 3, 4, 5, 6 };
```

***(arr + 3)**,这个表达式是什么意思呢？

首先，我们说数组在表达式中是一个指向整型的指针，所以此表达式表示 arr 指针向后移动了 3 个元素的长度。然后通过间接访问操作符从这个新地址开始获取这个位置的值。这个和下标的引用的执行过程完全相同。所以如下表达式是等同的：

```
*(arr + 3)
arr[3]
```

问题 1：数组下标可否为负值？

问题 2：请阅读如下代码，说出结果：

```
int arr[] = { 5, 3, 6, 8, 2, 9 };
int *p = arr + 2;
printf("*p = %d\n", *p);
printf("*p = %d\n", p[-1]);
```

那么是用下标还是指针来操作数组呢？对于大部分人而言，下标的可读性会强一些。

6.1.3 数组和指针

指针和数组并不是相等的。为了说明这个概念，请考虑下面两个声明：

```
int a[10];
int *b;
```

声明一个数组时，编译器根据声明所指定的元素数量为数组分配内存空间，然后再创建数组名，指向这段空间的起始位置。声明一个指针变量的时候，编译器只为指针本身分配内存空间，并不为任何整型值分配内存空间，指针并未初始化指向任何现有的内存空间。

因此，表达式*a 是完全合法的，但是表达式*b 却是非法的。*b 将访问内存中一个不确定的位置，将会导致程序终止。另一方面 b++可以通过编译，a++却不行，因为 a 是一个常量值。

6.1.4 作为函数参数的数组名

当一个数组名作为一个参数传递给一个函数的时候发生什么情况呢？我们现在知道数组名其实就是一个指向数组第 1 个元素的指针，所以很明白此时传递给函数的是一份指针

的拷贝。所以函数的形参实际上是一个指针。但是为了使程序员新手容易上手一些，编译器也接受数组形式的函数形参。因此下面两种函数原型是相等的：

```
int print_array(int *arr);
int print_array(int arr[]);
```

我们可以使用任何一种声明，但哪一个更准确一些呢？答案是指针。因为实参实际上是个指针，而不是数组。同样 sizeof arr 值是指针的长度，而不是数组的长度。

现在我们清楚了 ,为什么一维数组中无须写明它的元素数目了 ,因为形参只是一个指针，并不需要为数组参数分配内存。另一方面，这种方式使得函数无法知道数组的长度。如果函数需要知道数组的长度，它必须显式传递一个长度参数给函数。

6.2 多维数组

如果某个数组的维数不止 1 个，它就被称为多维数组。接下来的案例讲解以二维数组举例。

```
void test01(){
    //二维数组初始化
    int arr1[3][3] = {
        { 1, 2, 3 },
        { 4, 5, 6 },
        { 7, 8, 9 }
    };

    int arr2[3][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    int arr3[][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    //打印二维数组
    for (int i = 0; i < 3; i++){
        for (int j = 0; j < 3; j ++){
            printf("%d ",arr1[i][j]);
        }
        printf("\n");
    }
}
```

6.2.1 数组名

一维数组名的值是一个指针常量，它的类型是“指向元素类型的指针”，它指向数组的第 1 个元素。多维数组也是同理，多维数组的数组名也是指向第一个元素，只不过第一个元素是一个数组。例如：

```
int arr[3][10]
```

可以理解为这是一个一维数组，包含了 3 个元素，只是每个元素恰好是包含了 10 个元素的数组。arr 就表示指向它的第 1 个元素的指针，所以 arr 是一个指向了包含了 10 个整型元素的数组的指针。

6.2.2 指向数组的指针(数组指针)

数组指针，它是指针，指向数组的指针。

数组的类型由**元素类型**和**数组大小**共同决定：int array[5]　的类型为　int[5]；

C 语言可通过 typedef 定义一个数组类型：

定义数组指针有一下三种方式：

```
//方式一
void test01(){

    //先定义数组类型，再用数组类型定义数组指针
    int arr[10] = {1,2,3,4,5,6,7,8,9,10};
    //有 typedef 是定义类型，没有则是定义变量,下面代码定义了一个数组类型 ArrayType
    typedef int(ArrayType)[10];
    //int ArrayType[10]; //定义一个数组，数组名为 ArrayType

    ArrayType myarr; //等价于 int myarr[10];
    ArrayType* pArr = &arr; //定义了一个数组指针 pArr，并且指针指向数组 arr
    for (int i = 0; i < 10;i++){
        printf("%d ",(*pArr)[i]);
    }
    printf("\n");
```

```
}

//方式二
void test02(){

    int arr[10];
    //定义数组指针类型
    typedef int(*ArrayType)[10];
    ArrayType pArr = &arr; //定义了一个数组指针 pArr，并且指针指向数组 arr
    for (int i = 0; i < 10; i++){
        (*pArr)[i] = i + 1;
    }
    for (int i = 0; i < 10; i++){
        printf("%d ", (*pArr)[i]);
    }
    printf("\n");

}

//方式三
void test03(){

    int arr[10];
    int(*pArr)[10] = &arr;

    for (int i = 0; i < 10; i++){
        (*pArr)[i] = i + 1;

    }
    for (int i = 0; i < 10; i++){
        printf("%d ", (*pArr)[i]);
    }
    printf("\n");
}
```

6.2.3 指针数组(元素为指针)

5.3.1 栈区指针数组

```
//数组做函数函数，退化为指针
void array_sort(char** arr,int len){

```

```
    for (int i = 0; i < len; i++){
        for (int j = len - 1; j > i; j --){
            //比较两个字符串
            if (strcmp(arr[j-1],arr[j]) > 0){
                char* temp = arr[j - 1];
                arr[j - 1] = arr[j];
                arr[j] = temp;
            }
        }
    }

}

//打印数组
void array_print(char** arr,int len){
    for (int i = 0; i < len;i++){
        printf("%s\n",arr[i]);
    }
    printf("----------------------\n");
}

void test(){

    //主调函数分配内存
    //指针数组
    char* p[] = { "bbb", "aaa", "ccc", "eee", "ddd"};
    //char** p = { "aaa", "bbb", "ccc", "ddd", "eee" }; //错误
    int len = sizeof(p) / sizeof(char*);
    //打印数组
    array_print(p, len);
    //对字符串进行排序
    array_sort(p, len);
    //打印数组
    array_print(p, len);
}
```

5.3.2 堆区指针数组

```
//分配内存
char** allocate_memory(int n){

    if (n < 0 ){
        return NULL;
```

```
    }

    char** temp = (char**)malloc(sizeof(char*) * n);
    if (temp == NULL){
        return NULL;
    }

    //分别给每一个指针 malloc 分配内存
    for (int i = 0; i < n; i ++){
        temp[i] = malloc(sizeof(char)* 30);
        sprintf(temp[i], "%2d_hello world!", i + 1);
    }

    return temp;
}

//打印数组
void array_print(char** arr,int len){
    for (int i = 0; i < len;i++){
        printf("%s\n",arr[i]);
    }
    printf("----------------------\n");
}

//释放内存
void free_memory(char** buf,int len){
    if (buf == NULL){
        return;
    }
    for (int i = 0; i < len; i ++){
        free(buf[i]);
        buf[i] = NULL;
    }

    free(buf);
}

void test(){

    int n = 10;
    char** p = allocate_memory(n);
    //打印数组
    array_print(p, n);
    //释放内存
```

```
    free_memory(p, n);
}
```

6.2.4 二维数组三种参数形式

6.2.4.1 二维数组的线性存储特性

```
void PrintArray(int* arr, int len){
    for (int i = 0; i < len; i++){
        printf("%d ", arr[i]);
    }
    printf("\n");
}

//二维数组的线性存储
void test(){
    int arr[][3] = {
        { 1, 2, 3 },
        { 4, 5, 6 },
        { 7, 8, 9 }
    };

    int arr2[][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    int len = sizeof(arr2) / sizeof(int);

    //如何证明二维数组是线性的?
    //通过将数组首地址指针转成Int*类型，那么步长就变成了4，就可以遍历整个数组
    int* p = (int*)arr;
    for (int i = 0; i < len; i++){
        printf("%d ", p[i]);
    }
    printf("\n");

    PrintArray((int*)arr, len);
    PrintArray((int*)arr2, len);
}
```

6.2.4.2 二维数组的 3 种形式参数

```c
//二维数组的第一种形式
void PrintArray01(int arr[3][3]){
    for (int i = 0; i < 3; i++){
        for (int j = 0; j < 3; j++){
            printf("arr[%d][%d]:%d\n", i, j, arr[i][j]);
        }
    }
}

//二维数组的第二种形式
void PrintArray02(int arr[][3]){
    for (int i = 0; i < 3; i++){
        for (int j = 0; j < 3; j++){
            printf("arr[%d][%d]:%d\n", i, j, arr[i][j]);
        }
    }
}

//二维数组的第二种形式
void PrintArray03(int(*arr)[3]){
    for (int i = 0; i < 3; i++){
        for (int j = 0; j < 3; j++){
            printf("arr[%d][%d]:%d\n", i, j, arr[i][j]);
        }
    }
}

void test(){

    int arr[][3] = {
        { 1, 2, 3 },
        { 4, 5, 6 },
        { 7, 8, 9 }
    };

    PrintArray01(arr);
    PrintArray02(arr);
    PrintArray03(arr);
}
```

6.3 总结

6.3.1 编程提示

- 源代码的可读性几乎总是比程序的运行时效率更为重要
- 只要有可能，函数的指针形参都应该声明为 const
- 在多维数组的初始值列表中使用完整的多层花括号提供可读性

6.3.2 内容总结

在绝大多数表达式中，数组名的值是指向数组第 1 个元素的指针。这个规则只有两个例外，sizeof 和对数组名&。

指针和数组并不相等。当我们声明一个数组的时候，同时也分配了内存。但是声明指针的时候，只分配容纳指针本身的空间。

当数组名作为函数参数时，实际传递给函数的是一个指向数组第 1 个元素的指针。

我们不单可以创建指向普通变量的指针，也可创建指向数组的指针。

7. 结构体

7.1 结构体基础知识

7.1.1 结构体类型的定义

```
struct Person{
    char name[64];
    int age;
};

typedef struct _PERSON{
```

```
    char name[64];
    int age;
}Person;
```

注意：定义结构体类型时不要直接给成员赋值，结构体只是一个类型，编译器还没有为其分配空间，只有根据其类型定义变量时，才分配空间，有空间后才能赋值。

7.1.2 结构体变量的定义

```
struct Person{
    char name[64];
    int age;
}p1; //定义类型同时定义变量


struct{
    char name[64];
    int age;
}p2; //定义类型同时定义变量


struct Person p3; //通过类型直接定义
```

7.1.3 结构体变量的初始化

```
struct Person{
    char name[64];
    int age;
}p1 = {"john",10}; //定义类型同时初始化变量

struct{
    char name[64];
    int age;
}p2 = {"Obama",30}; //定义类型同时初始化变量

struct Person p3 = {"Edward",33}; //通过类型直接定义
```

7.1.4 结构体成员的使用

```
struct Person{
    char name[64];
    int age;
};
void test(){
    //在栈上分配空间
    struct Person p1;
    strcpy(p1.name, "John");
    p1.age = 30;
    //如果是普通变量，通过点运算符操作结构体成员
    printf("Name:%s Age:%d\n", p1.name, p1.age);

    //在堆上分配空间
    struct Person* p2 = (struct Person*)malloc(sizeof(struct Person));
    strcpy(p2->name, "Obama");
    p2->age = 33;
    //如果是指针变量，通过->操作结构体成员
    printf("Name:%s Age:%d\n", p2->name, p2->age);
}
```

7.1.5 结构体赋值

7.1.5.1 赋值基本概念

相同的两个结构体变量可以相互赋值，把一个结构体变量的值拷贝给另一个结构体，这两个变量还是两个独立的变量。

```
struct Person{
    char name[64];
    int age;
};

void test(){
    //在栈上分配空间
    struct Person p1 = { "John" , 30};
    struct Person p2 = { "Obama", 33 };
    printf("Name:%s Age:%d\n", p1.name, p1.age);
```

```
    printf("Name:%s Age:%d\n", p2.name, p2.age);
    //将 p2 的值赋值给 p1
    p1 = p2;
    printf("Name:%s Age:%d\n", p1.name, p1.age);
    printf("Name:%s Age:%d\n", p2.name, p2.age);
}
```

7.1.5.1 深拷贝和浅拷贝

```
//一个老师有 N 个学生
typedef struct _TEACHER{
    char* name;
}Teacher;


void test(){

    Teacher t1;
    t1.name = malloc(64);
    strcpy(t1.name , "John");

    Teacher t2;
    t2 = t1;

    //对手动开辟的内存，需要手动拷贝
    t2.name = malloc(64);
    strcpy(t2.name, t1.name);

    if (t1.name != NULL){
        free(t1.name);
        t1.name = NULL;
    }
    if (t2.name != NULL){
        free(t2.name);
        t1.name = NULL;
    }
}
```

7.1.6 结构体数组

```
struct Person{
```

```
    char name[64];
    int age;
};

void test(){
    //在栈上分配空间
    struct Person p1[3] = {
        { "John", 30 },
        { "Obama", 33 },
        { "Edward", 25}
    };

    struct Person p2[3] = { "John", 30, "Obama", 33, "Edward", 25 };
    for (int i = 0; i < 3;i ++){
        printf("Name:%s Age:%d\n",p1[i].name,p1[i].age);
    }
    printf("-----------------\n");
    for (int i = 0; i < 3; i++){
        printf("Name:%s Age:%d\n", p2[i].name, p2[i].age);
    }
    printf("-----------------\n");
    //在堆上分配结构体数组
    struct Person* p3 = (struct Person*)malloc(sizeof(struct Person) * 3);
    for (int i = 0; i < 3;i++){
        sprintf(p3[i].name, "Name_%d", i + 1);
        p3[i].age = 20 + i;
    }
    for (int i = 0; i < 3; i++){
        printf("Name:%s Age:%d\n", p3[i].name, p3[i].age);
    }
}
```

7.2 结构体嵌套指针

7.2.1 结构体嵌套一级指针

```
struct Person{
    char* name;
    int age;
};
```

```c
void allocate_memory(struct Person** person){
    if (person == NULL){
        return;
    }
    struct Person* temp = (struct Person*)malloc(sizeof(struct Person));
    if (temp == NULL){
        return;
    }
    //给 name 指针分配内存
    temp->name = (char*)malloc(sizeof(char)* 64);
    strcpy(temp->name, "John");
    temp->age = 100;

    *person = temp;
}

void print_person(struct Person* person){
    printf("Name:%s Age:%d\n",person->name,person->age);
}

void free_memory(struct Person** person){
    if (person == NULL){
        return;
    }
    struct Person* temp = *person;
    if (temp->name != NULL){
        free(temp->name);
        temp->name = NULL;
    }

    free(temp);
}

void test(){

    struct Person* p = NULL;
    allocate_memory(&p);
    print_person(p);
    free_memory(&p);
}
```

7.2.2 结构体嵌套二级指针

```c
//一个老师有 N 个学生
typedef struct _TEACHER{
    char name[64];
    char** students;
}Teacher;

void create_teacher(Teacher** teacher,int n,int m){

    if (teacher == NULL){
        return;
    }

    //创建老师数组
    Teacher* teachers = (Teacher*)malloc(sizeof(Teacher)* n);
    if (teachers == NULL){
        return;
    }

    //给每一个老师分配学生
    int num = 0;
    for (int i = 0; i < n; i ++){
        sprintf(teachers[i].name, "老师_%d", i + 1);
        teachers[i].students = (char**)malloc(sizeof(char*) * m);
        for (int j = 0; j < m;j++){
            teachers[i].students[j] = malloc(64);
            sprintf(teachers[i].students[j], "学生_%d", num + 1);
            num++;
        }
    }

    *teacher = teachers;
}

void print_teacher(Teacher* teacher,int n,int m){
    for (int i = 0; i < n; i ++){
        printf("%s:\n", teacher[i].name);
        for (int j = 0; j < m;j++){
            printf("  %s",teacher[i].students[j]);
        }
        printf("\n");
    }
}

void free_memory(Teacher** teacher,int n,int m){
```

```
    if (teacher == NULL){
        return;
    }

    Teacher* temp = *teacher;

    for (int i = 0; i < n; i ++){

        for (int j = 0; j < m;j ++){
            free(temp[i].students[j]);
            temp[i].students[j] = NULL;
        }

        free(temp[i].students);
        temp[i].students = NULL;
    }

    free(temp);

}

void test(){

    Teacher* p = NULL;
    create_teacher(&p,2,3);
    print_teacher(p, 2, 3);
    free_memory(&p,2,3);
}
```

7.3 结构体成员偏移量

```
//一旦结构体定义下来，则结构体中的成员内存布局就定下了
typedef struct Teacher
{
    char a;
    int b;
    int c;

} Teacher;

void test(){

    Teacher  t1;
```

```
    Teacher*p = NULL;
    p = &t1;

    int offsize1 = (int)&(p->b) - (int)p;  //age 相对于结构体 Teacher 的偏移量
    int offsize2 = (int)&(((Teacher *)0)->b);//绝对 0 地址 age 的偏移量
    int offsize3 = offsetof(Teacher, b);

    printf("offsize1:%d \n", offsize1);
    printf("offsize2:%d \n", offsize2);
    printf("offsize3:%d \n", offsize3);
}
```

7.4 结构体字节对齐

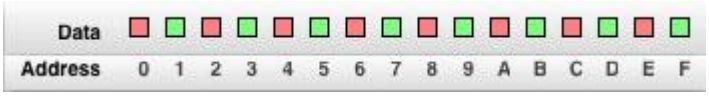
在用 sizeof 运算符求算某结构体所占空间时，并不是简单地将结构体中所有元素各自占的空间相加，这里涉及到内存字节对齐的问题。

从理论上讲，对于任何变量的访问都可以从任何地址开始访问，但是事实上不是如此，实际上访问特定类型的变量只能在特定的地址访问 ,这就需要各个变量在空间上按一定的规则排列， 而不是简单地顺序排列，这就是内存对齐。

7.4.1 内存对齐

7.4.1.1 内存对齐原因

我们知道内存的最小单元是一个字节，当 cpu 从内存中读取数据的时候，是一个一个字节读取，所以内存对我们应该是入下图这样：



但是实际上 cpu 将内存当成多个块 ,每次从内存中读取一个块 这个块的大小可能是 2、

4、8、16 等，

那么下面，我们来分析下非内存对齐和内存对齐的优缺点在哪？

内存对齐是操作系统为了提高访问内存的策略。操作系统在访问内存的时候，每次读取一定长度(这个长度是操作系统默认的对齐数，或者默认对齐数的整数倍)。如果没有对齐，为了访问一个变量可能产生二次访问。

至此大家应该能够简单明白，为什么要简单内存对齐？

- 提高存取数据的速度。比如有的平台每次都是从偶地址处读取数据，对于一个 int 型的变量，若从偶地址单元处存放，则只需一个读取周期即可读取该变量；但是若从奇地址单元处存放，则需要 2 个读取周期读取该变量。
- 某些平台只能在特定的地址处访问特定类型的数据，否则抛出硬件异常给操作系统。

7.4.1.1 如何内存对齐

- 对于标准数据类型，它的地址只要是它的长度的整数倍。
- 对于非标准数据类型，比如结构体，要遵循一下对齐原则：

1. 数组成员对齐规则。第一个数组成员应该放在 offset 为 0 的地方，以后每个数组成员应该放在 offset 为 **min（当前成员的大小，#pargama pack(n)）**整数倍的地方开始（比如 int 在 32 位机器为 4 字节，#pargama pack(2)，那么从 2 的倍数地方开始存储）。

2. 结构体总的大小，也就是 sizeof 的结果，必须是 **min（结构体内部最大成员，#pargama pack(n)）**的整数倍，不足要补齐。

3. 结构体做为成员的对齐规则。如果一个结构体 B 里嵌套另一个结构体 A,还是以

最大成员类型的大小对齐 ,但是结构体 A 的起点为 A 内部最大成员的整数倍的地方。(struct B 里存有 struct A ,A 里有 char ,int ,double 等成员 ,那 A 应该从 8 的整数倍开始存储。) , 结构体 A 中的成员的对齐规则仍满足原则 1、原则 2。

手动设置对齐模数:

- **#pragma pack(show)**

  显示当前 packing alignment 的字节数，以 warning message 的形式被显示。

- **#pragma pack(push)**

  将当前指定的 packing alignment 数组进行压栈操作，这里的栈是 the internal compiler stack,同事设置当前的 packing alignment 为 n；如果 n 没有指定，则将当前的 packing alignment 数组压栈。

- **#pragma pack(pop)**

  从 internal compiler stack 中删除最顶端的 reaord; 如果没有指定 n,则当前栈顶 record 即为新的 packing alignement 数值；如果指定了 n，则 n 成为新的 packing alignment 值

- **#pragma pack(n)**

  指定 packing 的数值 ,以字节为单位 缺省数值是 8 ,合法的数值分别是 1,2,4,8,16。

7.4.2 内存对齐案例

```
#pragma pack(4)

typedef struct _STUDENT{
    int a;
    char b;
    double c;
```

```
    float d;
}Student;

typedef struct _STUDENT2{
    char a;
    Student b;
    double c;
}Student2;

void test01(){

    //Student
    //a 从偏移量 0 位置开始存储
    //b 从 4 位置开始存储
    //c 从 8 位置开始存储
    //d 从 12 位置开存储
    //所以 Student 内部对齐之后的大小为 20 ， 整体对齐，整体为最大类型的整数倍 也就是 8 的整数倍 为 24

    printf("sizeof Student:%d\n",sizeof(Student));

    //Student2
    //a 从偏移量为 0 位置开始 8
    //b 从偏移量为 Student 内部最大成员整数倍开始， 也就是 8 开始 24
    //c 从 8 的整数倍地方开始,也就是 32 开始
    //所以结构体 Sutdnet2 内部对齐之后的大小为： 40 ， 由于结构体中最大成员为 8，必须为 8 的整数倍 所以大小为 40
    printf("sizeof Student2:%d\n", sizeof(Student2));
}
```

8. 文件操作

文件在今天的计算机系统中作用是很重要的。文件用来存放程序、文档、数据、表格、图片和其他很多种类的信息。作为一名程序员，您必须编程来创建、写入和读取文件。编写程序从文件读取信息或者将结果写入文件是一种经常性的需求。C 提供了强大的和文件进行通信的方法。使用这种方法我们可以在程序中打开文件，然后使用专门的 I/O 函数读取文件或者写入文件。

8.1 文件相关概念

8.1.1 文件的概念

一个文件通常就是磁盘上一段命名的存储区。但是对于操作系统来说，文件就会更复杂一些。例如，一个大文件可以存储在一些分散的区段中，或者还会包含一些操作系统可以确定其文件类型的附加数据，但是这些是操作系统，而不是我们程序员所要关心的事情。我们应该考虑如何在 C 程序中处理文件。

8.1.2 流的概念

流是一个动态的概念，可以将一个字节形象地比喻成一滴水，字节在设备、文件和程序之间的传输就是流，类似于水在管道中的传输，可以看出，流是对输入输出源的一种抽象，也是对传输信息的一种抽象。

C 语言中，I/O 操作可以简单地看作是从程序移进或移出字节，这种搬运的过程便称为流(stream)。程序只需要关心是否正确地输出了字节数据，以及是否正确地输入了要读取字节数据，特定 I/O 设备的细节对程序员是隐藏的。

8.1.2.1 文本流

文本流，也就是我们常说的以文本模式读取文件。文本流的有些特性在不同的系统中可能不同。其中之一就是文本行的最大长度。标准规定至少允许 254 个字符。另一个可能不同的特性是文本行的结束方式。例如在 Windows 系统中，文本文件约定以一个回车符和一个换行符结尾。但是在 Linux 下只使用一个换行符结尾。

标准 C 把文本定义为零个或者多个字符，后面跟一个表示结束的换行符(\n).对于那些文本行的外在表现形式与这个定义不同的系统上 ,库函数负责外部形式和内部形式之间的翻译。例如，在 Windows 系统中，在输出时，文本的换行符被写成一对回车/换行符。在输入时 ,文本中的回车符被丢弃。这种不必考虑文本的外部形势而操纵文本的能力简化了可移植程序的创建。

8.1.2.1 二进制流

二进制流中的字节将完全根据程序编写它们的形式写入到文件中 ,而且完全根据它们从文件或设备读取的形式读入到程序中。它们并未做任何改变。这种类型的流适用于非文本数据，但是如果你不希望 I/O 函数修改文本文件的行末字符，也可以把它们用于文本文件。

c 语言在处理这两种文件的时候并不区分，都看成是字符流，按字节进行处理。

我们程序中 ,经常看到的文本方式打开文件和二进制方式打开文件仅仅体现在换行符的处理上。

比如说，在 widows 下，文件的换行符是\r\n，而在 Linux 下换行符则是\n.

当对文件使用文本方式打开的时候 ,读写的 windows 文件中的换行符\r\n 会被替换成\n 读到内存中，当在 windows 下写入文件的时候，\n 被替换成\r\n 再写入文件。如果使用二进制方式打开文件 ,则不进行\r\n 和\n 之间的转换。 那么由于 Linux 下的换行符就是\n,所以文本文件方式和二进制方式无区别。

8.2 文件的操作

8.2.1 文件流总览

标准库函数是的我们在 C 程序中执行与文件相关的 I/O 任务非常方便。下面是关于文件 I/O 的一般概况。

1. 程序为同时处于活动状态的每个文件声明一个指针变量，其类型为 FILE*。这个指针指向这个 FILE 结构，当它处于活动状态时由流使用。
2. 流通过 fopen 函数打开。为了打开一个流，我们必须指定需要访问的文件或设备以及他们的访问方式(读、写、或者读写)。Fopen 和操作系统验证文件或者设备是否存在并初始化 FILE。
3. 根据需要对文件进行读写操作。
4. 最后调用 fclose 函数关闭流。关闭一个流可以防止与它相关的文件被再次访问 ,保证任何存储于缓冲区中的数据被正确写入到文件中 ,并且释放 FILE 结构。

标准 I/O 更为简单，因为它们并不需要打开或者关闭。

I/O 函数以三种基本的形式处理数据：**单个字符**、**文本行**和**二进制数据**。对于每种形式都有一组特定的函数对它们进行处理。

输入/输出函数家族

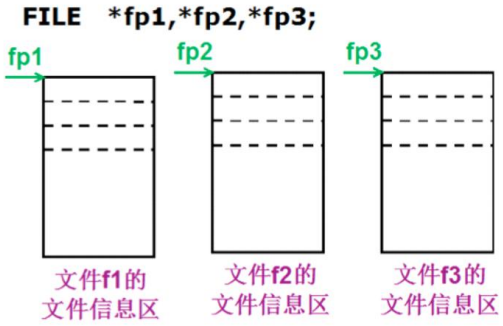
家族名	目的	可用于所有流	只用于 stdin 和 stdout
getchar	字符输入	fgetc、getc	getchar
putchar	字符输出	fputc、putc	putchar

gets	文本行输入	fgets	gets
puts	文本行输出	fputs	puts
scanf	格式化输入	fscanf	scanf
printf	格式化输出	fprintf	printf

8.2.2 文件指针

我们知道，文件是由操作系统管理的单元。当我们想操作一个文件的时候，让操作系统帮我们打开文件，操作系统把我们指定要打开文件的信息保存起来，并且返回给我们一个指针指向文件的信息。文件指针也可以理解为代指打开的文件。这个指针的类型为 FILE 类型。该类型定义在 stdio.h 头文件中。通过文件指针，我们就可以对文件进行各种操作。

对于每一个 ANSI C 程序，运行时系统必须提供至少三个流-标准输入(stdin)、标准输出(stdout)、标准错误(stderr)，它们都是一个指向 FILE 结构的指针。标准输入是缺省情况下的输入来源，标准输出时缺省情况下的输出设置。具体缺省值因编译器而异，通常标准输入为键盘设备、标准输出为终端或者屏幕。



ANSI C 并未规定 FILE 的成员，不同编译器可能有不同的定义。VS 下 FILE 信息如下：

```
struct _iobuf {
        char  *_ptr;          //文件输入的下一个位置
        int   _cnt;           //剩余多少字符未被读取
        char  *_base;         //指基础位置(应该是文件的其始位置)
        int   _flag;          //文件标志
```

```
        int   _file;        //文件的有效性验证
        int   _charbuf;     //检查缓冲区状况,如果无缓冲区则不读取
        int   _bufsiz;      //文件的大小
        char  *_tmpfname;   //临时文件名
};
typedef struct _iobuf FILE;
```

8.2.3 文件缓冲区

- **文件缓冲区**

ANSI C 标准采用“缓冲文件系统”处理数据文件 所谓缓冲文件系统是指系统自动地在内存区为程序中每一个正在使用的文件开辟一个文件缓冲区从内存向磁盘输出数据必须先送到内存中的缓冲区，装满缓冲区后才一起送到磁盘去 如果从磁盘向计算机读入数据，则一次从磁盘文件将一批数据输入到内存缓冲区(充满缓冲 区)，然后再从缓冲区逐个地将数据送到程序数据区(给程序变量) 。

那么文件缓冲区有什么作用呢？

如我们从磁盘里取信息，我们先把读出的数据放在缓冲区，计算机再直接从缓冲区中取数据，等缓冲区的数据取完后再去磁盘中读取，这样就可以减少磁盘的读写次数，再加上计算机对缓冲区的操作大大快于对磁盘的操作，故应用缓冲区可大大提高计算机的运行速度。

8.2.4 文件打开关闭

8.2.4.1 文件打开(fopen)

文件的打开操作表示将给用户指定的文件在内存分配一个 FILE 结构区，并将该结构的指针返回给用户程序，以后用户程序就可用此 FILE 指针来实现对指定文件的存取操作了。当使用打开函数时，必须给出文件名、文件操作方式(读、写或读写)。

```
FILE * fopen(const char * filename, const char * mode);
功能：打开文件
参数：
    filename：需要打开的文件名，根据需要加上路径
    mode：打开文件的权限设置
返回值：
    成功：文件指针
    失败：NULL
```

方式	含义
“r”	打开，只读，文件必须已经存在。
“w”	只写,如果文件不存在则创建,如果文件已存在则把文件长度截断(Truncate)为 0 字节。再重新写,也就是替换掉原来的文件内容文件指针指到头。
“a”	只能在文件末尾追加数据,如果文件不存在则创建
“rb”	打开一个二进制文件，只读
“wb”	打开一个二进制文件，只写
“ab"	打开一个二进制文件，追加
“r+”	允许读和写,文件必须已存在
“w+”	允许读和写,如果文件不存在则创建,如果文件已存在则把文件

方式	含义
	长度截断为 0 字节再重新写 。
“a+”	允许读和追加数据,如果文件不存在则创建
“rb+”	以读/写方式打开一个二进制文件
“wb+”	以读/写方式建立一个新的二进制文件
“ab+”	以读/写方式打开一个二进制文件进行追加

示例代码：

```c
void test(){

    FILE *fp = NULL;

    // "\\"这样的路径形式，只能在 windows 使用
    // "/"这样的路径形式，windows 和 linux 平台下都可用，建议使用这种
    // 路径可以是相对路径，也可是绝对路径
    fp = fopen("../test", "w");
    //fp = fopen("..\\test", "w");

    if (fp == NULL) //返回空，说明打开失败
    {
        //perror()是标准出错打印函数，能打印调用库函数出错原因
        perror("open");
        return -1;
    }
}
```

应该检查 fopen 的返回值!如何函数失败，它会返回一个 NULL 值。如果程序不检查错误，

这个 NULL 指针就会传给后续的 I/O 函数。它们将对这个指针执行间接访问，并将失败.

8.2.4.2 文件关闭(fclose)

文件操作完成后，如果程序没有结束，必须要用 fclose()函数进行关闭，这是因为对打开的文件进行写入时，若文件缓冲区的空间未被写入的内容填满，这些内容不会写到打开的文件中。只有对打开的文件进行关闭操作时，停留在文件缓冲区的内容才能写到该文件中去，从而使文件完整。再者一旦关闭了文件，该文件对应的 FILE 结构将被释放，从而使关闭的文件得到保护，因为这时对该文件的存取操作将不会进行。文件的关闭也意味着释放了该文件的缓冲区。

```
int fclose(FILE * stream);
功能：关闭先前 fopen()打开的文件。此动作让缓冲区的数据写入文件中，并释放系统所提供的文件资源。
参数：
    stream：文件指针
返回值：
    成功：0
    失败：-1
```

它表示该函数将关闭 FILE 指针对应的文件，并返回一个整数值。若成功地关闭了文件，则返回一个 0 值，否则返回一个非 0 值.

8.2.4 文件读写函数回顾

- 按照字符读写文件：fgetc(), fputc()
- 按照行读写文件：fputs(), fgets()
- 按照块读写文件：fread(), fwirte()
- 按照格式化读写文件：fprintf(), fscanf()
- 按照随机位置读写文件：fseek(), ftell(), rewind()

8.2.4.1 字符读写函数回顾

```
int fputc(int ch, FILE * stream);
功能：将 ch 转换为 unsigned char 后写入 stream 指定的文件中
参数：
    ch：需要写入文件的字符
    stream：文件指针
返回值：
    成功：成功写入文件的字符
    失败：返回-1

int fgetc(FILE * stream);
功能：从 stream 指定的文件中读取一个字符
参数：
    stream：文件指针
返回值：
    成功：返回读取到的字符
    失败：-1

int feof(FILE * stream);
功能：检测是否读取到了文件结尾
参数：
    stream：文件指针
返回值：
    非 0 值：已经到文件结尾
    0：没有到文件结尾
```

```
void test(){

    //写文件
    FILE* fp_write= NULL;
    //写方式打开文件
    fp_write = fopen("./mydata.txt", "w+");
    if (fp_write == NULL){
        return;
    }

    char buf[] = "this is a test for fputc!";
    for (int i = 0; i < strlen(buf);i++){
        fputc(buf[i], fp_write);
    }

    fclose(fp_write);
```

```
    //读文件
    FILE* fp_read = NULL;
    fp_read = fopen("./mydata.txt", "r");
    if (fp_read == NULL){
        return;
    }

#if 0
    //判断文件结尾 注意：多输出一个空格
    while (!feof(fp_read)){
        printf("%c",fgetc(fp_read));
    }
#else
    char ch;
    while ((ch = fgetc(fp_read)) != EOF){
        printf("%c", ch);
    }
#endif
}
```

将把流指针 fp 指向的文件中的一个字符读出，并赋给 ch，当执行 fgetc()函数时，若当时文件指针指到文件尾，即遇到文件结束标志 EOF(其对应值为-1)，该函数返回一个 -1 给 ch，在程序中常用检查该函数返回值是否为 -1 来判断是否已读到文件尾，从而决定是否继续。

8.2.4.2 行读写函数回顾

```
int fputs(const char * str, FILE * stream);
功能：将 str 所指定的字符串写入到 stream 指定的文件中， 字符串结束符 '\0' 不写入文件。
参数：
    str：字符串
    stream：文件指针
返回值：
    成功：0
    失败：-1

char * fgets(char * str, int size, FILE * stream);
功能：从 stream 指定的文件内读入字符，保存到 str 所指定的内存空间，直到出现换行字符、读到文件结尾或是已读了 size - 1 个字符为止，最后会自动加上字符 '\0' 作为字符串结束。
```

参数：
    str：字符串
    size：指定最大读取字符串的长度（size - 1）
    stream：文件指针
返回值：
    成功：成功读取的字符串
    读到文件尾或出错： NULL

```
void test(){

    //写文件
    FILE* fp_write= NULL;
    //写方式打开文件
    fp_write = fopen("./mydata.txt", "w+");
    if (fp_write == NULL){
        perror("fopen:");
        return;
    }

    char* buf[] = {
        "01 this is a test for pfutc!\n",
        "02 this is a test for pfutc!\n",
        "03 this is a test for pfutc!\n",
        "04 this is a test for pfutc!\n",
    };
    for (int i = 0; i < 4; i ++){
        fputs(buf[i], fp_write);
    }

    fclose(fp_write);

    //读文件
    FILE* fp_read = NULL;
    fp_read = fopen("./mydata.txt", "r");
    if (fp_read == NULL){
        perror("fopen:");
        return;
    }

    //判断文件结尾
    while (!feof(fp_read)){
        char temp[1024] = { 0 };
        fgets(temp, 1024, fp_read);
```

```
        printf("%s",temp);
    }

    fclose(fp_read);
}
```

8.2.4.3 块读写函数回顾

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
功能：以数据块的方式给文件写入内容
参数：
    ptr：准备写入文件数据的地址
    size： size_t 为 unsigned int 类型，此参数指定写入文件内容的块数据大小
    nmemb：写入文件的块数，写入文件数据总大小为：size * nmemb
    stream：已经打开的文件指针
返回值：
    成功：实际成功写入文件数据的块数，此值和 nmemb 相等
    失败：0

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
功能：以数据块的方式从文件中读取内容
参数：
    ptr：存放读取出来数据的内存空间
    size： size_t 为 unsigned int 类型，此参数指定读取文件内容的块数据大小
    nmemb：读取文件的块数，读取文件数据总大小为：size * nmemb
    stream：已经打开的文件指针
返回值：
    成功：实际成功读取到内容的块数，如果此值比 nmemb 小，但大于 0，说明读到文件的结尾。
    失败：0
```

```
typedef struct _TEACHER{
    char name[64];
    int age;
}Teacher;

void test(){

    //写文件
    FILE* fp_write= NULL;
    //写方式打开文件
    fp_write = fopen("./mydata.txt", "wb");
    if (fp_write == NULL){
```

```
        perror("fopen:");
        return;
    }

    Teacher teachers[4] = {
        { "Obama", 33 },
        { "John", 28 },
        { "Edward", 45},
        { "Smith", 35}
    };

    for (int i = 0; i < 4; i ++){
        fwrite(&teachers[i],sizeof(Teacher),1, fp_write);
    }
    //关闭文件
    fclose(fp_write);

    //读文件
    FILE* fp_read = NULL;
    fp_read = fopen("./mydata.txt", "rb");
    if (fp_read == NULL){
        perror("fopen:");
        return;
    }

    Teacher temps[4];
    fread(&temps, sizeof(Teacher), 4, fp_read);
    for (int i = 0; i < 4;i++){
        printf("Name:%s Age:%d\n",temps[i].name,temps[i].age);
    }

    fclose(fp_read);
}
```

8.2.4.4 格式化读写函数回顾

```
int fprintf(FILE * stream, const char * format, ...);
功能：根据参数 format 字符串来转换并格式化数据，然后将结果输出到 stream 指定的文件中，
指定出现字符串结束符 '\0' 为止。
参数：
    stream：已经打开的文件
    format：字符串格式，用法和 printf()一样
```

返回值：
    成功：实际写入文件的字符个数
    失败：-1

```
int fscanf(FILE * stream, const char * format, ...);
```

功能：从 stream 指定的文件读取字符串，并根据参数 format 字符串来转换并格式化数据。
参数：
    stream：已经打开的文件
    format：字符串格式，用法和 scanf()一样
返回值：
    成功：实际从文件中读取的字符个数
    失败： - 1

注意：fscanf 遇到空格和换行时结束。

```
void test(){

    //写文件
    FILE* fp_write= NULL;
    //写方式打开文件
    fp_write = fopen("./mydata.txt", "w");
    if (fp_write == NULL){
        perror("fopen:");
        return;
    }

    fprintf(fp_write,"hello world:%d!",10);

    //关闭文件
    fclose(fp_write);

    //读文件
    FILE* fp_read = NULL;
    fp_read = fopen("./mydata.txt", "rb");
    if (fp_read == NULL){
        perror("fopen:");
        return;
    }

    char temps[1024] = { 0 };
    while (!feof(fp_read)){
        fscanf(fp_read, "%s", temps);
        printf("%s", temps);
```

```
    }

    fclose(fp_read);
}
```

8.2.5.5 随机读写函数回顾

```
int fseek(FILE *stream, long offset, int whence);
功能：移动文件流（文件光标）的读写位置。
参数：
    stream：已经打开的文件指针
    offset：根据 whence 来移动的位移数（偏移量），可以是正数，也可以负数，如果正数，则相对于 whence 往右移动，如果是负数，则相对于 whence 往左移动。如果向前移动的字节数超过了文件开头则出错返回，如果向后移动的字节数超过了 文件末尾，再次写入时将增大文件尺寸。
    whence：其取值如下：
        SEEK_SET：从文件开头移动 offset 个字节
        SEEK_CUR：从当前位置移动 offset 个字节
        SEEK_END：从文件末尾移动 offset 个字节
返回值：
    成功：0
    失败：-1

long ftell(FILE *stream);
功能：获取文件流（文件光标）的读写位置。
参数：
    stream：已经打开的文件指针
返回值：
    成功：当前文件流（文件光标）的读写位置
    失败：-1

void rewind(FILE *stream);
功能：把文件流（文件光标）的读写位置移动到文件开头。
参数：
    stream：已经打开的文件指针
返回值：
    无返回值
```

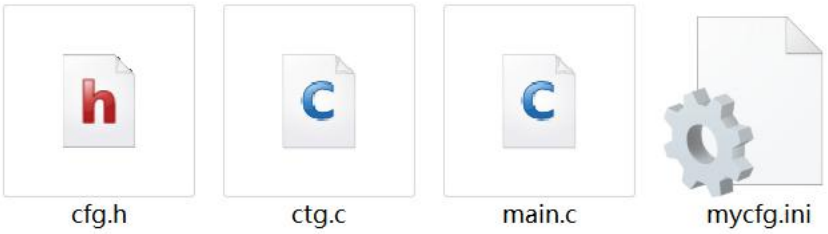
```
typedef struct _TEACHER{
    char name[64];
    int age;
}Teacher;
```

```c
void test(){
    //写文件
    FILE* fp_write = NULL;
    //写方式打开文件
    fp_write = fopen("./mydata.txt", "wb");
    if (fp_write == NULL){
        perror("fopen:");
        return;
    }

    Teacher teachers[4] = {
        { "Obama", 33 },
        { "John", 28 },
        { "Edward", 45 },
        { "Smith", 35 }
    };

    for (int i = 0; i < 4; i++){
        fwrite(&teachers[i], sizeof(Teacher), 1, fp_write);
    }
    //关闭文件
    fclose(fp_write);

    //读文件
    FILE* fp_read = NULL;
    fp_read = fopen("./mydata.txt", "rb");
    if (fp_read == NULL){
        perror("fopen:");
        return;
    }

    Teacher temp;
    //读取第三个数组
    fseek(fp_read , sizeof(Teacher) * 2 , SEEK_SET);
    fread(&temp, sizeof(Teacher), 1, fp_read);
    printf("Name:%s Age:%d\n",temp.name,temp.age);

    memset(&temp,0,sizeof(Teacher));

    fseek(fp_read, -(int)sizeof(Teacher), SEEK_END);
    fread(&temp, sizeof(Teacher), 1, fp_read);
    printf("Name:%s Age:%d\n", temp.name, temp.age);

    rewind(fp_read);
```

```
    fread(&temp, sizeof(Teacher), 1, fp_read);
    printf("Name:%s Age:%d\n", temp.name, temp.age);

    fclose(fp_read);
}
```

8.4 文件读写案例

8.4.1 读写配置文件




```
struct info{
    char key[64];
    char val[128];
};

struct config{
    FILE *fp; //保存文件指针
    struct info *list; //保存配置信息
    int lines; //配置信息条数
};

//加载配置文件
int load_file(char *path, struct config **myconfig){
    if (NULL == path){
        return -1;
    }
    //以读写的方式打开文件
    FILE *fp = fopen(path, "r+");
    if (NULL ==fp){
        printf("文件打开失败!\n");
        return -2;
    }

    //配置文件信息分配内存
    struct config *conf = (struct config *)malloc(sizeof(struct config));
```

```
    conf->fp = fp;
    conf->list = NULL;

    //指针的间接赋值
    *myconfig = conf;

    return 0;
}

//统计文件行数
int count_file(struct config *config){
    if (NULL == config){
        return -1;
    }
    char buf[1024] = { 0 };
    int lines = 0;
    while (fgets(buf, 1024, config->fp)){
        //如果是注释则不统计
        if (buf[0] == '#'){ continue; }
        lines++;
    }
    //将文件指针重置到开始位置
    fseek(config->fp,0, SEEK_SET);

    return lines;
}

//解析配置文件
int parse_file(struct config *config){
    if (NULL == config){
        return -1;
    }

    //获得配置文件行数
    config->lines = count_file(config);
    //给每一行配置信息分配内存
    config->list = (struct info *)malloc(sizeof(struct info) *
config->lines);
    int index = 0;
    char buf[1024] = { 0 };
    while (fgets(buf, 1024, config->fp)){
        //去除每一行最后的\n字符
        buf[strlen(buf) - 1] = '\0';
        //如果是注释则不显示
```

```
        if (buf[0] == '#'){
            continue;
        }

        memset(config->list[index].key, 0, 64);
        memset(config->list[index].val, 0, 128);

        char *delimit = strchr(buf, ':');
        strncpy(config->list[index].key, buf, delimit - buf);
        strncpy(config->list[index].val, delimit + 1, strlen(delimit +
1));

        memset(buf, 0 , 1024);

        index++;
    }

    return 0;
}

const char *get_file(struct config *config, char *key){
    if (NULL == config){
        return NULL;
    }
    if (NULL == key){
        return NULL;
    }

    for (int i = 0; i < config->lines;i ++){
        if (strcmp(config->list[i].key,key) == 0){
            return config->list[i].val;
        }
    }

    return NULL;
}

void destroy_file(struct config *config){
    if (NULL == config){
        return;
    }
    //关闭文件指针
    fclose(config->fp);
    config->fp = NULL;
```

```
    //释放配置信息
    free(config->list);
    config->list = NULL;
    free(config);
}

void test(){
    char *path = "./my.ini";
    struct config *conf = NULL;
    load_file(path, &conf);
    parse_file(conf);
    printf("%s\n", get_file(conf, "username"));
    printf("%s\n", get_file(conf, "password"));
    printf("%s\n", get_file(conf, "server_ip"));
    printf("%s\n", get_file(conf, "server_port"));
    printf("%s\n", get_file(conf, "aaaa"));
    destroy_file(conf);
}
```

10. 函数指针和递归函数

10.1 函数指针

10.1.1 函数类型

通过什么来区分两个不同的函数？

一个函数在编译时被分配一个入口地址，这个地址就称为函数的指针，函数名代表函数的入口地址。

函数三要素： 名称、参数、返回值。C 语言中的函数有自己特定的类型。

c 语言中通过 typedef 为函数类型重命名：

```
typedef int f(int, int);      // f 为函数类型
typedef void p(int);      // p 为函数类型
```

这一点和数组一样，因此我们可以用一个指针变量来存放这个入口地址，然后通过该指针变量调用函数。

注意：通过函数类型定义的变量是不能够直接执行，因为没有函数体。只能通过类型定义一个函数指针指向某一个具体函数，才能调用。

```
typedef int(p)(int, int);

void my_func(int a,int b){
    printf("%d %d\n",a,b);
}

void test(){

    p p1;
    //p1(10,20); //错误，不能直接调用，只描述了函数类型，但是并没有定义函数体，没有函数体无法调用
    p* p2 = my_func;
    p2(10,20); //正确，指向有函数体的函数入口地址
}
```

10.1.2 函数指针(指向函数的指针)

- 函数指针定义方式(先定义函数类型，根据类型定义指针变量);
- 先定义函数指针类型，根据类型定义指针变量;
- 直接定义函数指针变量;

```
int my_func(int a,int b){
    printf("ret:%d\n", a + b);
    return 0;
}

//1. 先定义函数类型，通过类型定义指针
void test01(){
    typedef int(FUNC_TYPE)(int, int);
    FUNC_TYPE* f = my_func;
    //如何调用?
    (*f)(10, 20);
    f(10, 20);
```

```
}

//2. 定义函数指针类型
void test02(){
    typedef int(*FUNC_POINTER)(int, int);
    FUNC_POINTER f = my_func;
    //如何调用?
    (*f)(10, 20);
    f(10, 20);
}

//3. 直接定义函数指针变量
void test03(){

    int(*f)(int, int) = my_func;
    //如何调用?
    (*f)(10, 20);
    f(10, 20);
}
```

10.1.3 函数指针数组

函数指针数组，每个元素都是函数指针。

```
void func01(int a){
    printf("func01:%d\n",a);
}
void func02(int a){
    printf("func02:%d\n", a);
}
void func03(int a){
    printf("func03:%d\n", a);
}

void test(){

#if 0
    //定义函数指针
    void(*func_array[])(int) = { func01, func02, func03 };
#else
    void(*func_array[3])(int);
    func_array[0] = func01;
    func_array[1] = func02;
```

```
    func_array[2] = func03;
#endif

    for (int i = 0; i < 3; i ++){
        func_array[i](10 + i);
        (*func_array[i])(10 + i);
    }
}
```

10.1.4 函数指针做函数参数(回调函数)

函数参数除了是普通变量，还可以是函数指针变量。

```
//形参为普通变量
void fun( int x ){}
//形参为函数指针变量
void fun( int(*p)(int a) ){}
```

函数指针变量常见的用途之一是把指针作为参数传递到其他函数 ,指向函数的指针也可以作为参数，以实现函数地址的传递。

```
//加法计算器
int plus(int a,int b){
    return a + b;
}

//减法计算器
int minus(int a,int b){
    return a - b;
}

//计算器
#if 0
int caculator(int a,int b,int(*func)(int,int)){
    return func(a, b);
}
#else
typedef int(*FUNC_POINTER)(int, int);
int caculator(int a, int b, FUNC_POINTER func){
    return func(a, b);
}
#endif
```

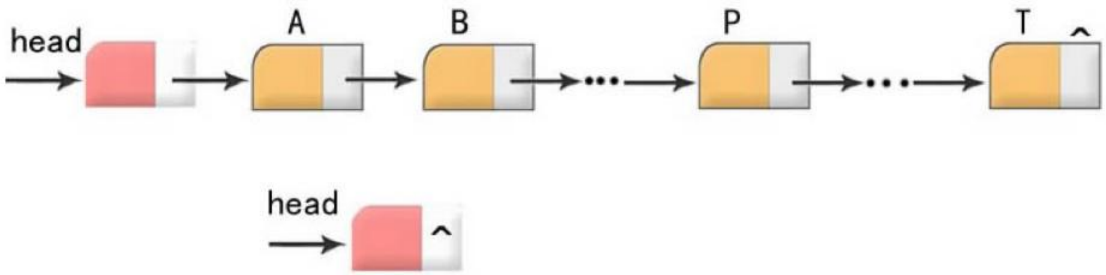
注意：函数指针和指针函数的区别：

- 函数指针是指向函数的指针；
- 指针函数是返回类型为指针的函数；

9. 链表

9.1 链表基本概念

9.1.1 什么是链表



- 链表是一种常用的数据结构，它通过指针将一些列数据结点，连接成一个数据链。相对于数组，链表具有更好的动态性（非顺序存储）。
- 数据域用来存储数据，指针域用于建立与下一个结点的联系。
- 建立链表时无需预先知道数据总量的，可以随机的分配空间，可以高效的在链表中的任意位置实时插入或删除数据。
- 链表的开销，主要是访问顺序性和组织链的空间损失。

数组和链表的区别：

数组：一次性分配一块连续的存储区域。

优点：随机访问元素效率高

缺点：1) 需要分配一块连续的存储区域（很大区域，有可能分配失败）

2) 删除和插入某个元素效率低

链表：无需一次性分配一块连续的存储区域，只需分配 n 块节点存储区域，通过指针建立关系。

优点：1) 不需要一块连续的存储区域

2) 删除和插入某个元素效率高

缺点：随机访问元素效率低

9.1.2 有关结构体的自身引用

问题 1：请问结构体可以嵌套本类型的结构体变量吗？

问题 2：请问结构体可以嵌套本类型的结构体指针变量吗？

```
typedef struct _STUDENT{
    char name[64];
    int age;
struct _STUDENT *s;
}Student;

typedef struct _TEACHER{
    char name[64];
    Student stu; //结构体可以嵌套其他类型的结构体
    //Teacher stu;
    //struct _TEACHER teacher; //此时 Teacher 类型的成员还没有确定，编译器无法分配内存
```

```
    struct _TEACHER* teacher; //不论什么类型的指针，都只占 4 个字节，编译器可确定内存分配
}Teacher;
```

- 结构体可以嵌套另外一个结构体的任何类型变量;
- 结构体嵌套本结构体普通变量（不可以）。本结构体的类型大小无法确定，类型本质：固定大小内存块别名;
- 结构体嵌套本结构体指针变量（可以），指针变量的空间能确定，32 位， 4 字节， 64 位， 8 字节;

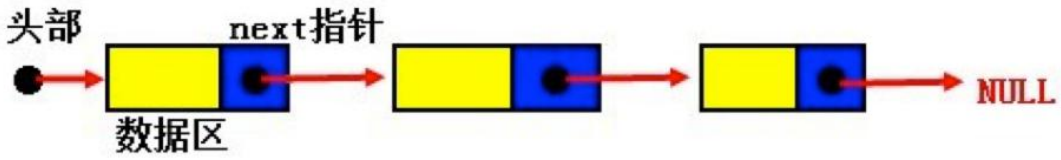
9.1.3 链表节点

大家思考一下，我们说链表是由一系列的节点组成，那么如何表示一个包含了数据域和指针域的节点呢？

链表的节点类型实际上是结构体变量，此结构体包含数据域和指针域：

- 数据域用来存储数据；
- 指针域用于建立与下一个结点的联系，当此节点为尾节点时，指针域的值为 NULL；

```
typedef struct Node
{
    //数据域
    int id;
    char name[50];

    //指针域
    struct Node *next;
}Node;
```



9.1.4 链表的分类

链表分为：静态链表和动态链表

静态链表和动态链表是线性表链式存储结构的两种不同的表示方式：

- 所有结点都是在程序中定义的，不是临时开辟的，也不能用完后释放，这种链表称为“静态链表”。
- 所谓动态链表，是指在程序执行过程中从无到有地建立起一个链表，即一个一个地开辟结点和输入各结点数据，并建立起前后相链的关系。

9.1.4.1 静态链表

```
typedef struct Stu
{
    int id; //数据域
    char name[100];

    struct Stu *next; //指针域
}Stu;

void test()
{
    //初始化三个结构体变量
    Stu s1 = { 1, "yuri", NULL };
    Stu s2 = { 2, "lily", NULL };
    Stu s3 = { 3, "lilei", NULL };

    s1.next = &s2; //s1 的 next 指针指向 s2
    s2.next = &s3;
    s3.next = NULL; //尾结点
```

```
    Stu *p = &s1;
    while (p != NULL)
    {
        printf("id = %d, name = %s\n", p->id, p->name);

        //结点往后移动一位
        p = p->next;
    }
}
```

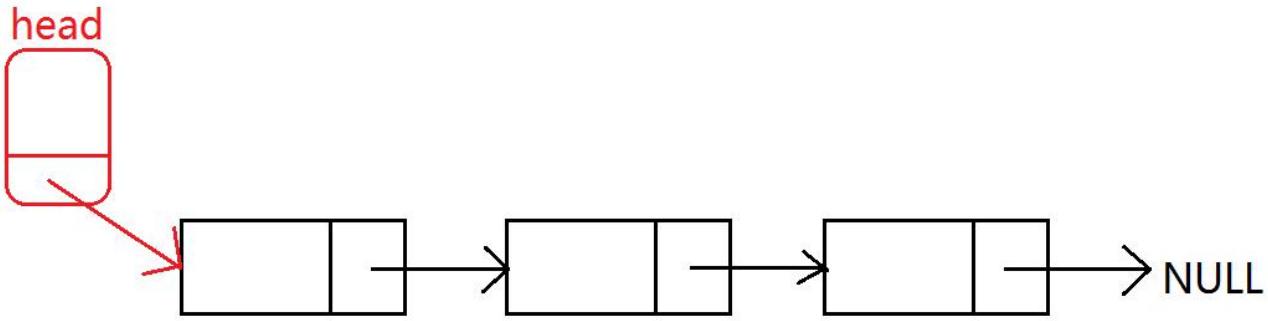
9.1.4.2 动态链表

```
typedef struct Stu{
    int id; //数据域
    char name[100];

    struct Stu *next; //指针域
}Stu;

void test(){
    //动态分配 3 个节点
    Stu *s1 = (Stu *)malloc(sizeof(Stu));
    s1->id = 1;
    strcpy(s1->name, "yuri");

    Stu *s2 = (Stu *)malloc(sizeof(Stu));
    s2->id = 2;
    strcpy(s2->name, "lily");

    Stu *s3 = (Stu *)malloc(sizeof(Stu));
    s3->id = 3;
    strcpy(s3->name, "lilei");

    //建立节点的关系
    s1->next = s2; //s1 的 next 指针指向 s2
    s2->next = s3;
    s3->next = NULL; //尾结点

    //遍历节点
    Stu *p = s1;
    while (p != NULL)
```

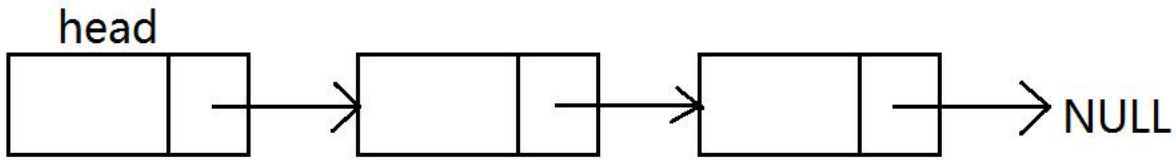
```
    {
        printf("id = %d, name = %s\n", p->id, p->name);

        //结点往后移动一位
        p = p->next;
    }

    //释放节点空间
    p = s1;
    Stu *tmp = NULL;
    while (p != NULL)
    {
        tmp = p;
        p = p->next;

        free(tmp);
        tmp = NULL;
    }
}
```

9.1.4.3 带头和不带头链表

- 带头链表：固定一个节点作为头结点(数据域不保存有效数据)，起一个标志位的作用，以后不管链表节点如果改变，此头结点固定不变。

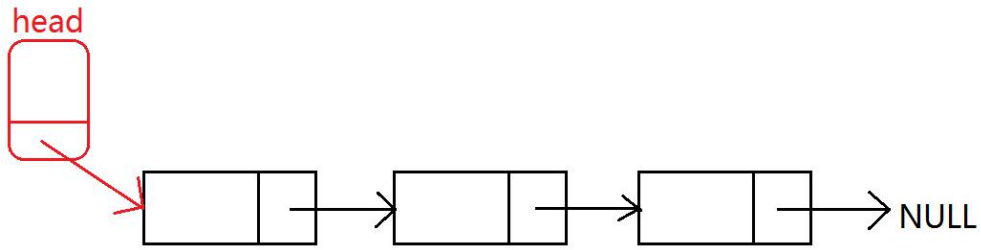


- 不带头链表：头结点不固定，根据实际需要变换头结点(如在原来头结点前插入新节点，然后，新节点重新作为链表的头结点)。

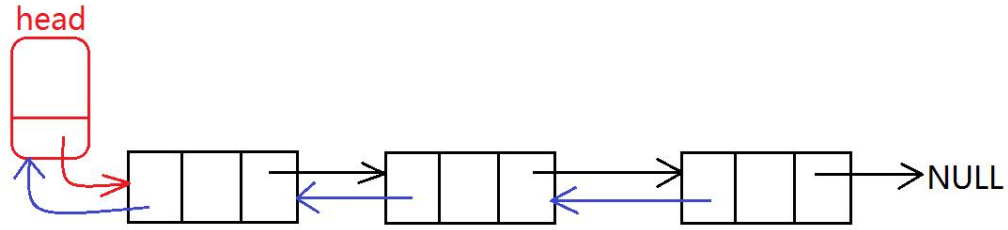


9.1.4.4 单向链表、双向链表、循环链表

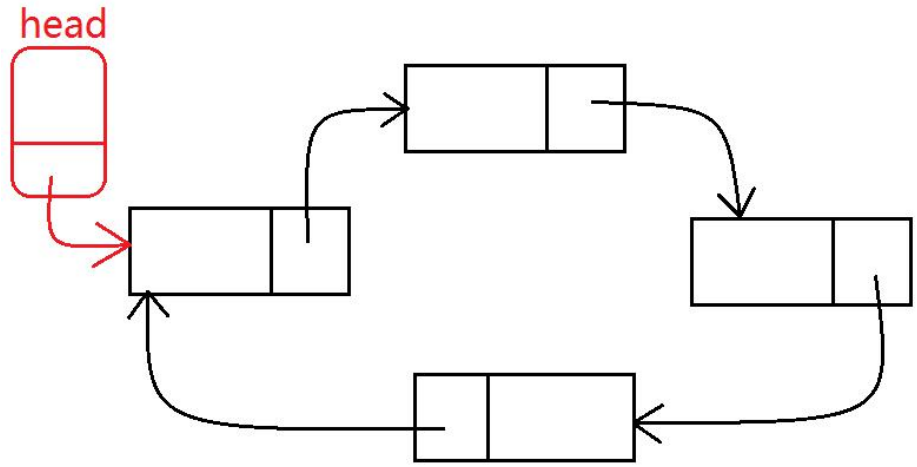
单向链表：



双向链表：



循环链表：



9.2 链表基本操作

9.2.1 创建链表

使用结构体定义节点类型：

```
typedef struct _LINKNODE
{
    int id; //数据域
    struct _LINKNODE* next; //指针域
}link_node;
```

编写函数：link_node* init_linklist()

建立带有头结点的单向链表，循环创建结点，结点数据域中的数值从键盘输入，以 -1 作为输入结束标志，链表的头结点地址由函数值返回.

```
typedef struct _LINKNODE{
    int data;
    struct _LINKNODE* next;
}link_node;

link_node* init_linklist(){

    //创建头结点指针
    link_node* head = NULL;
    //给头结点分配内存
    head = (link_node*)malloc(sizeof(link_node));
    if (head == NULL){
        return NULL;
    }
    head->data = -1;
    head->next = NULL;

    //保存当前节点
    link_node* p_current = head;
    int data = -1;
    //循环向链表中插入节点
    while (1){
```

```
        printf("please input data:\n");
        scanf("%d",&data);

        //如果输入-1，则退出循环
        if (data == -1){
            break;
        }

        //给新节点分配内存
        link_node* newnode = (link_node*)malloc(sizeof(link_node));
        if (newnode == NULL){
            break;
        }

        //给节点赋值
        newnode->data = data;
        newnode->next = NULL;

        //新节点入链表，也就是将节点插入到最后一个节点的下一个位置
        p_current->next = newnode;
        //更新辅助指针 p_current
        p_current = newnode;
    }

    return head;
}
```

9.2.2 遍历链表

编写函数：void foreach_linklist(link_node* head)

顺序输出单向链表各项结点数据域中的内容：

```
//遍历链表
void foreach_linklist(link_node* head){
    if (head == NULL){
        return;
    }

    //赋值指针变量
```

```
    link_node* p_current = head->next;
    while (p_current != NULL){
        printf("%d ",p_current->data);
        p_current = p_current->next;
    }
    printf("\n");
}
```

9.2.3 插入节点

编写函数: void insert_linklist(link_node* head,int val,int data).

在指定值后面插入数据 data,如果值 val 不存在，则在尾部插入。

```
//在值 val 前插入节点
void insert_linklist(link_node* head, int val, int data){

    if (head == NULL){
        return;
    }

    //两个辅助指针
    link_node* p_prev = head;
    link_node* p_current = p_prev->next;
    while (p_current != NULL){
        if (p_current->data == val){
            break;
        }
        p_prev = p_current;
        p_current = p_prev->next;
    }

    //如果 p_current 为 NULL，说明不存在值为 val 的节点
    if (p_current == NULL){
        printf("不存在值为%d 的节点!\n",val);
        return;
    }

    //创建新的节点
    link_node* newnode = (link_node*)malloc(sizeof(link_node));
    newnode->data = data;
    newnode->next = NULL;
```

```
    //新节点入链表
    newnode->next = p_current;
    p_prev->next = newnode;
}
```

9.2.4 删除节点

编写函数: void remove_linklist(link_node* head,int val)

删除第一个值为 val 的结点.

```
//删除值为 val 的节点
void remove_linklist(link_node* head,int val){
    if (head == NULL){
        return;
    }

    //辅助指针
    link_node* p_prev = head;
    link_node* p_current = p_prev->next;

    //查找值为 val 的节点
    while (p_current != NULL){
        if (p_current->data == val){
            break;
        }
        p_prev = p_current;
        p_current = p_prev->next;
    }
    //如果 p_current 为 NULL，表示没有找到
    if (p_current == NULL){
        return;
    }

    //删除当前节点： 重新建立待删除节点(p_current)的前驱后继节点关系
    p_prev->next = p_current->next;
    //释放待删除节点的内存
    free(p_current);
}
```

9.2.5 销毁链表

编写函数: void destroy_linklist(link_node* head)

销毁链表，释放所有节点的空间.

```
//销毁链表
void destroy_linklist(link_node* head){
    if (head == NULL){
        return;
    }
    //赋值指针
    link_node* p_current = head;
    while (p_current != NULL){
        //缓存当前节点下一个节点
        link_node* p_next = p_current->next;
        free(p_current);
        p_current = p_next;
    }
}
```

11. 预处理

11.1 预处理的基本概念

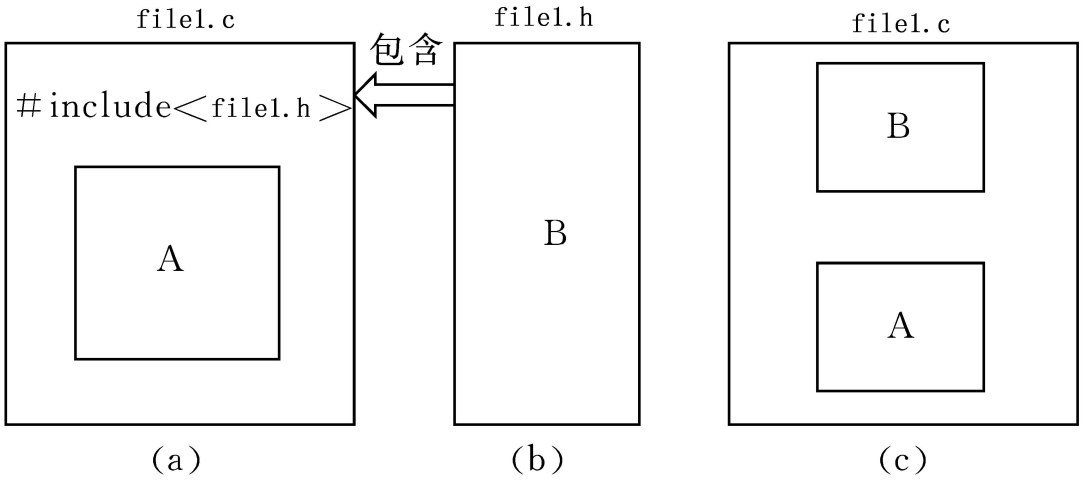
C 语言对源程序处理的四个步骤：预处理、编译、汇编、链接。

预处理是在程序源代码被编译之前，由预处理器（Preprocessor）对程序源代码进行的处理。这个过程并不对程序的源代码语法进行解析，但它会把源代码分割或处理成为特定的符号为下一步的编译做准备工作。

11.1 文件包含指令(#include)

11.1.1 文件包含处理

“文件包含处理”是指一个源文件可以将另外一个文件的全部内容包含进来。C 语言提供了#include 命令用来实现“文件包含”的操作。



11.1.2 #incude<>和#include""区别

- "" 表示系统先在 file1.c 所在的当前目录找 file1.h，如果找不到，再按系统指定的目录检索。
- < > 表示系统直接按系统指定的目录检索。

注意：

1. #include <>常用于包含库函数的头文件；
2. #include ""常用于包含自定义的头文件；
3. 理论上#include 可以包含任意格式的文件(.c .h 等) ，但一般用于头文件的包含；

11.2 宏定义

11.2.1 无参数的宏定义(宏常量)

如果在程序中大量使用到了 100 这个值，那么为了方便管理，我们可以将其定义为：

const int num = 100; 但是如果我们使用 num 定义一个数组，在不支持 c99 标准的编译器上是不支持的，因为 num 不是一个编译器常量，如果想得到了一个编译器常量，那么可以使用：

#define num 100

在编译预处理时，将程序中在该语句以后出现的所有的 num 都用 100 代替。这种方法使用户能以一个简单的名字代替一个长的字符串，在预编译时将宏名替换成字符串的过程称为“宏展开”。宏定义，只在宏定义的文件中起作用。

```
#define PI 3.1415
void test(){
    double r = 10.0;
    double s = PI * r * r;
    printf("s = %lf\n", s);
}
```

说明：

1) 宏名一般用大写，以便于与变量区别；

2) 宏定义可以是常数、表达式等；

3) 宏定义不作语法检查，只有在编译被宏展开后的源程序才会报错；

4) 宏定义不是 C 语言，不在行末加分号；

5) 宏名有效范围为从定义到本源文件结束；

6) 可以用#undef 命令终止宏定义的作用域；

7) 在宏定义中，可以引用已定义的宏名；

11.2.2 带参数的宏定义(宏函数)

在项目中，经常把一些短小而又频繁使用的函数写成宏函数，这是由于宏函数没有普通函数参数压栈、跳转、返回等的开销，可以调高程序的效率。

宏通过使用参数，可以创建外形和作用都与函数类似地类函数宏(function-like macro).宏的参数也用圆括号括起来。

```
#define SUM(x,y) ((x)+(y))
void test(){

    //仅仅只是做文本替换 下例替换为 int ret = ((10)+(20));
    //不进行计算
    int ret = SUM(10, 20);
    printf("ret:%d\n",ret);
}
```

注意:

1) 宏的名字中不能有空格，但是在替换的字符串中可以有空格。ANSI C 允许在参数列表中使用空格；

2) 用括号括住每一个参数，并括住宏的整体定义。

3) 用大写字母表示宏的函数名。

4) 如果打算宏代替函数来加快程序运行速度。假如在程序中只使用一次宏对程序的运行时间没有太大提高。

11.3 条件编译

11.3.1 基本概念

一般情况下，源程序中所有的行都参加编译。但有时希望对部分源程序行只在满足一定条件时才编译，即对这部分源程序行指定编译条件。

测试存在：

```
#ifdef 标识符
    程序段 1
#else
    程序段 2
#endif
```

测试不存在：

```
#ifndef 标识符
    程序段 1
#else
    程序段 2
#endif
```

根据表达式定义：

```
#if 表达式
    程序段 1
#else
    程序段 2
#endif
```

11.3.2 条件编译

- 防止头文件被重复包含引用；

```
#ifndef _SOMEFILE_H
#define _SOMEFILE_H

//需要声明的变量、函数
//宏定义
//结构体

#endif
```

11.4 一些特殊的预定宏

C 编译器，提供了几个特殊形式的预定义宏，在实际编程中可以直接使用，很方便。

```
//  __FILE__          宏所在文件的源文件名
//  __LINE__          宏所在行的行号
//  __DATE__          代码编译的日期
//  __TIME__          代码编译的时间

void test()
{
    printf("%s\n", __FILE__);
    printf("%d\n", __LINE__);
    printf("%s\n", __DATE__);
    printf("%s\n", __TIME__);
}
```

12. 动态库的封装和使用

12.1 库的基本概念

库是已经写好的、成熟的、可复用的代码。每个程序都需要依赖很多底层库，不可能每个人的代码从零开始编写代码，因此库的存在具有非常重要的意义。

在我们的开发的应用中经常有一些公共代码是需要反复使用的 ,就把这些代码编译为库文件。

库可以简单看成一组目标文件的集合 ,将这些目标文件经过压缩打包之后形成的一个文件。像在 Windows 这样的平台上 ,最常用的 c 语言库是由集成按开发环境所附带的运行库，这些库一般由编译厂商提供。

12.2 windows 下静态库创建和使用

12.2.1 静态库的创建

1. 创建一个新项目，在已安装的模板中选择“常规”，在右边的类型下选择“空项目”，在名称和解决方案名称中输入 staticlib。点击确定。

2.在解决方案资源管理器的头文件中添加,mylib.h 文件，在源文件添加 mylib.c 文件（即实现文件）。

3.在 mylib.h 文件中添加如下代码：

```
#ifndef TEST_H
#define TEST_H

int myadd(int a,int b);

#endif
```

4.在 mylib.c 文件中添加如下代码：

```
#include"test.h"
int myadd(int a, int b){
    return a + b;

}
```

5. 配置项目属性。因为这是一个静态链接库 ,所以应在项目属性的“配置属性”下选择“常规”，在其下的配置类型中选择“静态库（.lib）。

6.编译生成新的解决方案，在 Debug 文件夹下会得到 mylib.lib (对象文件库），将该.lib 文件和相应头文件给用户，用户就可以使用该库里的函数了。

12.2.2 静态库的使用

方法一：配置项目属性

A、添加工程的头文件目录：工程---属性---配置属性---c/c++---常规---附加包含目录：加上头文件存放目录。

B、添加文件引用的 lib 静态库路径：工程---属性---配置属性---链接器---常规---附加库目录：加上 lib 文件存放目录。

C　然后添加工程引用的 lib 文件名 :工程---属性---配置属性---链接器---输入---附加依赖项：加上 lib 文件名。

方法二：使用编译语句

```
#pragma comment(lib,"./mylib.lib")
```

方法三：添加工程中

就像你添加.h 和.c 文件一样,把 lib 文件添加到工程文件列表中去.

切换到"解决方案视图",--->选中要添加 lib 的工程-->点击右键-->"添加"-->"现有项"-->选择 lib 文件-->确定.

12.2.3 静态库优缺点

- 静态库对函数库的链接是放在编译时期完成的 ,静态库在程序的链接阶段被复制到了程序中，和程序运行的时候没有关系；
- 程序在运行时与函数库再无瓜葛，移植方便。
- 浪费空间和资源，所有相关的目标文件与牵涉到的函数库被链接合成一个可执行文件。

内存和磁盘空间

静态链接这种方法很简单，原理上也很容易理解，在操作系统和硬件不发达的早期，绝大部门系统采用这种方案。随着计算机软件的发展，这种方法的缺点很快暴露出来，那就是静态链接的方式对于计算机内存和磁盘空间浪费非常严重。特别是多进程操作系统下，静态链接极大的浪费了内存空间。在现在的 linux 系统中，一个普通程序会用到 c 语言静态库至少在 1MB 以上，那么如果磁盘中有 2000 个这样的程序，就要浪费将近 2GB 的磁盘空间。

程序开发和发布

空间浪费是静态链接的一个问题，另一个问题是静态链接对程序的更新、部署和发布也会带来很多麻烦。比如程序中所使用的 mylib.lib 是由一个第三方厂商提供的，当该厂商更新容量 mylib.lib 的时候，那么我们的程序就要拿到最新版的 mylib.lib，然后将其重新编译

链接后，将新的程序整个发布给用户。这样的做缺点很明显，即一旦程序中有任何模块更新，整个程序就要重新编译链接、发布给用户，用户要重新安装整个程序。

12.3 windows 下动态库创建和使用

要解决空间浪费和更新困难这两个问题，最简单的办法就是把程序的模块相互分割开来，形成独立的文件，而不是将他们静态的链接在一起。简单地讲，就是不对哪些组成程序的目标程序进行链接，等程序运行的时候才进行链接。也就是说，把整个链接过程推迟到了运行时再进行，这就是动态链接的基本思想。

12.3.1 动态库的创建

1. 创建一个新项目，在已安装的模板中选择“常规”，在右边的类型下选择“空项目”，在名称和解决方案名称中输入 mydll。点击确定。

2.在解决方案资源管理器的头文件中添加,mydll.h 文件，在源文件添加 mydll.c 文件（即实现文件）。

3.在 test.h 文件中添加如下代码：

```
#ifndef TEST_H
#define TEST_H

__declspec(dllexport) int myminus(int a, int b);


#endif
```

5.在 test.c 文件中添加如下代码：

```
#include"test.h"
__declspec(dllexport) int myminus(int a, int b){
```

```
    return a - b;

}
```

5. 配置项目属性。因为这是一个动态链接库，所以应在项目属性的“配置属性”下选择“常规”，在其下的配置类型中选择“动态库（.dll）。

6.编译生成新的解决方案，在 Debug 文件夹下会得到 mydll.dll (对象文件库），将该.dll 文件、.lib 文件和相应头文件给用户，用户就可以使用该库里的函数了。

疑问一：__declspec(dllexport)是什么意思？

动态链接库中定义有两种函数：导出函数(export function)和内部函数(internal function)。 导出函数可以被其它模块调用，内部函数在定义它们的 DLL 程序内部使用。

疑问二：动态库的 lib 文件和静态库的 lib 文件的区别？

在使用动态库的时候，往往提供两个文件：一个引入库（.lib）文件（也称“导入库文件”）和一个 DLL（.dll）文件。虽然引入库的后缀名也是“lib”，但是，动态库的引入库文件和静态库文件有着本质的区别，对一个 DLL 文件来说，其引入库文件（.lib）包含该 DLL 导出的函数和变量的符号名，而.dll 文件包含该 DLL 实际的函数和数据。在使用动态库的情况下，在编译链接可执行文件时，只需要链接该 DLL 的引入库文件，该 DLL 中的函数代码和数据并不复制到可执行文件，直到可执行程序运行时，才去加载所需的 DLL，将该 DLL 映射到进程的地址空间中，然后访问 DLL 中导出的函数。

12.3.2 动态库的使用

方法一：隐式调用

创建主程序 TestDll，将 mydll.h、mydll.dll 和 mydll.lib 复制到源代码目录下。

(P.S：头文件 Func.h 并不是必需的，只是 C++中使用外部函数时，需要先进行声明)

在程序中指定链接引用链接库 ：#pragma comment(lib,"./mydll.lib")

方法二：显式调用

```
    HANDLE hDll; //声明一个 dll 实例文件句柄
    hDll = LoadLibrary("mydll.dll"); //导入动态链接库
    MYFUNC minus_test; //创建函数指针
    //获取导入函数的函数指针
    minus_test = (MYFUNC)GetProcAddress(hDll, "myminus");
```

12. 递归函数

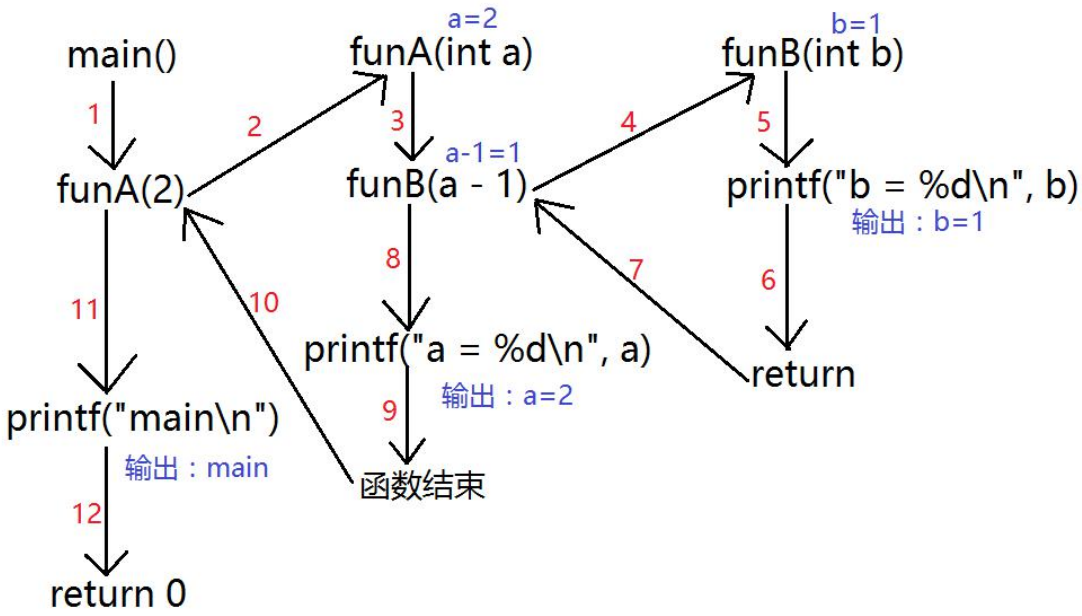
12.1 递归函数基本概念

C 通过运行时堆栈来支持递归函数的实现。递归函数就是直接或间接调用自身的函数。

12.2 普通函数调用

```
void funB(int b){
    printf("b = %d\n", b);
}

void funA(int a){
    funB(a - 1);
    printf("a = %d\n", a);
}

int main(void){
    funA(2);
    printf("main\n");
    return 0;
}
```

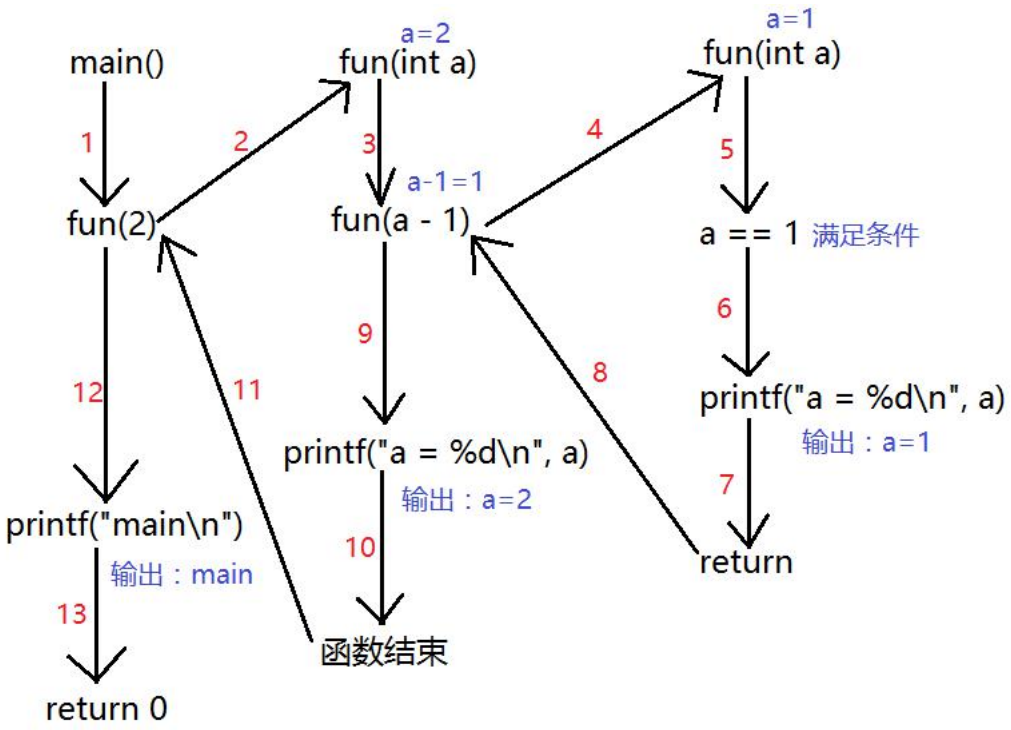
函数的调用流程如下：



12.3 递归函数调用

```c
void fun(int a){

    if (a == 1){
        printf("a = %d\n", a);
        return; //中断函数很重要
    }

    fun(a - 1);
    printf("a = %d\n", a);
}

int main(void){

    fun(2);
    printf("main\n");

    return 0;
}
```

函数的调用流程如下：



作业：

递归实现给出一个数 8793，依次打印千位数字 8、百位数字 7、十位数字 9、个位数字 3。

```
void recursion(int val){
    if (val == 0){
        return;
    }
    int ret = val / 10;
    recursion(ret);
    printf("%d ",val % 10);
}
```

12.4 递归实现字符串反转

```
int reverse1(char *str){
    if (str == NULL)
    {
        return -1;
    }
```

```
    if (*str == '\0') // 函数递归调用结束条件
    {
        return 0;
    }

    reverse1(str + 1);
    printf("%c", *str);

    return 0;
}

char buf[1024] = { 0 };  //全局变量

int reverse2(char *str){
    if (str == NULL)
    {
        return -1;
    }

    if ( *str == '\0' ) // 函数递归调用结束条件
    {
        return 0;
    }

    reverse2(str + 1);
    strncat(buf, str, 1);

    return 0;
}

int reverse3(char *str, char *dst){
    if (str == NULL || dst == NULL)
    {
        return -1;
    }

    if (*str == '\0') // 函数递归调用结束条件
    {
        return 0;
    }

    reverse3(str + 1);
```

```
    strncat(dst, str, 1);

    return 0;
}
```

12.4 递归实现链表逆序打印

13. 面向接口编程

13.1 案例背景

一般的企业信息系统都有成熟的框架。软件框架一般不发生变化，能自由的集成第三方厂商的产品。

13.2 案例需求

要求在企业信息系统框架中集成第三方厂商的 socket 通信产品和第三方厂商加密产品。软件设计要求：模块要求松、接口要求紧。

13.3 案例要求

1）能支持多个厂商的 socket 通信产品入围

2）能支持多个第三方厂商加密产品的入围

3）企业信息系统框架不轻易发生框架

13.4 编程提示

1）抽象通信接口结构体设计（CSocketProtocol）

2）框架接口设计（framework）

3） a) 通信厂商 1 入围（CSckImp1）

b) 通信厂商 2 入围（CSckImp2）

4） a) 抽象加密接口结构体设计（CEncDesProtocol）

b) 升级框架函数（增加加解密功能）

c) 加密厂商 1 入围(CHwImp)、加密厂商 2 入围(CCiscoImp)

5）框架接口分文件