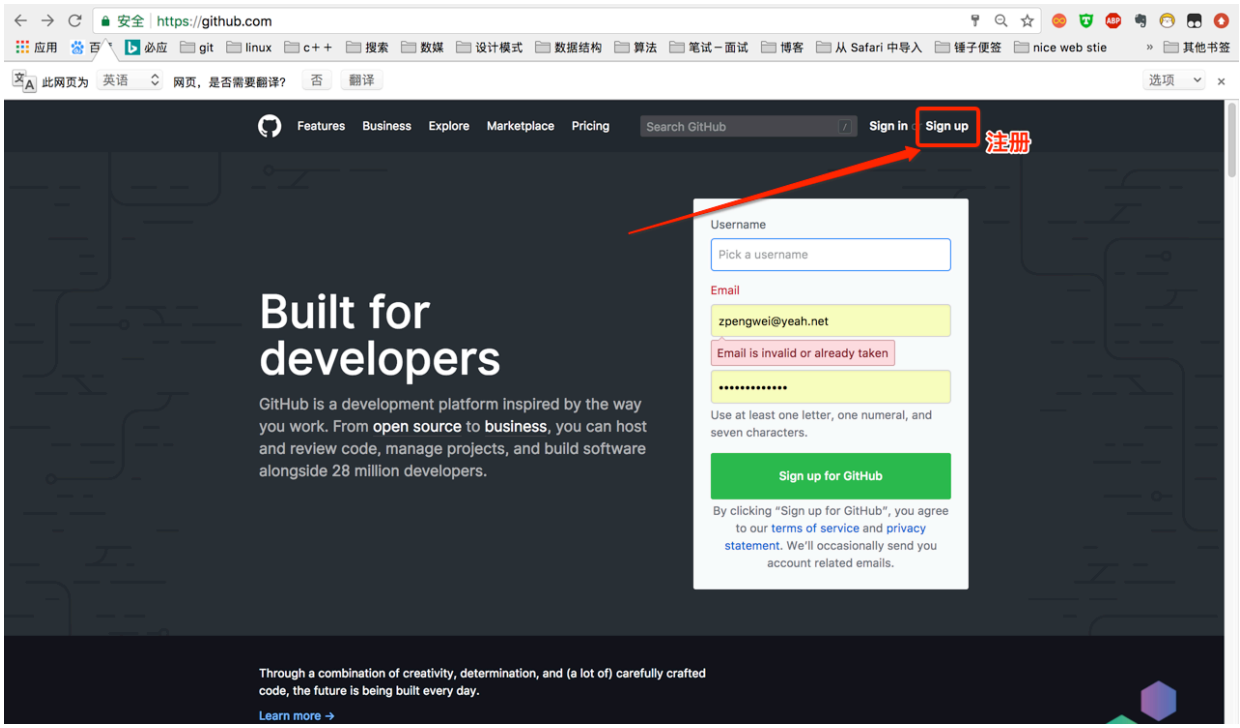
github注册文档

比特科技提供@版权

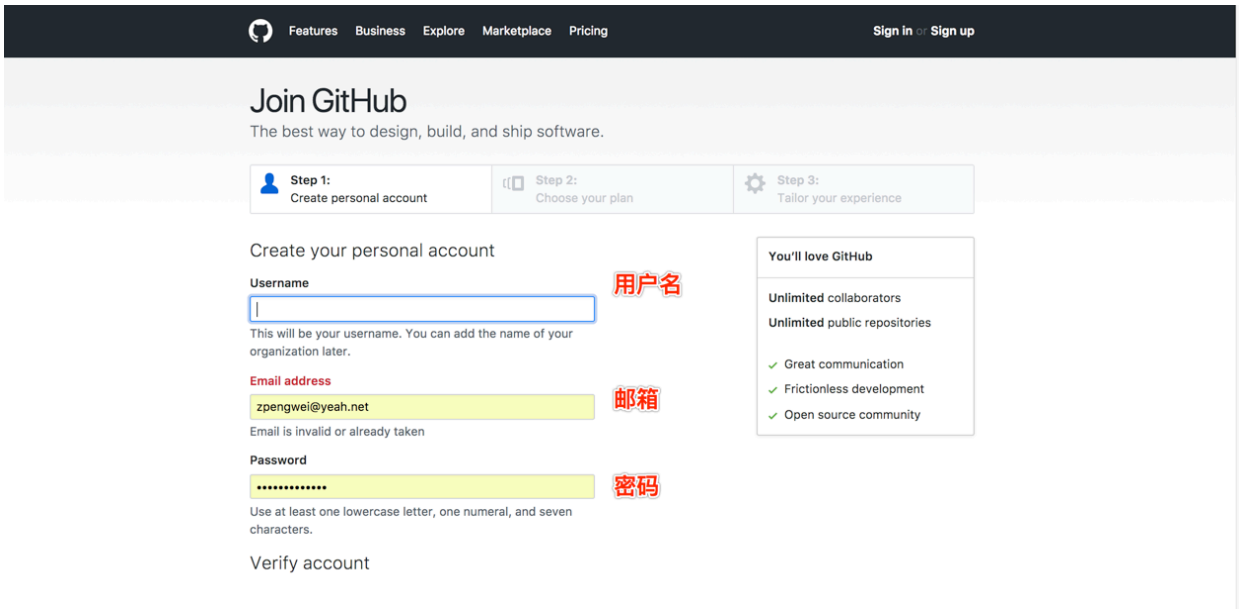
第一步：打开www.github.com

会看到如下界面

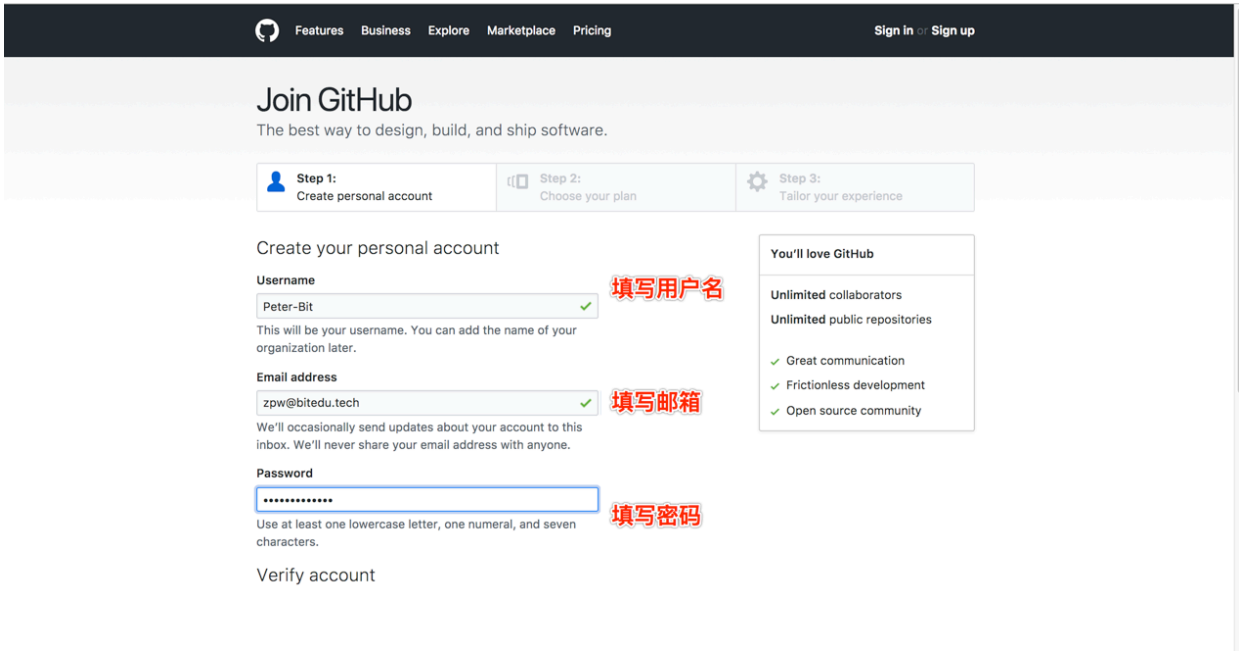


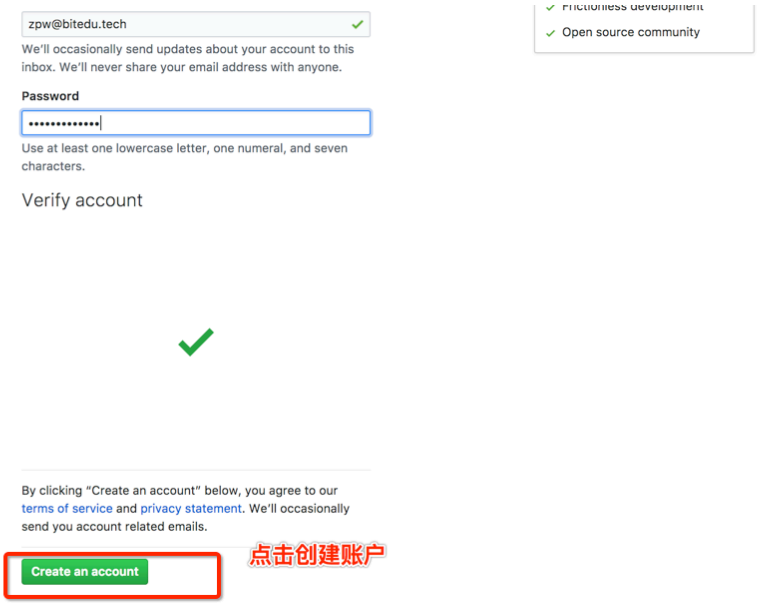
第二步：选择【sign up】

看到如下界面

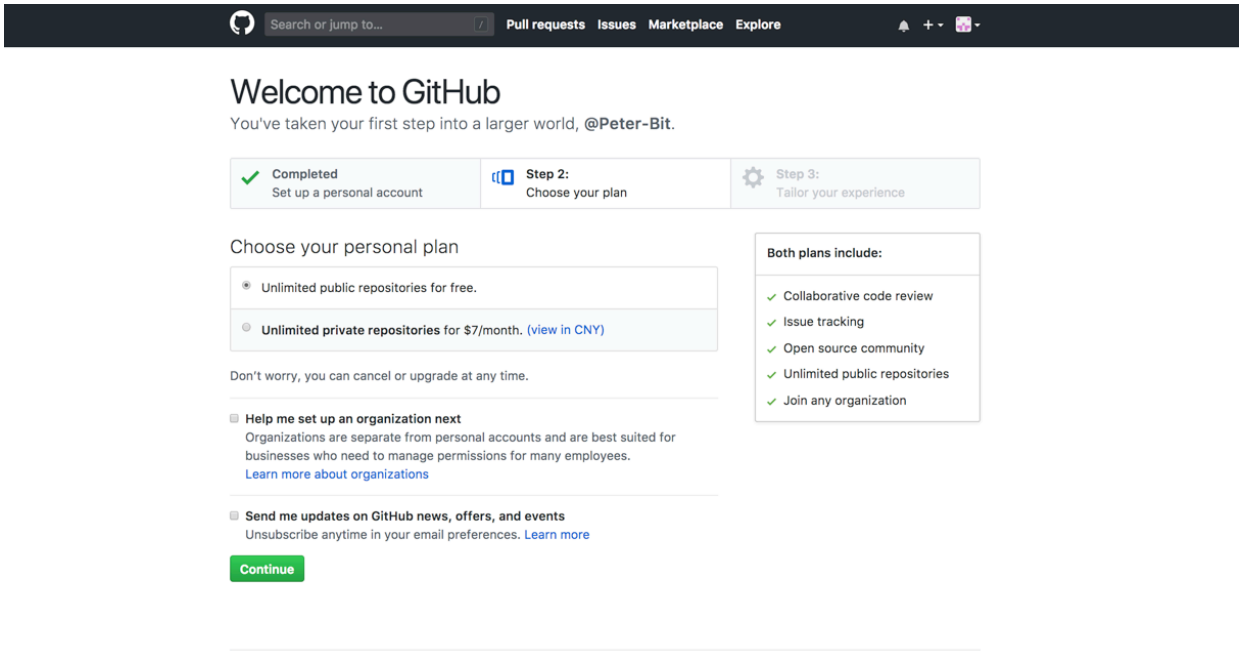

Features Business Explore Marketplace Pricing
Sign in or Sign up
Join GitHub
The best way to design, build, and ship software.
Step 1: Create personal account
Step 2: Choose your plan
Step 3: Tailor your experience
Create your personal account
Username
用户名
This will be your username. You can add the name of your organization later.
Email address
zpengwei@yeah.net
邮箱
Email is invalid or already taken
Password
密码
Use at least one lowercase letter, one numeral, and seven characters.
Verify account
You'll love GitHub
Unlimited collaborators
Unlimited public repositories
Great communication
Frictionless development
Open source community

第三步：填写账户信息：

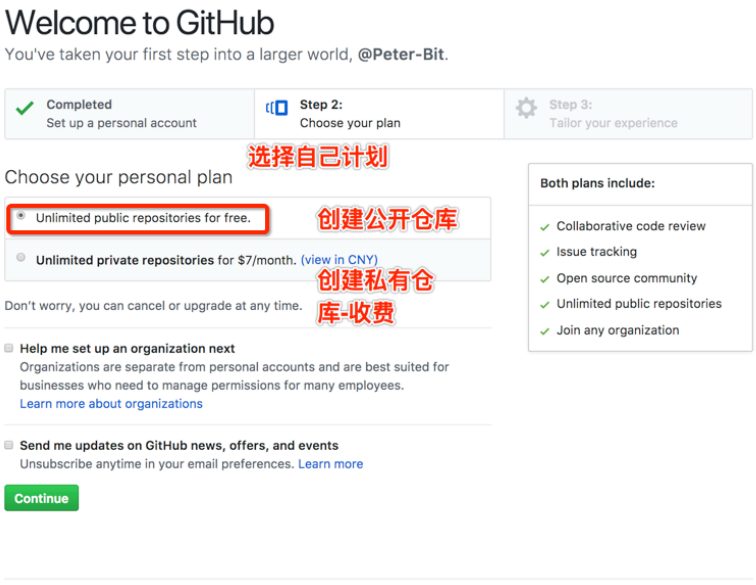

Features Business Explore Marketplace Pricing
Sign in or Sign up
Join GitHub
The best way to design, build, and ship software.
Step 1: Create personal account
Step 2: Choose your plan
Step 3: Tailor your experience
Create your personal account
Username
Peter-Bit
填写用户名
This will be your username. You can add the name of your organization later.
Email address
zpw@bitedu.tech
填写邮箱
We'll occasionally send updates about your account to this inbox. We'll never share your email address with anyone.
Password
填写密码
Use at least one lowercase letter, one numeral, and seven characters.
Verify account
You'll love GitHub
Unlimited collaborators
Unlimited public repositories
Great communication
Frictionless development
Open source community



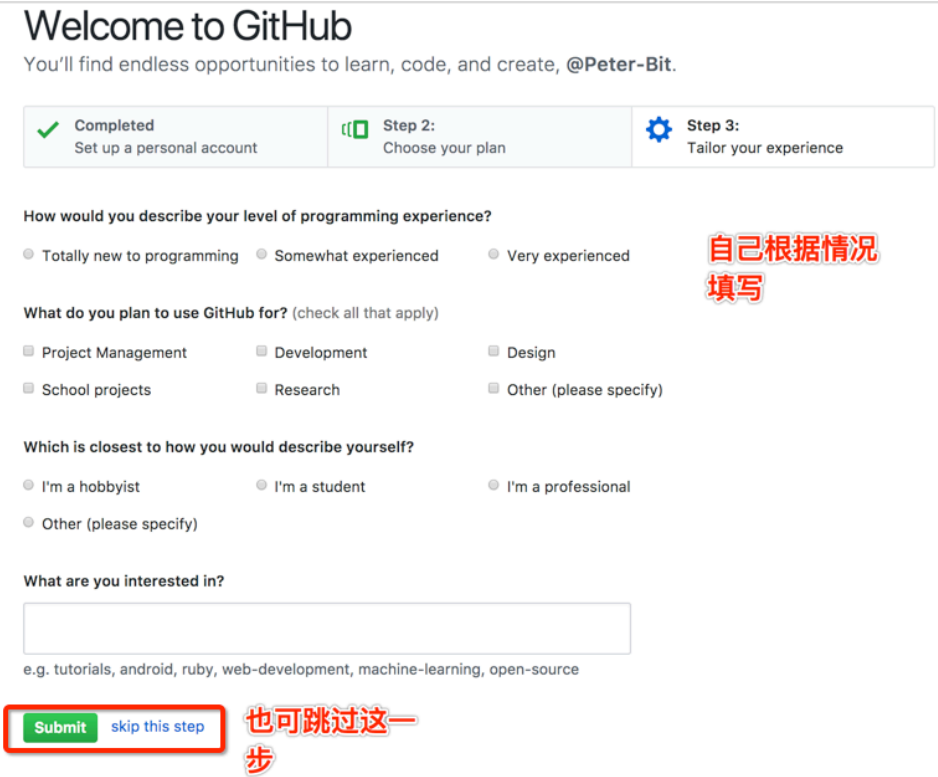
第四步：点击【Create an account】之后



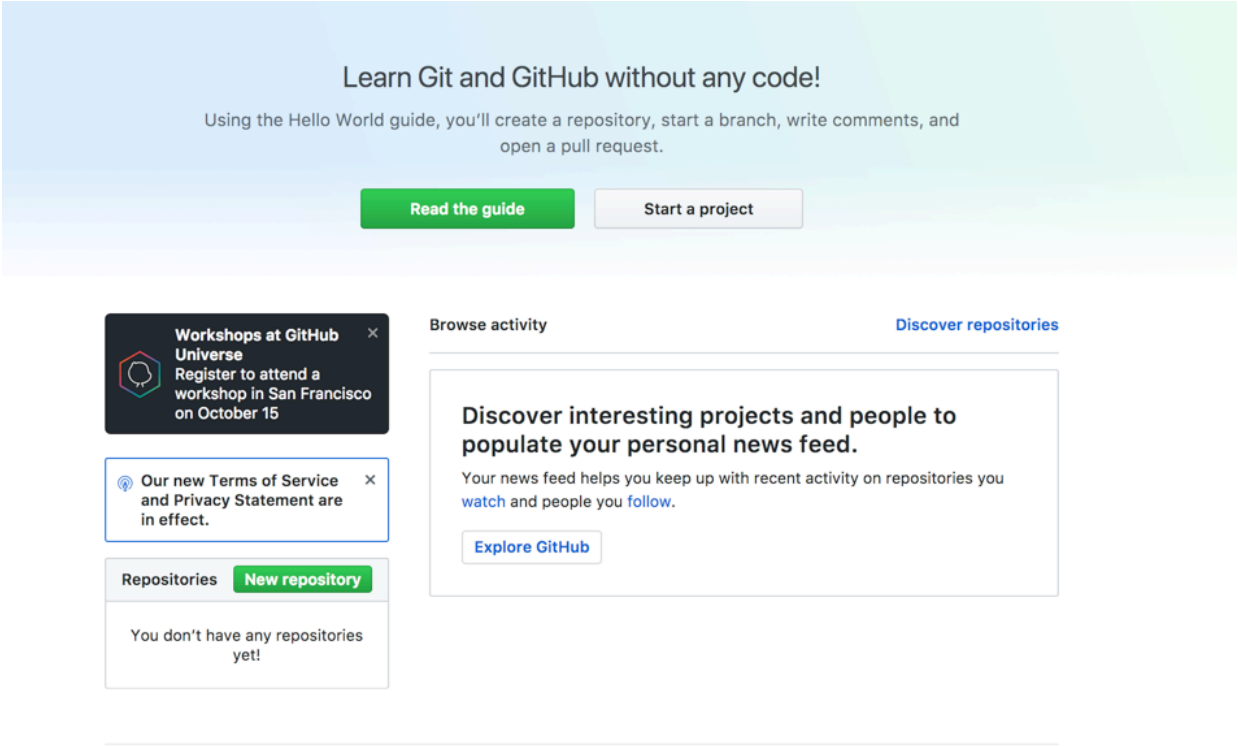
第五步：选择共有仓库



第六步：点击【Continue】继续



第七步：点击【Submit】提交



恭喜你：github账户创建成功

BIT-0-C语言课程课前准备

正文开始@比特就业课

---

1. 自我介绍

比特讲师

2. 创建QQ群、微信群

> 1. QQ群一般开课前都创建了，问学生加进去没?
> 2. QQ群的作用：消息的通知，课表，尽量不要屏蔽。
> 3. 文件共享：课件、代码、视频。
> 4. 后期建微信群

3. 作业提交：

> **按时提交**
> 交作业使用教务系统：https://v.bitedu.vip/login
>
> 下节课上课之前最好能搞定。

4. 关于博客

养成写博客的习惯。

博客的重要性：

1. 自己写博客，是对所学知识的总结
2. 写博客可以记录你学习的一个过程和心得，给面试官更多了解你的机会，同时增加面试的谈资。
3. 写博客说明你是一个愿意分享的人。

> CSDN - 推荐 https://blog.csdn.net/

后期技术好了，可以自己搭建博客~

5. github - gitee 的重要性。

github网址：https://github.com/ （国外/网络要求高/访问慢）

gitee网址：https://gitee.com/ （国内/访问速度快）

1. 大公司喜欢的东西
2. 下去先了解，再注册一下github账号
3. 慢慢扩展学习 git 教程链接 （不能只懂三板斧）

6. 一些有逼格的工具

- 印象笔记（有道云笔记）-笔记可以检索，笔记丢不了，随时随地方便复习。
- xmind-思维导图，整理一门课程学完后的框架。

7. 如何学好C语言

7.1 鼓励你，为你叫好。

> C生万物
>
> 编程之本
>
> 长远IT职业发展的首选
>
> C语言是母体语言，是人机交互接近底层的桥梁
>
> 学会C/C++，相当于掌握技术核心
>
> 知识点一竿子打通。
>
> IT行业，一般每10年就有一次变革
>
> 近50年间，在TIOBE 排行榜中，C/C++位置长期霸占前三名，没有丝毫撼动，可谓经典永不过时！

7.2 挤时间学习

- 欲戴王冠，必承其重。
- 如果你总是和别人走一样的路怎么才能保证超越别人，那就得付出不一样的努力。

7.3 拒绝做伸手党

- 遇到问题，先尝试自己解决

7.4 学好编程，不仅仅是学好C语言

必须要学好：

> **计算机语言、算法和数据结构、操作系统、计算机网络、项目实战**

7.5 利用好比特的学习资源

- 不懂就要问

关注：【比特就业课】公众号，后期找工作要使用。

BIT-1-初识C语言

> 基本了解C语言的基础知识，对C语言有一个大概的认识。
>
> 每个知识点就是简单认识，不做详细讲解，后期课程都会细讲。

本章重点：

- 什么是C语言
- 第一个C语言程序
- 数据类型
- 变量、常量
- 字符串+转义字符+注释
- 选择语句
- 循环语句
- 函数
- 数组
- 操作符
- 常见关键字
- define 定义常量和宏
- 指针
- 结构体

正文开始@比特就业课

1. 什么是C语言?

> C语言是一门通用计算机编程语言，广泛应用于底层开发。C语言的设计目标是提供一种能以简易的方式编译、处理低级存储器、产
>
> 生少量的机器码以及不需要任何运行环境支持便能运行的编程语言。

> 尽管C语言提供了许多低级处理的功能，但仍然保持着良好跨平台的特性，以一个标准规格写出的C语言程序可在许多电脑平台上进
>
> 行编译，甚至包含一些嵌入式处理器（单片机或称MCU）以及超级电脑等作业平台。

> 二十世纪八十年代，为了避免各开发厂商用的C语言语法产生差异，由美国国家标准局为C语言制定了一套完整的美国国家标准语
>
> 法，称为ANSI C，作为C语言最初的标准。 [1] 目前2011年12月8日，国际标准化组织（ISO）和国际电工委员会（IEC）发布的**C11**
>
> **标准**是C语言的第三个官方标准，也是C语言的最新标准，该标准更好的支持了汉字函数名和汉字标识符，一定程度上实现了汉
>
> 字编程。

> C语言是一门面向过程的计算机编程语言，与C++，Java等面向对象的编程语言有所不同。

> 其编译器主要有Clang、**GCC**、WIN-TC、SUBLIME、**MSVC**、Turbo C等。

2. 第一个C语言程序

```c
#include <stdio.h>

int main()
{
    printf("hello bit\n");
    printf("he he\n");
    return 0;
}
//解释:
//main函数是程序的入口
//一个工程中main函数有且仅有一个
```

3. 数据类型

```c
char        //字符数据类型
short       //短整型
int         //整形
long        //长整型
long long   //更长的整形
float       //单精度浮点数
double      //双精度浮点数
//C语言有没有字符串类型?
```

- 为什么出现这么的类型?
- 每种类型的大小是多少?

```c
#include <stdio.h>
int main()
{
    printf("%d\n", sizeof(char));
    printf("%d\n", sizeof(short));
    printf("%d\n", sizeof(int));
    printf("%d\n", sizeof(long));
    printf("%d\n", sizeof(long long));
    printf("%d\n", sizeof(float));
    printf("%d\n", sizeof(double));
    printf("%d\n", sizeof(long double));
    return 0;
}
```

注意:存在这么多的类型,其实是为了更加丰富的表达生活中的各种值。

类型的使用:

```c
char ch = 'w';
int weight = 120;
int salary = 20000;
```

3. 变量、常量

生活中的有些值是不变的(比如:圆周率,性别,身份证号码,血型等等)

有些值是可变的(比如:年龄,体重,薪资)。

不变的值，C语言中用**常量**的概念来表示，变得值C语言中用**变量**来表示。

3.1 定义变量的方法

```
int age = 150;
float weight = 45.5f;
char ch = 'w';
```

3.2 变量的分类

- 局部变量
- 全局变量

```
#include <stdio.h>

int global = 2019;//全局变量
int main()
{
    int local = 2018;//局部变量
    //下面定义的global会不会有问题?
    int global = 2020;//局部变量
    printf("global = %d\n", global);
    return 0;
}
```

总结:

> 上面的局部变量global变量的定义其实没有什么问题的!
>
> 当局部变量和全局变量同名的时候，局部变量优先使用。

3.3 变量的使用

```
#include <stdio.h>
int main()
{
    int num1 = 0;
    int num2 = 0;
    int sum = 0;
    printf("输入两个操作数:>");
    scanf("%d %d", &num1, &num2);
    sum = num1 + num2;
    printf("sum = %d\n", sum);
    return 0;
}
//这里介绍一下输入，输出语句
//scanf
//printf
```

3.4 变量的作用域和生命周期

作用域

> 作用域（scope）是程序设计概念，通常来说，一段程序代码中所用到的名字并不总是有效/可用的
>
> 而限定这个名字的可用性的代码范围就是这个名字的作用域。

1. 局部变量的作用域是变量所在的局部范围。
2. 全局变量的作用域是整个工程。

生命周期

> 变量的生命周期指的是变量的创建到变量的销毁之间的一个时间段

1. 局部变量的生命周期是：进入作用域生命周期开始，出作用域生命周期结束。
2. 全局变量的生命周期是：整个程序的生命周期。

3.5 常量

C语言中的常量和变量的定义的形式有所差异。

C语言中的常量分为以下以下几种：

- 字面常量
- `const` 修饰的常变量
- #define 定义的标识符常量
- 枚举常量

```
#include <stdio.h>
//举例
enum Sex
{
    MALE,
    FEMALE,
    SECRET
};
//括号中的MALE,FEMALE,SECRET是枚举常量

int main()
{
    //字面常量演示
    3.14;//字面常量
    1000;//字面常量

    //const 修饰的常变量
    const float pai = 3.14f;    //这里的pai是const修饰的常变量
    pai = 5.14;//是不能直接修改的!

    //#define的标识符常量 演示
    #define MAX 100
    printf("max = %d\n", MAX);

    //枚举常量演示
    printf("%d\n", MALE);
    printf("%d\n", FEMALE);
    printf("%d\n", SECRET);
    //注：枚举常量的默认是从0开始，依次向下递增1的
```

```
    return 0;
}
```

注：

上面例子上的`pai`被称为`const`修饰的常变量，`const`修饰的常变量在C语言中只是在语法层面限制了变量`pai`不能直接被改变，但是`pai`本质上还是一个变量的，所以叫常变量。

4. 字符串+转义字符+注释

4.1 字符串

```
"hello bit.\n"
```

这种由双引号（Double Quote）引起来的一串字符称为字符串字面值（String Literal），或者简称字符串。

注：字符串的结束标志是一个`\0`的转义字符。在计算字符串长度的时候`\0`是结束标志，不算作字符串内容。

```
#include <stdio.h>
//下面代码，打印结果是什么？为什么？（突出'\0'的重要性）
int main()
{
    char arr1[] = "bit";
    char arr2[] = {'b', 'i', 't'};
    char arr3[] = {'b', 'i', 't',  '\0'};
    printf("%s\n", arr1);
    printf("%s\n", arr2);
    printf("%s\n", arr3);
    return 0;
}
```

4.2 转义字符

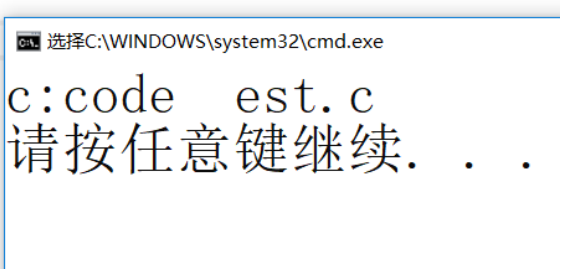
加入我们要在屏幕上打印一个目录：`c:\code\test.c`

我们该如何写代码？

```
#include <stdio.h>

int main()
{
    printf("c:\code\test.c\n");
    return 0;
}
```

实际上程序运行的结果是这样的：

```c
#include <stdio.h>
int main()
{
    printf("c:\code\test.c\n");
    return 0;
}
```



这里就不得不提一下转义字符了。转义字符顾名思义就是转变意思。

下面看一些转义字符。

转义字符	释义
\?	在书写连续多个问号时使用，防止他们被解析成三字母词
\'	用于表示字符常量'
\“	用于表示一个字符串内部的双引号
\\	用于表示一个反斜杠，防止它被解释为一个转义序列符。
\a	警告字符，蜂鸣
\b	退格符
\f	进纸符
\n	换行
\r	回车
\t	水平制表符
\v	垂直制表符
\ddd	ddd表示1~3个八进制的数字。 如： \130 X
\xdd	dd表示2个十六进制数字。 如： \x30 0

```c
#include <stdio.h>
int main()
{
    //问题1：在屏幕上打印一个单引号'，怎么做？
    //问题2：在屏幕上打印一个字符串，字符串的内容是一个双引号“，怎么做？
    printf("%c\n", '\'');
    printf("%s\n", "\"");
    return 0;
}
```

笔试题：

```c
//程序输出什么？
#include <stdio.h>

int main()
{
    printf("%d\n", strlen("abcdef"));
    // \62被解析成一个转义字符
    printf("%d\n", strlen("c:\test\628\test.c"));
    return 0;
}
```

5. 注释

1. 代码中有不需要的代码可以直接删除，也可以注释掉
2. 代码中有些代码比较难懂，可以加一下注释文字

比如：

```c
#include <stdio.h>

int Add(int x, int y)
{
    return x+y;
}
/*C语言风格注释
int Sub(int x, int y)
{
    return x-y;
}
*/
int main()
{
    //C++注释风格
    //int a = 10;
    //调用Add函数，完成加法
    printf("%d\n", Add(1, 2));
    return 0;
}
```

注释有两种风格：

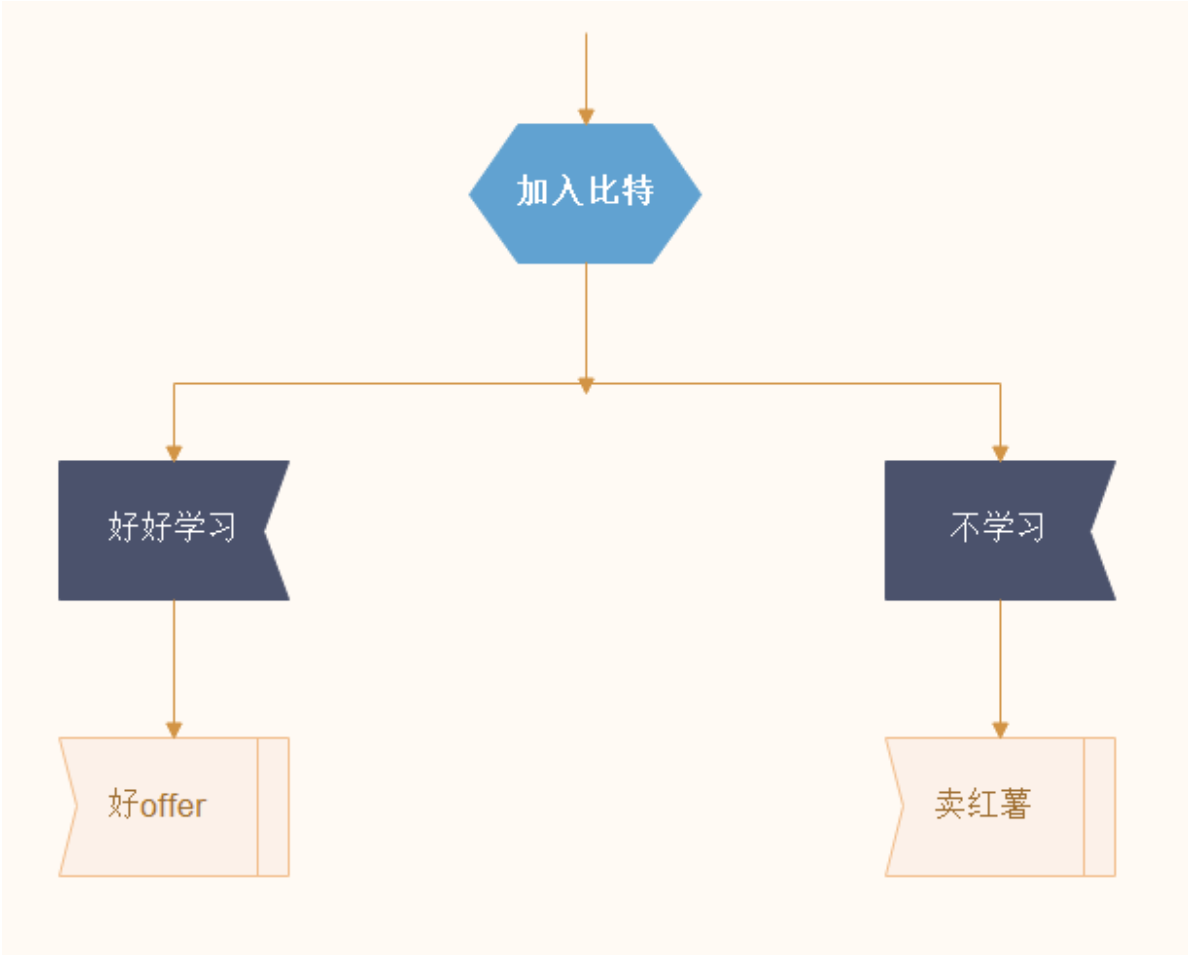
- C语言风格的注释 `/*xxxxxx*/`
  - 缺陷：不能嵌套注释
- C++风格的注释 `//xxxxxxxx`
  - 可以注释一行也可以注释多行

6. 选择语句

如果你好好学习，校招时拿一个好offer，走上人生巅峰。

如果你不学习，毕业等于失业，回家卖红薯。

这就是选择！

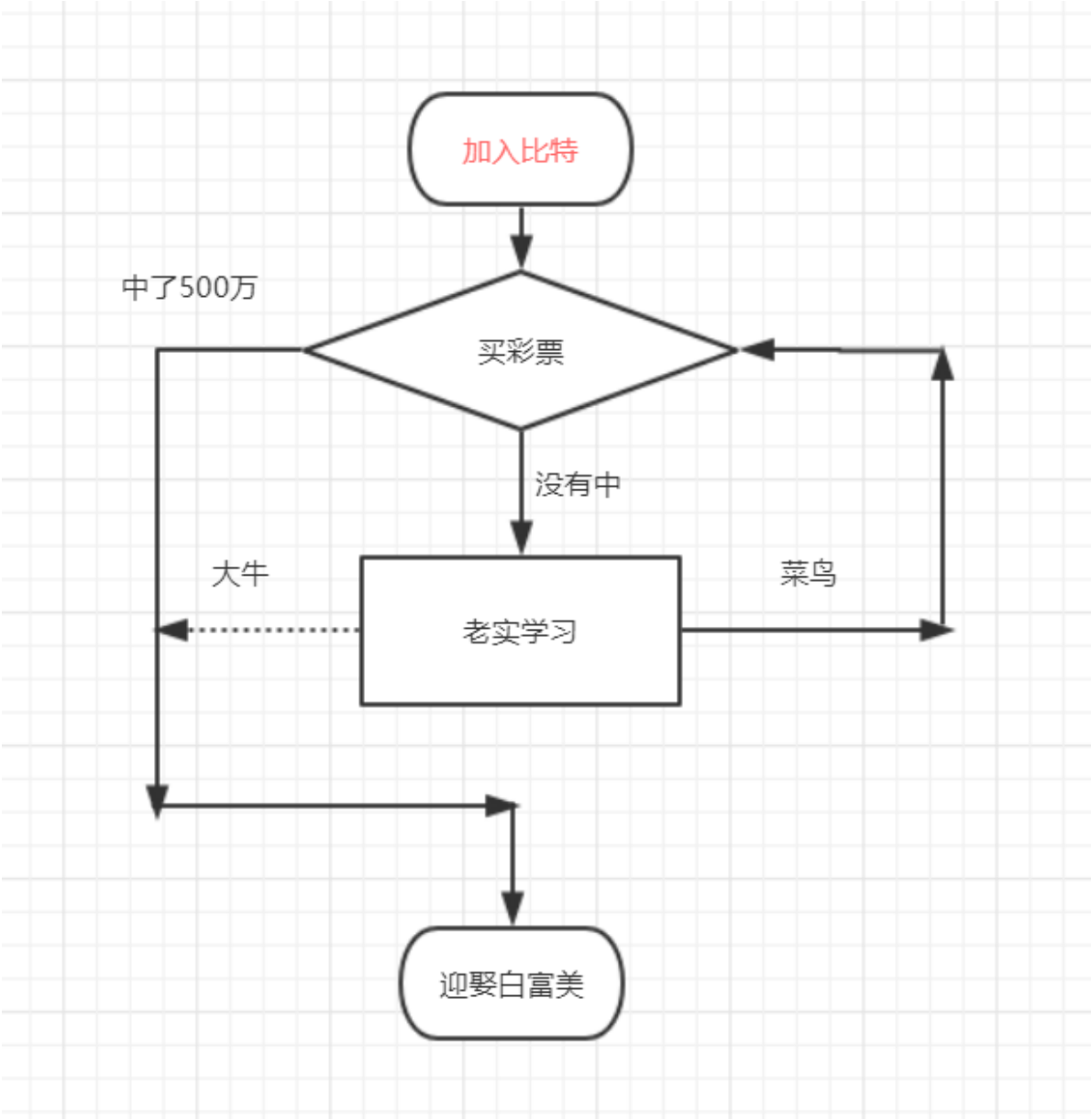



```c
#include <stdio.h>

int main()
{
    int coding = 0;
    printf("你会去敲代码吗？（选择1 or 0）:>");
    scanf("%d", &coding);
    if(coding == 1)
    {
        prinf("坚持，你会有好offer\n");
    }
    else
    {
        printf("放弃，回家卖红薯\n");
    }
    return 0;
}
```

7. 循环语句

有些事必须一直做，比如我日复一日的讲课，比如大家，日复一日的学习。

还比如：



C语言中如何实现循环呢?

- while语句-讲解
- for语句（后期讲）
- do ... while语句（后期讲）

```c
//while循环的实例
#include <stdio.h>

int main()
{
    printf("加入比特\n");
    int line = 0;
    while(line<=20000)
    {
        line++;
        printf("我要继续努力敲代码\n");
    }
    if(line>20000)
        printf("好offer\n");
    return 0;
}
```

8. 函数

```c
#include <stdio.h>

int main()
{
    int num1 = 0;
    int num2 = 0;
    int sum = 0;
    printf("输入两个操作数:>");
    scanf("%d %d", &num1, &num2);
    sum = num1 + num2;
    printf("sum = %d\n", sum);
    return 0;
}
```

上述代码，写成函数如下：

```c
#include <stdio.h>

int Add(int x, int y)
{
    int z = x+y;
    return z;
}

int main()
{
    int num1 = 0;
    int num2 = 0;
    int sum = 0;
    printf("输入两个操作数:>");
    scanf("%d %d", &num1, &num2);
    sum = Add(num1, num2);
    printf("sum = %d\n", sum);
    return 0;
}
```

函数的特点就是简化代码，代码复用。

9. 数组

要存储1-10的数字，怎么存储?

C语言中给了数组的定义：一组相同类型元素的集合

9.1 数组定义

```c
int arr[10] = {1,2,3,4,5,6,7,8,9,10};//定义一个整形数组，最多放10个元素
```

9.2 数组的下标

C语言规定：数组的每个元素都有一个下标，下标是从0开始的。

数组可以通过下标来访问的。

比如：

```
int arr[10] = {0};
//如果数组10个元素，下标的范围是0-9
```

int arr[10]	0	0	0	0	0	0	0	0	0	0
下标	0	1	2	3	4	5	6	7	8	9

9.3 数组的使用

```
#include <stdio.h>

int main()
{
    int i = 0;
    int arr[10] = {1,2,3,4,5,6,7,8,9,10};
    for(i=0; i<10; i++)
    {
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

10. 操作符

简单介绍为主，后面课件重点讲。

> 算术操作符

```
+   -   *   /   %
```

> 移位操作符

```
>>  <<
```

> 位操作符

```
&   ^   |
```

赋值操作符

```
=   +=   -=   *=   /=   &=   ^=   |=   >>=   <<=
```

单目操作符

```
!           逻辑反操作
-           负值
+           正值
&           取地址
sizeof      操作数的类型长度（以字节为单位）
~           对一个数的二进制按位取反
--          前置、后置--
++          前置、后置++
*           间接访问操作符(解引用操作符)
(类型)      强制类型转换
```

关系操作符

```
>
>=
<
<=
!=      用于测试“不相等”
==      用于测试“相等”
```

逻辑操作符

```
&&          逻辑与
||          逻辑或
```

条件操作符

```
exp1 ? exp2 : exp3
```

逗号表达式

```
exp1, exp2, exp3, …expN
```

下标引用、函数调用和结构成员

```
[]  ()  .  ->
```

11. 常见关键字

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

注：关键字，先介绍下面几个，后期遇到讲解。

11.1 关键字 typedef

> typedef 顾名思义是类型定义，这里应该理解为类型重命名。

比如：

```
//将unsigned int 重命名为uint_32, 所以uint_32也是一个类型名
typedef unsigned int uint_32;

int main()
{
    //观察num1和num2,这两个变量的类型是一样的
    unsigned int num1 = 0;
    uint_32 num2 = 0;
    return 0;
}
```

11.2 关键字static

> 在C语言中：
>
> static是用来修饰变量和函数的
>
> 1. 修饰局部变量-称为静态局部变量
> 2. 修饰全局变量-称为静态全局变量
> 3. 修饰函数-称为静态函数

11.2.1 修饰局部变量

```
//代码1
#include <stdio.h>
void test()
{
    int i = 0;
    i++;
    printf("%d ", i);
}
int main()
{
    int i = 0;
    for(i=0; i<10; i++)
    {
        test();
    }
    return 0;
}
//代码2
#include <stdio.h>
void test()
{
    //static修饰局部变量
    static int i = 0;
    i++;
    printf("%d ", i);
}
int main()
{
```

```
    int i = 0;
    for(i=0; i<10; i++)
    {
        test();
    }
    return 0;
}
```

对比代码1和代码2的效果理解static修饰局部变量的意义。

结论：

> static修饰局部变量改变了变量的生命周期
>
> 让静态局部变量出了作用域依然存在，到程序结束，生命周期才结束。

11.2.2 修饰全局变量

```
//代码1
//add.c
int g_val = 2018;
//test.c
int main()
{
    printf("%d\n", g_val);
    return 0;
}

//代码2
//add.c
static int g_val = 2018;
//test.c
int main()
{
    printf("%d\n", g_val);
    return 0;
}
```

代码1正常，代码2在编译的时候会出现连接性错误。

结论：

> 一个全局变量被static修饰，使得这个全局变量只能在本源文件内使用，不能在其他源文件内使用。

11.2.3 修饰函数

```
//代码1
//add.c
int Add(int x, int y)
{
    return c+y;
}
//test.c
int main()
{
    printf("%d\n", Add(2, 3));
    return 0;
```

```
}

//代码2
//add.c
static int Add(int x, int y)
{
    return c+y;
}
//test.c
int main()
{
    printf("%d\n", Add(2, 3));
    return 0;
}
```

代码1正常，代码2在编译的时候会出现连接性错误.

结论：

> 一个函数被static修饰，使得这个函数只能在本源文件内使用，不能在其他源文件内使用。

剩余关键字后续课程中陆续会讲解。

12. #define 定义常量和宏

```
//define定义标识符常量
#define MAX 1000

//define定义宏
#define ADD(x, y) ((x)+(y))

#include <stdio.h>

int main()
{
    int sum = ADD(2, 3);
    printf("sum = %d\n", sum);

    sum = 10*ADD(2, 3);
    printf("sum = %d\n", sum);

    return 0;
}
```

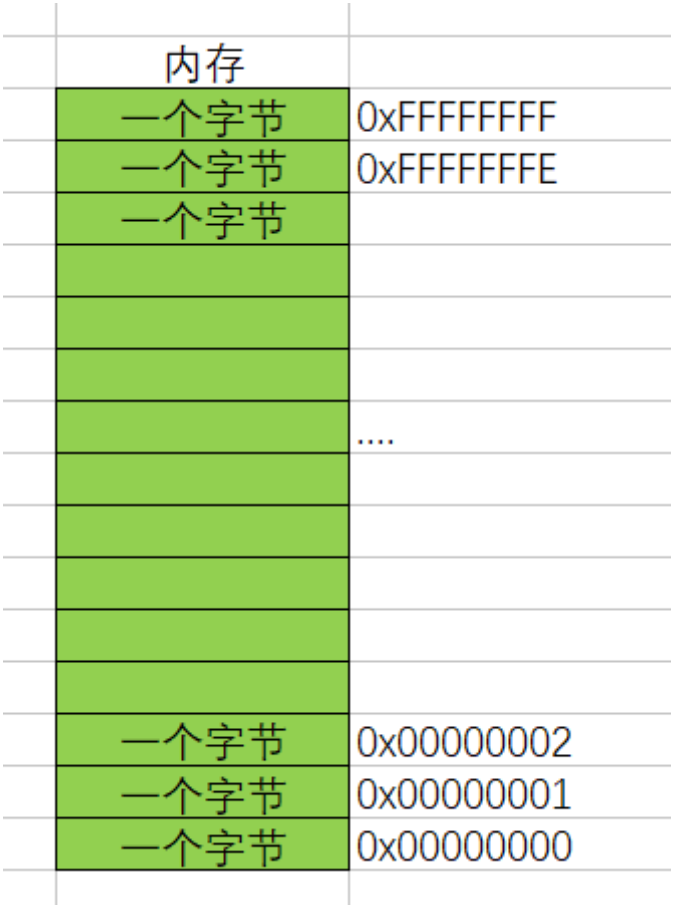
13. 指针

13.1 内存

内存是电脑上特别重要的存储器，计算机中程序的运行都是在内存中进行的 。

所以为了有效的使用内存，就把内存划分成一个个小的内存单元，每个内存单元的大小是**1个字节。**

为了能够有效的访问到内存的每个单元，就给内存单元进行了编号，这些编号被称为该**内存单元的地址。**

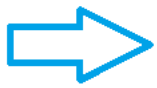


变量是创建内存中的（在内存中分配空间的），每个内存单元都有地址，所以变量也是有地址的。

取出变量地址如下：

```c
#include <stdio.h>

int main()
{
    int num = 10;
    &num;//取出num的地址
    //注：这里num的4个字节，每个字节都有地址，取出的是第一个字节的地址（较小的地址）
    printf("%p\n", &num);//打印地址，%p是以地址的形式打印
    return 0;
}
```

```c
int main()
{
    int num = 10;
    &num;//取出num的地址
    printf("%p\n", &num);
    return 0;
}
```



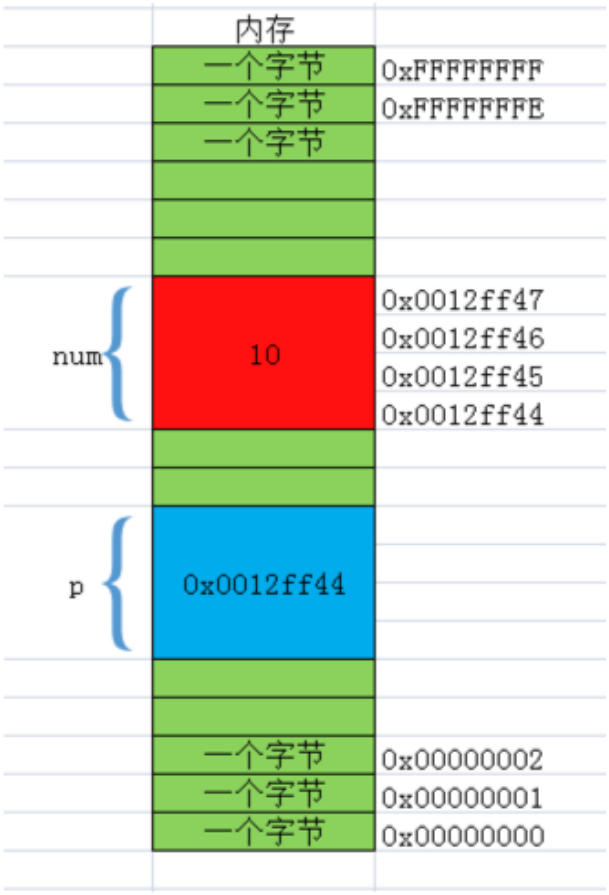
那地址如何存储，需要定义指针变量。

```c
int num = 10;
int *p;//p为一个整形指针变量
p = &num;
```

指针的使用实例：

```c
#include <stdio.h>

int main()
{
    int num = 10;
    int *p = &num;
    *p = 20;
    return 0;
}
```

```c
#include <stdio.h>
int main()
{
    int num = 10;
    int *p = &num;
    *p = 20;
    return 0;
}
```



以整形指针举例，可以推广到其他类型，如：

```c
#include <stdio.h>

int main()
{
    char ch = 'w';
    char* pc = &ch;
    *pc = 'q';
    printf("%c\n", ch);
    return 0;
}
```

13.2 指针变量的大小

```c
#include <stdio.h>
//指针变量的大小取决于地址的大小
//32位平台下地址是32个bit位（即4个字节）
//64位平台下地址是64个bit位（即8个字节）

int main()
{
    printf("%d\n", sizeof(char *));
    printf("%d\n", sizeof(short *));
    printf("%d\n", sizeof(int *));
    printf("%d\n", sizeof(double *));
    return 0;
}
```

结论：指针大小在32位平台是4个字节，64位平台是8个字节。

14. 结构体

结构体是C语言中特别重要的知识点，结构体使得C语言有能力描述复杂类型。

比如描述学生，学生包含：`名字+年龄+性别+学号`这几项信息。

这里只能使用结构体来描述了。

例如：

```
struct Stu
{
    char name[20];//名字
    int age;      //年龄
    char sex[5];  //性别
    char id[15];  //学号
};
```

结构体的初始化：

```
//打印结构体信息
struct Stu s = {"张三", 20, "男", "20180101"};

//.为结构成员访问操作符
printf("name = %s age = %d sex = %s id = %s\n", s.name, s.age, s.sex, s.id);
//->操作符
struct Stu *ps = &s;
printf("name = %s age = %d sex = %s id = %s\n", ps->name, ps->age, ps->sex, ps-
>id);
```

本章完。

BIT-2-分支语句和循环语句

> 分支语句

- if
- switch

> 循环语句

- while
- for
- do while

> goto语句

正文开始@比特就业课

1. 什么是语句?

C语句可分为以下五类：

1. 表达式语句
2. 函数调用语句
3. 控制语句
4. 复合语句
5. 空语句

本周后面介绍的是控制语句。

控制语句用于控制程序的执行流程，以实现程序的各种结构方式，它们由特定的语句定义符组成，C语言有九种控制语句。

可分成以下三类：

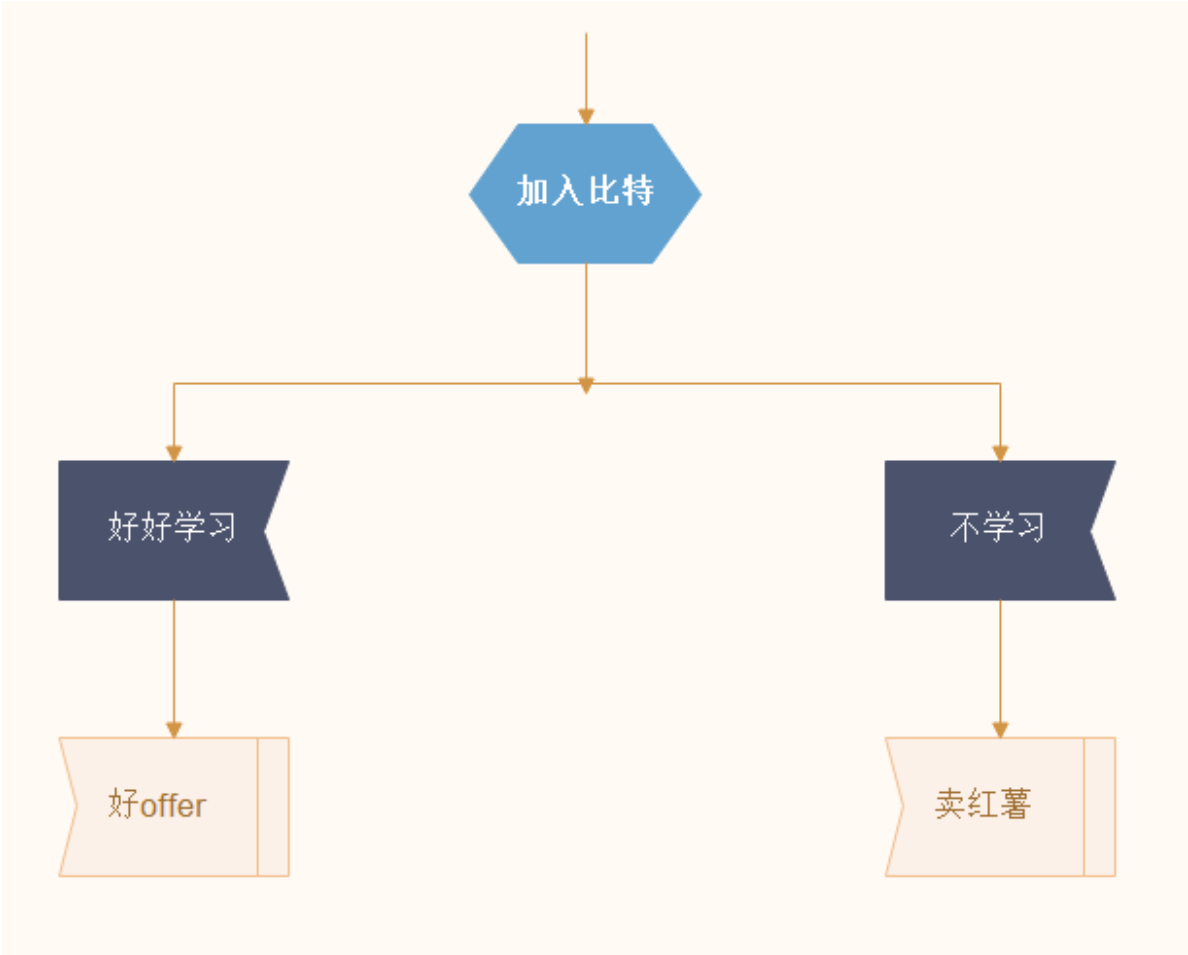
1. 条件判断语句也叫分支语句：if语句、switch语句；
2. 循环执行语句：do while语句、while语句、for语句；
3. 转向语句：break语句、goto语句、continue语句、return语句。

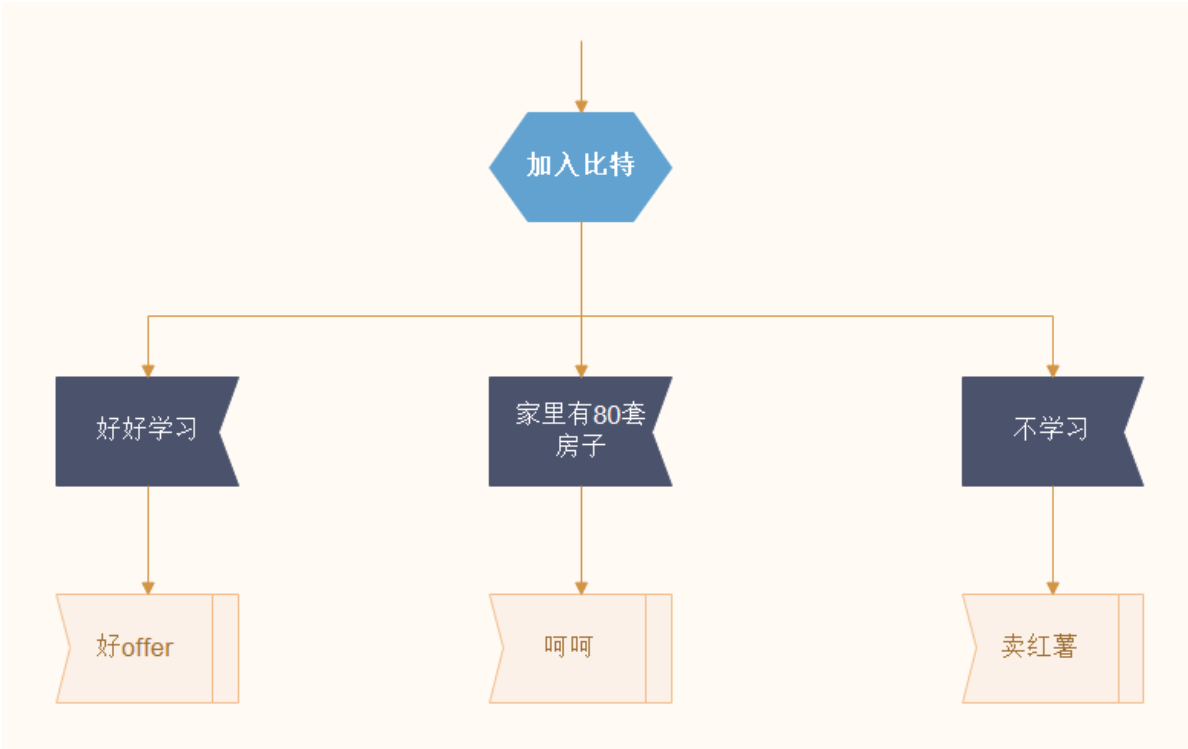
2. 分支语句（选择结构）

如果你好好学习，校招时拿一个好offer，走上人生巅峰。

如果你不学习，毕业等于失业，回家卖红薯。

这就是选择！





2.1 if语句

那if语句的语法结构是怎么样的呢?

```
语法结构:
if(表达式)
    语句;
```

```
if(表达式)
    语句1;
else
    语句2;

//多分支
if(表达式1)
    语句1;
else if(表达式2)
    语句2;
else
    语句3;
```

课堂演示代码:

```
#include <stdio.h>
//代码1
int main()
{
    int age = 0;
    scanf("%d", &age);
    if(age<18)
    {
        printf("未成年\n");
    }
}
//代码2
#include <stdio.h>
int main()
{
    int age = 0;
    scanf("%d", &age);
    if(age<18)
    {
        printf("未成年\n");
    }
    else
    {
        printf("成年\n");
    }
}
//代码3
#include <stdio.h>
int main()
{
    int age = 0;
    scanf("%d", &age);
    if(age<18)
    {
        printf("少年\n");
    }
    else if(age>=18 && age<30)
    {
        printf("青年\n");
    }
    else if(age>=30 && age<50)
    {
```

```
        printf("中年\n");
    }
    else if(age>=50 && age<80)
    {
        printf("老年\n");
    }
    else
    {
        printf("老寿星\n");
    }

}
```

解释一下：

如果表达式的结果为真，则语句执行。

在C语言中如何表示真假？

> **0表示假，非0表示真。**

如果条件成立，要执行多条语句，怎应该使用代码块。

```
#include <stdio.h>
int main()
{
    if(表达式)
    {
        语句列表1;
    }
    else
    {
        语句列表2;
    }
    return 0;
}
```

这里的一对`{ }`就是一个代码块。

2.1.1 悬空else

当你写了这个代码：

```c
#include <stdio.h>
int main()
{
    int a = 0;
    int b = 2;
    if(a == 1)
        if(b == 2)
            printf("hehe\n");
    else
        printf("haha\n");
    return 0;
}
```

改正：

```c
//适当的使用{}可以使代码的逻辑更加清楚。
//代码风格很重要
#include <stdio.h>

int main()
{
    int a = 0;
    int b = 2;
    if(a == 1)
    {
        if(b == 2)
        {
            printf("hehe\n");
        }
    }
    else
    {
         printf("haha\n");
    }
    return 0;
}
```

else的匹配：else是和它离的最近的if匹配的。

2.1.2 if书写形式的对比

```c
//代码1
if (condition) {
    return x;
}
return y;

//代码2
if(condition)
{
    return x;
}
else
{
```

```
    return y;
}

//代码3
int num = 1;
if(num == 5)
{
    printf("hehe\n");
}

//代码4
int num = 1;
if(5 == num)
{
    printf("hehe\n");
}
```

代码2和代码4更好，逻辑更加清晰，不容易出错。

2.1.3 练习

> 1. 判断一个数是否为奇数
> 2. 输出1-100之间的奇数

2.2 switch语句

switch语句也是一种分支语句。
常常用于多分支的情况。

比如：

> 输入1，输出星期一
>
> 输入2，输出星期二
>
> 输入3，输出星期三
>
> 输入4，输出星期四
>
> 输入5，输出星期五
>
> 输入6，输出星期六
>
> 输入7，输出星期日

那我没写成`if...else if ...else if`的形式太复杂，那我们就得有不一样的语法形式。

这就是switch 语句。

```
switch(整型表达式)
{
    语句项;
}
```

而**语句项**是什么呢?

```
//是一些case语句：
//如下：
case 整形常量表达式：
     语句；
```

2.2.1 在switch语句中的 break

在switch语句中，我们没办法直接实现分支，搭配break使用才能实现真正的分支。

比如：

```
#include <stdio.h>
int main()
{
    int day = 0;
    switch(day)
    {
        case 1:
            printf("星期一\n");
            break;
        case 2:
            printf("星期二\n");
            break;
        case 3:
            printf("星期三\n");
            break;
        case 4:
            printf("星期四\n");
            break;
        case 5:
            printf("星期五\n");
            break;
        case 6:
            printf("星期六\n");
            break;
        case 7:
            printf("星期天\n");
            break;
    }
    return 0;
}
```

有时候我们的需求变了：

> 1. 输入1-5，输出的是“weekday”;
> 2. 输入6-7，输出“weekend”

所以我们的代码就应该这样实现了：

```
#include <stdio.h>
//switch代码演示
int main()
{
    int day = 0;
    switch(day)
    {
        case 1:
```

```
        case 2:
        case 3:
        case 4:
        case 5:
            printf("weekday\n");
            break;
        case 6:
        case 7:
            printf("weekend\n");
            break;
    }
    return 0;
}
```

break语句 的实际效果是把语句列表划分为不同的分支部分。

编程好习惯

> 在最后一个 case 语句的后面加上一条 break语句。
>
> （之所以这么写是可以避免出现在以前的最后一个 case 语句后面忘了添加 break语句）。

2.2.2 default子句

如果表达的值与所有的case标签的值都不匹配怎么办?

其实也没什么，结构就是所有的语句都被跳过而已。

程序并不会终止，也不会报错，因为这种情况在C中并不认为是个错误。

但是，如果你并不想忽略不匹配所有标签的表达式的值时该怎么办呢?

你可以在语句列表中增加一条default子句，把下面的标签

`default:`

写在任何一个`case`标签可以出现的位置。

当`switch`表达式的值并不匹配所有`case`标签的值时，这个`default`子句后面的语句就会执行。

所以，每个switch语句中只能出现一条default子句。

但是它可以出现在语句列表的任何位置，而且语句流会像执行一个case标签一样执行default子句。

编程好习惯

> 在每个 switch 语句中都放一条default子句是个好习惯，甚至可以在后边再加一个 break 。

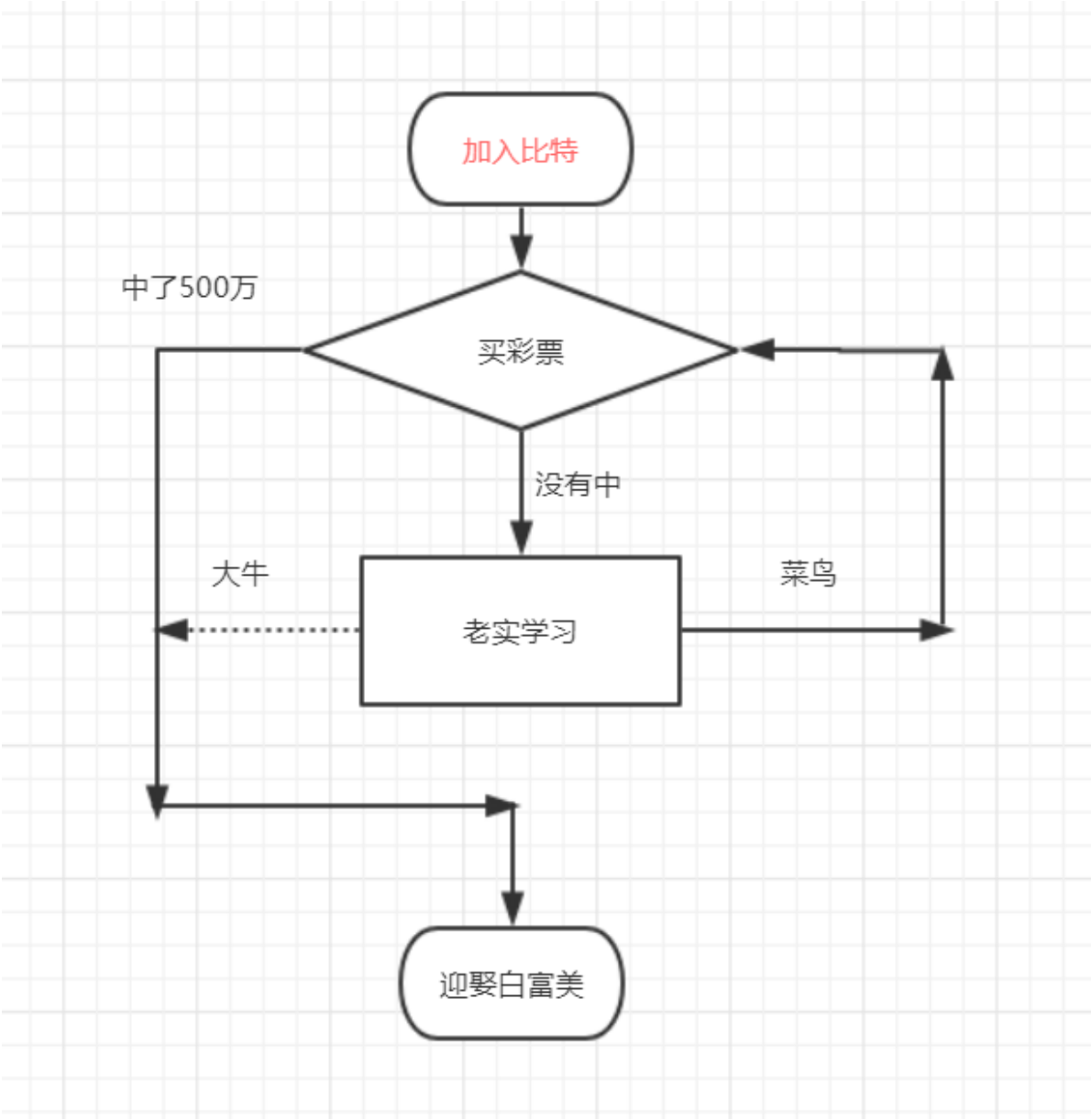
2.2.3 练习

```
#include <stdio.h>

int main()
{
    int n = 1;
```

```
    int m = 2;
    switch (n)
    {
    case 1:
            m++;
    case 2:
            n++;
    case 3:
            switch (n)
            {//switch允许嵌套使用
             case 1:
                    n++;
             case 2:
                    m++;
                    n++;
                    break;
            }
    case 4:
            m++;
            break;
    default:
            break;
    }
    printf("m = %d, n = %d\n", m, n);
    return 0;
}
```

3. 循环语句

- while
- for
- do while



3.1 while循环

我们已经掌握了，if语句：

```
if(条件)
      语句;
```

当条件满足的情况下，if语句后的语句执行，否则不执行。

但是这个语句只会执行一次。

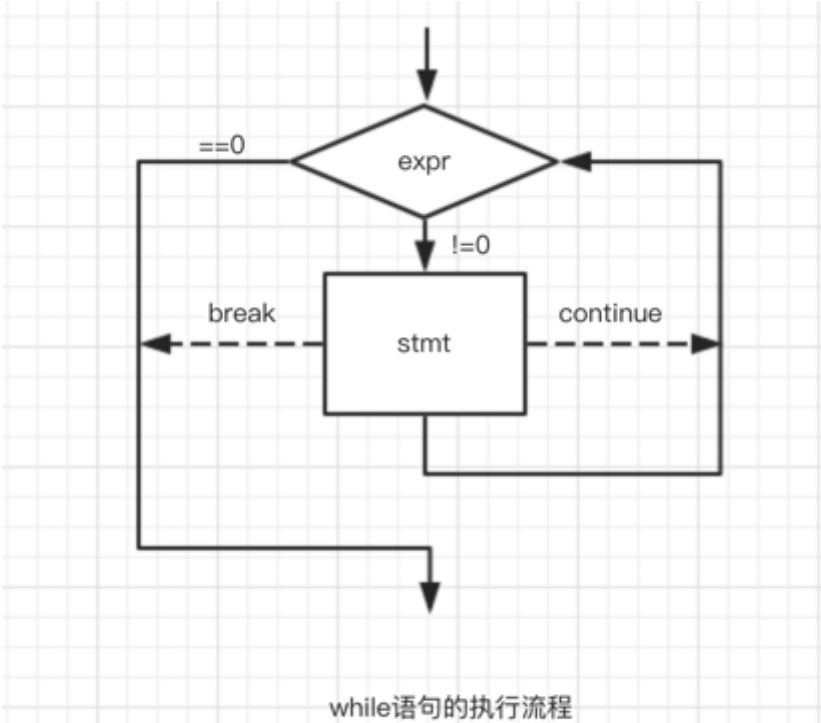
由于我们发现生活中很多的实际的例子是：同一件事情我们需要完成很多次。

那我们怎么做呢？

C语言中给我们引入了：`while`语句，可以实现循环。

```
//while 语法结构

while(表达式)
     循环语句;
```

while语句执行的流程：



while语句的执行流程

比如我们实现：

> 在屏幕上打印1-10的数字。

```
#include <stdio.h>

int main()
{
    int i = 1;
    while(i<=10)
    {
        printf("%d ", i);
        i = i+1;
    }
    return 0;
}
```

上面的代码已经帮我了解了`while`语句的基本语法，那我们再了解一下：

3.1.1 while语句中的break和continue

break介绍

```
//break 代码实例
#include <stdio.h>
int main()
{
    int i = 1;
    while(i<=10)
    {
        if(i == 5)
            break;
```

```
        printf("%d ", i);
        i = i+1;
    }
    return 0;
}
```

这里代码输出的结果是什么?

> 1 2 3 4
>
> 1 2 3 4 5
>
> 1 2 3 4 5 6 7 8 9 10
>
> 1 2 3 4 6 7 8 9 10

总结:
break在while循环中的作用:

> 其实在循环中只要遇到break，就停止后期的所有的循环，直接终止循环。
>
> 所以：while中的break是用于**永久**终止循环的。

continue介绍

```
//continue 代码实例1
#include <stdio.h>
int main()
{
    int i = 1;
    while(i<=10)
    {
        if(i == 5)
            continue;
        printf("%d ", i);
        i = i+1;
    }
    return 0;
}
```

这里代码输出的结果是什么?

> 1 2 3 4
>
> 1 2 3 4 5
>
> 1 2 3 4 5 6 7 8 9 10
>
> 1 2 3 4 6 7 8 9 10

```
//continue 代码实例2
#include <stdio.h>
int main()
{
    int i = 1;
    while(i<=10)
    {
        i = i+1;
```

```
        if(i == 5)
            continue;
        printf("%d ", i);
    }
    return 0;
}
```

这里代码输出的结果是什么?

> 1 2 3 4
>
> 1 2 3 4 5
>
> 1 2 3 4 5 6 7 8 9 10
>
> 1 2 3 4 6 7 8 9 10
>
> 2 3 4 6 7 8 9 10

总结:
continue在while循环中的作用就是:

> continue是用于终止本次循环的，也就是本次循环中continue后边的代码不会再执行，
>
> 而是直接跳转到while语句的判断部分。进行下一次循环的入口判断。

再看几个代码:

```
//代码什么意思?
//代码1
#include <stdio.h>
int main()
{
    int ch = 0;
    while ((ch = getchar()) != EOF)
       putchar(ch);
    return 0;
}

这里的代码适当的修改是可以用来清理缓冲区的.

//代码2
#include <stdio.h>
int main()
{
    char ch = '\0';
    while ((ch = getchar()) != EOF)
    {
         if (ch < '0' || ch > '9')
            continue;
         putchar(ch);
    }
    return 0;
}

//这个代码的作用是：只打印数字字符，跳过其他字符的、
```

3.2 for循环

我们已经知道了while循环，但是我们为什么还要一个for循环呢?

首先来看看for循环的语法:

3.2.1 语法

```
for(表达式1; 表达式2; 表达式3)
    循环语句;
```

表达式1
表达式1为**初始化部分**，用于初始化循环变量的。

表达式2
表达式2为**条件判断部分**，用于判断循环时候终止。

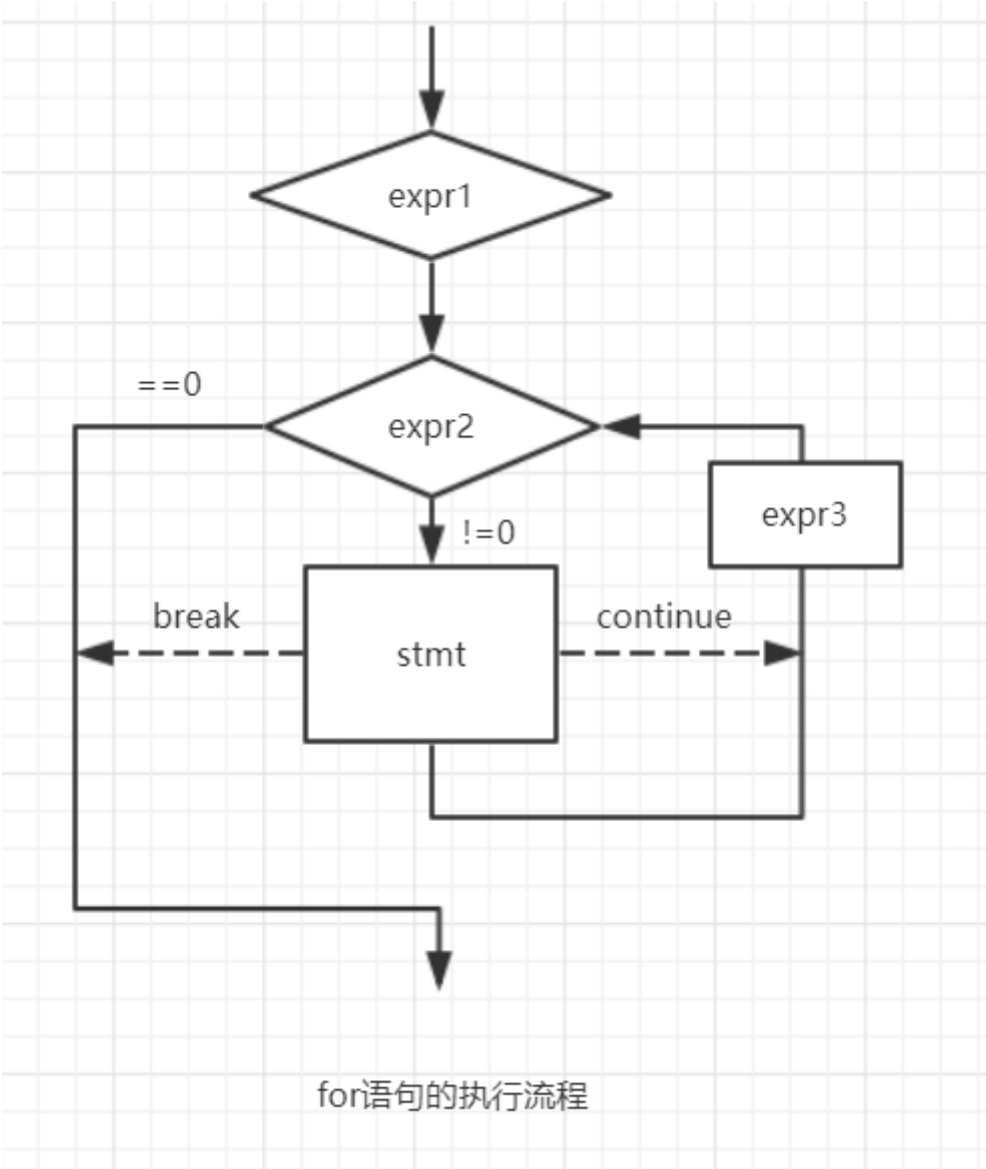
表达式3
表达式3为**调整部分**，用于循环条件的调整。

实际的问题:

> 使用for循环 在屏幕上打印1-10的数字。

```c
#include <stdio.h>

int main()
{
    int i = 0;
    //for(i=1/*初始化*/; i<=10/*判断部分*/; i++/*调整部分*/)
    for(i=1; i<=10; i++)
    {
        printf("%d ", i);
    }
    return 0;
}
```

for循环的执行流程图：



for语句的执行流程

现在我们对比一下for循环和while循环。

```c
int i = 0;
//实现相同的功能，使用while
i=1;//初始化部分
while(i<=10)//判断部分
{
    printf("hehe\n");
    i = i+1;//调整部分
}

//实现相同的功能，使用while
for(i=1; i<=10; i++)
{
    printf("hehe\n");
}
```

可以发现在while循环中依然存在循环的三个必须条件，但是由于风格的问题使得三个部分很可能偏离较远，这样

查找修改就不够集中和方便。所以，for循环的风格更胜一筹；for循环使用的频率也最高。

3.2.2 break和continue在for循环中

我们发现在for循环中也可以出现break和continue，他们的意义和在while循环中是一样的。

但是还是有些差异：

```
//代码1
#include <stdio.h>
int main()
{
    int i = 0;
    for(i=1; i<=10; i++)
    {
        if(i == 5)
            break;
        printf("%d ",i);
    }
    return 0;
}

//代码2
#include <stdio.h>
int main()
{
    int i = 0;
    for(i=1; i<=10; i++)
    {
        if(i == 5)
            continue;
        printf("%d ",i);
    }
    return 0;
}
```

3.2.3 for语句的循环控制变量

建议：

> 1. 不可在for 循环体内修改循环变量，防止 for 循环失去控制。
> 2. 建议for语句的循环控制变量的取值采用“前闭后开区间”写法。

```
int i = 0;
//前闭后开的写法
for(i=0; i<10; i++)
{}

//两边都是闭区间
for(i=0; i<=9; i++)
{}
```

3.2.4 一些for循环的变种

```
#include <stdio.h>
int main()
{
    //代码1
    for(;;)
    {
        printf("hehe\n");
    }
    //for循环中的初始化部分，判断部分，调整部分是可以省略的，但是不建议初学时省略，容易导致问
题。

    //代码2
    int i = 0;
    int j = 0;
    //这里打印多少个hehe?
    for(i=0; i<10; i++)
    {
        for(j=0; j<10; j++)
        {
            printf("hehe\n");
        }
    }

    //代码3
    int i = 0;
    int j = 0;
    //如果省略掉初始化部分，这里打印多少个hehe?
    for(; i<10; i++)
    {
        for(; j<10; j++)
        {
            printf("hehe\n");
        }
    }

    //代码4-使用多余一个变量控制循环
    int x, y;
    for (x = 0, y = 0; x<2 && y<5; ++x, y++)
    {
        printf("hehe\n");
    }
    return 0;
}
```

3.2.5 一道笔试题：

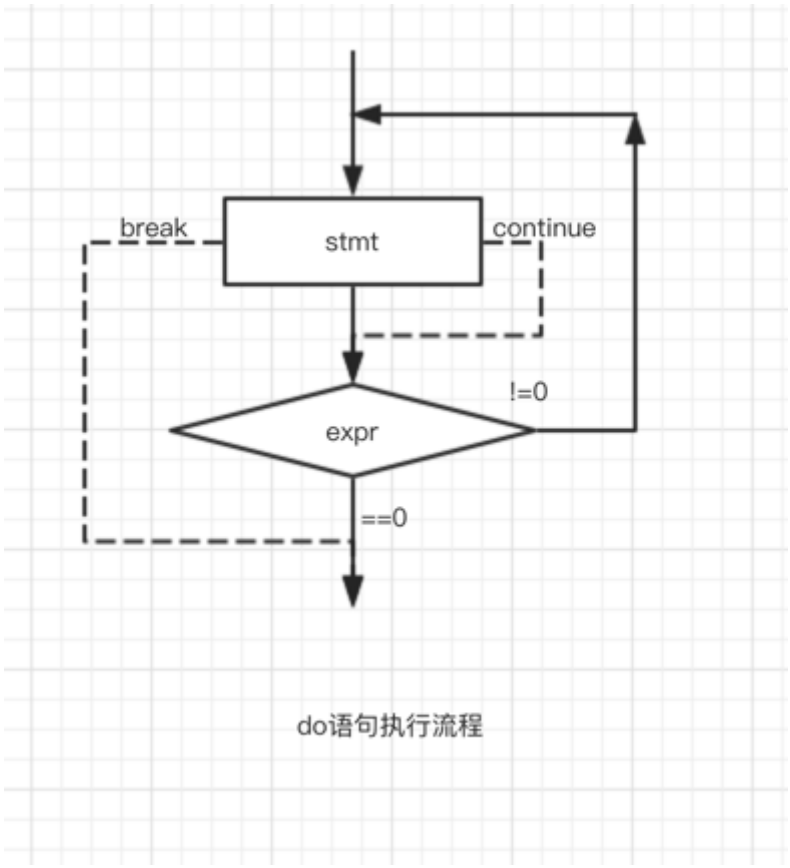
```c
//请问循环要循环多少次?
#include <stdio.h>
int main()
{
    int i = 0;
    int k = 0;
    for(i =0,k=0; k=0; i++,k++)
        k++;
    return 0;
}
```

3.3 do...while()循环

3.3.1 do语句的语法:

```c
do
    循环语句;
while(表达式);
```

3.3.2 执行流程



do语句执行流程

3.3.3 do语句的特点

循环至少执行一次，使用的场景有限，所以不是经常使用。

```c
#include <stdio.h>
int main()
{
    int i = 10;
    do
    {
        printf("%d\n", i);
    }while(i<10);
    return 0;
}
```

3.3.4 do while循环中的break和continue

```c
#include <stdio.h>
int main()
{
    int i = 10;

    do
    {
        if(5 == i)
            break;
        printf("%d\n", i);
    }while(i<10);

    return 0;
}

#include <stdio.h>
int main()
{
    int i = 10;

    do
    {
        if(5 == i)
            continue;
        printf("%d\n", i);
    }while(i<10);

    return 0;
}
```

3.4 练习

1. 计算 n的阶乘。
2. 计算 `1!+2!+3!+……+10!`
3. 在一个有序数组中查找具体的某个数字n。（讲解二分查找）
4. 编写代码，演示多个字符从两端移动，向中间汇聚。
5. 编写代码实现，模拟用户登录情景，并且只能登录三次。（只允许输入三次密码，如果密码正确则提示登录成，如果三次均输入错误，则退出程序。

3.4.1 练习参考代码：

```
//代码1
//编写代码，演示多个字符从两端移动，向中间汇聚
#include <stdio.h>
int main()
{
    char arr1[] = "welcome to bit...";
    char arr2[] = "##################";
    int left = 0;
    int right = strlen(arr1)-1;
    printf("%s\n", arr2);
    //while循环实现
    while(left<=right)
    {
        Sleep(1000);
        arr2[left] = arr1[left];
        arr2[right] = arr1[right];
        left++;
        right--;
        printf("%s\n", arr2);
    }
    //for循环实现
    for (left=0, right=strlen(src)-1;
         left <= right;
         left++, right--)
    {
        Sleep(1000);
        arr2[left] = arr1[left];
        arr2[right] = arr1[right];
        printf( "%s\n", target);
    }
    retutn 0;
}

//代码2

int main()
{
    char psw[10] = "" ;
    int i = 0;
    int j = 0;
    for (i = 0; i < 3 ; ++i)
    {
        printf( "please input:");
        scanf("%s", psw);
        if (strcmp(psw, "password" ) == 0)
            break;
    }
    if (i == 3)
        printf("exit\n");
    else
        printf( "log in\n");
}
```

3.4.2 折半查找算法

比如我买了一双鞋，你好奇问我多少钱，我说不超过300元。你还是好奇，你想知道到底多少，我就让你猜，你会怎么猜?

答案：你每次猜中间数。

代码实现：

实现在主函数内：

```
int main()
{
    int arr[] = {1,2,3,4,5,6,7,8,9,10};
    int left = 0;
    int right = sizeof(arr)/sizeof(arr[0])-1;
    int key = 7;
    int mid = 0;
    while(left<=right)
    {
        mid = (left+right)/2;
        if(arr[mid]>key)
        {
            right = mid-1;
        }
        else if(arr[mid] < key)
        {
            left = mid+1;
        }
        else
            break;
    }
    if(left <= right)
        printf("找到了,下标是%d\n", mid);
    else
        printf("找不到\n");
}
```

如果实现一个二分查找函数：

```
int bin_search(int arr[], int left, int right, int key)
{
    int mid = 0;
    while(left<=right)
    {
        mid = (left+right)>>1;
        if(arr[mid]>key)
        {
            right = mid-1;
        }
        else if(arr[mid] < key)
        {
            left = mid+1;
        }
        else
            return mid;//找到了，返回下标
    }
```

```
    return -1;//找不到
}
```

3.4.3 猜数字游戏实现

参考代码：

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

void menu()
{
    printf("*********************************\n");
    printf("***********  1.play     **********\n");
    printf("***********  0.exit     **********\n");
    printf("*********************************\n");
}

//RAND_MAX--rand函数能返回随机数的最大值。
void game()
{
    int random_num = rand()%100+1;
    int input = 0;
    while(1)
    {
        printf("请输入猜的数字>:");
        scanf("%d", &input);
        if(input > random_num)
        {
            printf("猜大了\n");
        }
        else if(input < random_num)
        {
            printf("猜小了\n");
        }
        else
        {
            printf("恭喜你，猜对了\n");
            break;
        }
    }
}
int main()
{
    int input = 0;
    srand((unsigned)time(NULL));
    do
    {
        menu();
        printf("请选择>:");
        scanf("%d", &input);
        switch(input)
        {
        case 1:
            game();
            break;
```

```
        case 0:
            break;
        default:
            printf("选择错误,请重新输入!\n");
            break;
        }
    }while(input);
    return 0;
}
```

4. goto语句

C语言中提供了可以随意滥用的 goto语句和标记跳转的标号。

从理论上 goto语句是没有必要的，实践中没有goto语句也可以很容易的写出代码。

但是某些场合下goto语句还是用得着的，最常见的用法就是终止程序在某些深度嵌套的结构的处理过程。

例如：一次跳出两层或多层循环。

多层循环这种情况使用break是达不到目的的。它只能从最内层循环退出到上一层的循环。

goto语言真正适合的场景如下：

```
for(...)
    for(...)
    {
        for(...)
        {
            if(disaster)
                goto error;
        }
    }
    …
error:
    if(disaster)
         // 处理错误情况
```

下面是使用goto语句的一个例子，然后使用循环的实现方式替换goto语句：

一个关机程序

```
#include <stdio.h>
int main()
{
    char input[10] = {0};
    system("shutdown -s -t 60");
again:
    printf("电脑将在1分钟内关机，如果输入：我是猪，就取消关机!\n请输入:>");
    scanf("%s", input);
```

```
    if(0 == strcmp(input, "我是猪"))
    {
        system("shutdown -a");
    }
    else
    {
        goto again;
    }
    return 0;
}
```

而如果不适用goto语句，则可以使用循环：

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

int main()
{
    char input[10] = {0};
    system("shutdown -s -t 60");
    while(1)
    {
        printf("电脑将在1分钟内关机，如果输入：我是猪，就取消关机!\n请输入:>");
        scanf("%s", input);
        if(0 == strcmp(input, "我是猪"))
        {
            system("shutdown -a");
            break;
        }
    }
    return 0;
}
```

关于shutdown命令的扩展-（请点这里）

本章完。

BIT-3-函数

1. 函数是什么
2. 库函数
3. 自定义函数
4. 函数参数
5. 函数调用
6. 函数的嵌套调用和链式访问
7. 函数的声明和定义
8. 函数递归

正文开始@比特就业课

1. 函数是什么?

数学中我们常见到函数的概念。但是你了解C语言中的函数吗?
维基百科中对函数的定义：子程序

> - 在计算机科学中，子程序（英语：Subroutine, procedure, function, routine, method, subprogram, callable unit），是一个大型程序中的某部分代码， 由一个或多个语句块组成。它负责完成某项特定任务，而且相较于其他代 码，具备相对的独立性。
> - 一般会有输入参数并有返回值，提供对过程的封装和细节的隐藏。这些代码通常被集成为软件库。

2. C语言中函数的分类：

1. 库函数
2. 自定义函数

2.1 库函数：

为什么会有库函数?

1. 我们知道在我们学习C语言编程的时候，总是在一个代码编写完成之后迫不及待的想知道结果，想把这个结果打印到我们的屏幕上看看。这个时候我们会频繁的使用一个功能：将信息按照一定的格式打印到屏幕上（printf）。
2. 在编程的过程中我们会频繁的做一些字符串的拷贝工作（strcpy）。
3. 在编程是我们也计算，总是会计算n的k次方这样的运算（pow）。

像上面我们描述的基础功能，它们不是业务性的代码。我们在开发的过程中每个程序员都可能用的到，为了支持可移植性和提高程序的效率，所以C语言的基础库中提供了一系列类似的库函数，方便程序员进行软件开发。

那怎么学习库函数呢?

这里我们简单的看看：www.cplusplus.com

C Library

The elements of the C language library are also included as a subset of the C++ Standard library. These cover many aspects, from general utility functions and macros to input/output functions and dynamic memory management functions:

<cassert> (assert.h)	C Diagnostics Library (header)
<cctype> (ctype.h)	Character handling functions (header)
<cerrno> (errno.h)	C Errors (header)
<cfenv> (fenv.h)	Floating-point environment (header)
<cfloat> (float.h)	Characteristics of floating-point types (header)
<cinttypes> (inttypes.h)	C integer types (header)
<ciso646> (iso646.h)	ISO 646 Alternative operator spellings (header)
<climits> (limits.h)	Sizes of integral types (header)
<clocale> (locale.h)	C localization library (header)
<cmath> (math.h)	C numerics library (header)
<csetjmp> (setjmp.h)	Non local jumps (header)
<csignal> (signal.h)	C library to handle signals (header)
<cstdarg> (stdarg.h)	Variable arguments handling (header)
<cstdbool> (stdbool.h)	Boolean type (header)
<cstddef> (stddef.h)	C Standard definitions (header)
<cstdint> (stdint.h)	Integer types (header)
<cstdio> (stdio.h)	C library to perform Input/Output operations (header)
<cstdlib> (stdlib.h)	C Standard General Utilities Library (header)
<cstring> (string.h)	C Strings (header)
<ctgmath> (tgmath.h)	Type-generic math (header)
<ctime> (time.h)	C Time Library (header)
<cuchar> (uchar.h)	Unicode characters (header)
<cwchar> (wchar.h)	Wide characters (header)
<cwctype> (wctype.h)	Wide character type (header)

简单的总结，C语言常用的库函数都有：

- IO函数
- 字符串操作函数
- 字符操作函数
- 内存操作函数
- 时间/日期函数
- 数学函数
- 其他库函数

我们参照文档，学习几个库函数：(教会学生怎么使用文档来学习库函数)。

strcpy

```
char * strcpy ( char * destination, const char * source );
```

memset

```
void * memset ( void * ptr, int value, size_t num );
```

注：
但是库函数必须知道的一个秘密就是：使用库函数，必须包含`#include`对应的头文件。

这里对照文档来学习上面几个库函数，目的是掌握库函数的使用方法。

2.1.1 如何学会使用库函数?

需要全部记住吗? **No**
需要学会查询工具的使用:

> MSDN(Microsoft Developer Network)
>
> www.cplusplus.com
>
> http://en.cppreference.com（英文版）
>
> http://zh.cppreference.com（中文版）

英文很重要。最起码得看懂文献。

2.2 自定义函数

如果库函数能干所有的事情，那还要程序员干什么?

所有更加重要的是**自定义函数。**

自定义函数和库函数一样，有函数名，返回值类型和函数参数。

但是不一样的是这些都是我们自己来设计。这给程序员一个很大的发挥空间。

函数的组成:

```
ret_type fun_name(para1, * )
{
    statement;//语句项
}

ret_type 返回类型
fun_name 函数名
para1    函数参数
```

我们举一个例子:

> 写一个函数可以找出两个整数中的最大值。

```
#include <stdio.h>

//get_max函数的设计
int get_max(int x, int y)
{
    return (x>y)?(x):(y);
}

int main()
{
    int num1 = 10;
    int num2 = 20;
    int max = get_max(num1, num2);
    printf("max = %d\n", max);
```

```
    return 0;
}
```

再举个例子：

> 写一个函数可以交换两个整形变量的内容。

```
#include <stdio.h>
//实现成函数，但是不能完成任务
void Swap1(int x, int y)
{
    int tmp = 0;
    tmp = x;
    x = y;
    y = tmp;
}

//正确的版本
void Swap2(int *px, int *py)
{
    int tmp = 0;
    tmp = *px;
    *px = *py;
    *py = tmp;
}
int main()
{
    int num1 = 1;
    int num2 = 2;
    Swap1(num1, num2);
    printf("Swap1::num1 = %d num2 = %d\n", num1, num2);
    Swap2(&num1, &num2);
    printf("Swap2::num1 = %d num2 = %d\n", num1, num2);
    return 0;
}
```

3. 函数的参数

3.1 实际参数（实参）：

> 真实传给函数的参数，叫实参。
>
> 实参可以是：常量、变量、表达式、函数等。
>
> 无论实参是何种类型的量，在进行函数调用时，它们都必须有确定的值，以便把这些值传送给形参。

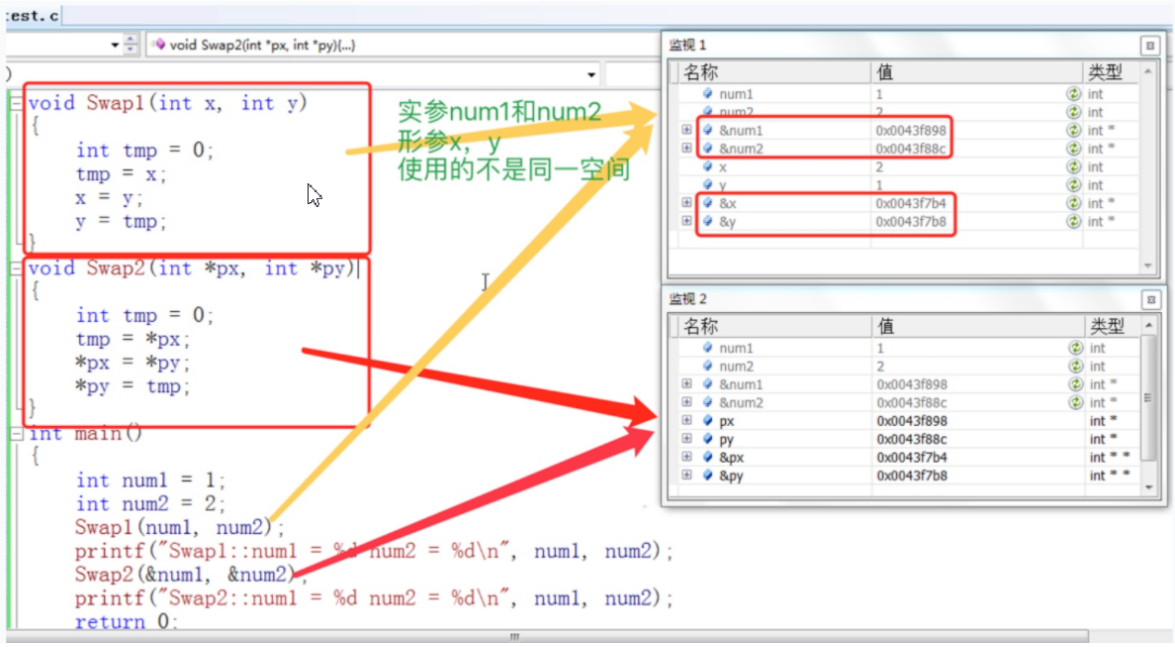
3.2 形式参数（形参）：

> 形式参数是指函数名后括号中的变量，因为形式参数只有在函数被调用的过程中才实例化（分配内存单
>
> 元），所以叫形式参数。形式参数当函数调用完成之后就自动销毁了。因此形式参数只在函数中有效。

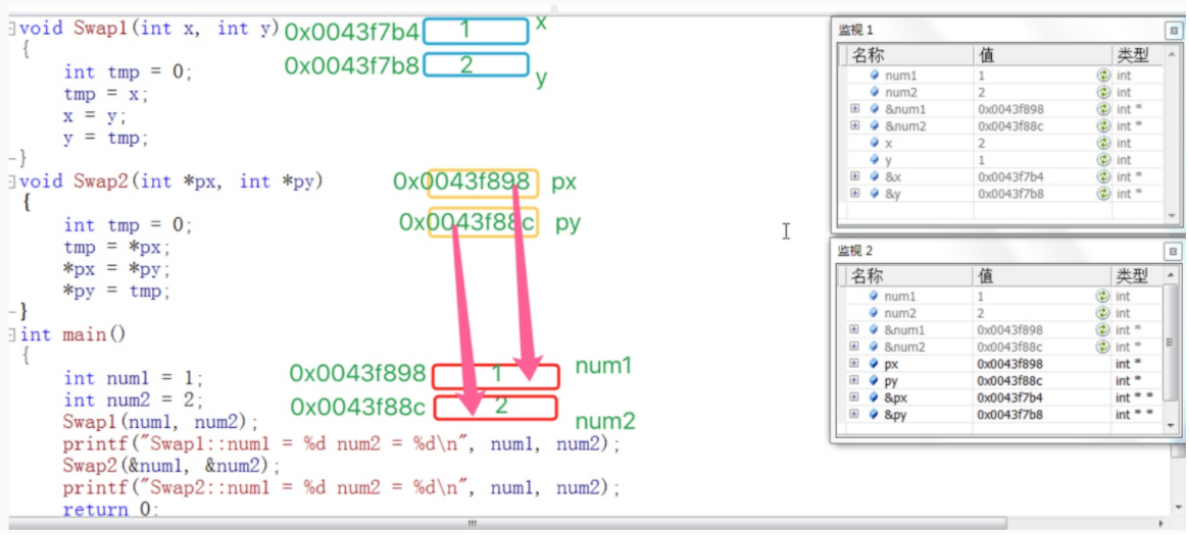
上面`Swap1`和`Swap2`函数中的参数`x，y，px，py`都是**形式参数**。在main函数中传给`Swap1`的`num1`，`num2`和传

给`Swap2`函数的`&num1`，`&num2`是**实际参数**。

这里我们对函数的实参和形参进行分析：



代码对应的内存分配如下：



这里可以看到`Swap1`函数在调用的时候，`x`，`y`拥有自己的空间，同时拥有了和实参一模一样的内容。

所以我们可以简单的认为：**形参实例化之后其实相当于实参的一份临时拷贝。**

4. 函数的调用：

4.1 传值调用

> 函数的形参和实参分别占有不同内存块，对形参的修改不会影响实参。

4.2 传址调用

> - 传址调用是把函数外部创建变量的内存地址传递给函数参数的一种调用函数的方式。
> - 这种传参方式可以让函数和函数外边的变量建立起真正的联系，也就是函数内部可以直接操作函数外部的变量。

4.3 练习

> 1. 写一个函数可以判断一个数是不是素数。
> 2. 写一个函数判断一年是不是闰年。
> 3. 写一个函数，实现一个整形有序数组的二分查找。
> 4. 写一个函数，每调用一次这个函数，就会将`num`的值增加1。

```
int main()
{
    int num = 0;
    //调用函数，使得num每次增加1
    return 0;
}
```

5. 函数的嵌套调用和链式访问

函数和函数之间可以根据实际的需求进行组合的，也就是互相调用的。

5.1 嵌套调用

```
#include <stdio.h>
void new_line()
{
    printf("hehe\n");
}
void three_line()
{
    int i = 0;
    for(i=0; i<3; i++)
    {
        new_line();
    }
}
int main()
{
    three_line();
    return 0;
}
```

函数可以嵌套调用，但是不能嵌套定义。

5.2 链式访问

> 把一个函数的返回值作为另外一个函数的参数。

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

int main()
{
    char arr[20] = "hello";
    int ret = strlen(strcat(arr,"bit"));//这里介绍一下strlen函数
    printf("%d\n", ret);
    return 0;
}
```

```
#include <stdio.h>
int main()
{
    printf("%d", printf("%d", printf("%d", 43)));
    //结果是啥？
    //注：printf函数的返回值是打印在屏幕上字符的个数
    return 0;
}
```

6. 函数的声明和定义

6.1 函数声明：

> 1. 告诉编译器有一个函数叫什么，参数是什么，返回类型是什么。但是具体是不是存在，函数声明决定不了。
> 2. 函数的声明一般出现在函数的使用之前。要满足**先声明后使用**。
> 3. 函数的声明一般要放在头文件中的。

6.2 函数定义：

> 函数的定义是指函数的具体实现，交待函数的功能实现。

test.h的内容
放置函数的声明

```
#ifndef __TEST_H__
#define __TEST_H__
//函数的声明
int Add(int x, int y);

#endif //__TEST_H__
```

test.c的内容
放置函数的实现

```c
#include "test.h"
//函数Add的实现
int Add(int x, int y)
{
    return x+y;
}
```

这种分文件的书写形式，在三字棋和扫雷的时候，再教学生分模块来写。

7. 函数递归

7.1 什么是递归?

> 程序调用自身的编程技巧称为递归（ recursion）。
>
> 递归做为一种算法在程序设计语言中广泛应用。 一个过程或函数在其定义或说明中有直接或间接调用自身的
>
> 一种方法，它通常把一个大型复杂的问题层层转化为一个与原问题相似的规模较小的问题来求解，递归策略
>
> 只需少量的程序就可描述出解题过程所需要的多次重复计算，大大地减少了程序的代码量。
>
> **递归的主要思考方式在于：把大事化小**

7.2 递归的两个必要条件

- 存在限制条件，当满足这个限制条件的时候，递归便不再继续。
- 每次递归调用之后越来越接近这个限制条件。

7.2.1 练习1：（画图讲解）

> 接受一个整型值（无符号），按照顺序打印它的每一位。
> 例如：
> 输入：1234，输出 1 2 3 4.

参考代码：

```c
#include <stdio.h>
void print(int n)
{
    if(n>9)
    {
        print(n/10);
    }
    printf("%d ", n%10);
}
int main()
{
    int num = 1234;
    print(num);
    return 0;
}
```

7.2.2 练习2：（画图讲解）

> 编写函数不允许创建临时变量，求字符串的长度。

参考代码：

```
#incude <stdio.h>
int Strlen(const char*str)
{
    if(*str == '\0')
        return 0;
    else
        return 1+Strlen(str+1);
}
int main()
{
    char *p = "abcdef";
    int len = Strlen(p);
    printf("%d\n", len);
    return 0;
}
```

7.3 递归与迭代

7.3.1 练习3：

> 求n的阶乘。（不考虑溢出）

参考代码：

```
int factorial(int n)
{
    if(n <= 1)
        return 1;
    else
        return n * factorial(n-1);
}
```

7.3.2 练习4：

> 求第n个斐波那契数。（不考虑溢出）

参考代码：

```
int fib(int n)
{
    if (n <= 2)
        return 1;
    else
        return fib(n - 1) + fib(n - 2);
}
```

但是我们发现**有问题**；

- 在使用`fib`这个函数的时候如果我们要计算第50个斐波那契数字的时候特别耗费时间。
- 使用`factorial`函数求10000的阶乘（不考虑结果的正确性），程序会崩溃。

为什么呢?

- 我们发现`fib`函数在调用的过程中很多计算其实在一直重复。
  如果我们把代码修改一下:

```
int count = 0;//全局变量
int fib(int n)
{
    if(n == 3)
        count++;
    if (n <= 2)
        return 1;
    else
        return fib(n - 1) + fib(n - 2);
}
```

最后我们输出看看count，是一个很大很大的值。

那我们如何改进呢?

- 在调试`factorial`函数的时候，如果你的参数比较大，那就会报错：`stack overflow（栈溢出）`这样的信息。
  系统分配给程序的栈空间是有限的，但是如果出现了死循环，或者（死递归），这样有可能导致一直开辟栈空间，最终产生栈空间耗尽的情况，这样的现象我们称为栈溢出。

那如何解决上述的问题:

1. 将递归改写成非递归。
2. 使用static对象替代`nonstatic`局部对象。在递归函数设计中，可以使用`static`对象替代`nonstatic`局部对象（即栈对象），这不

   仅可以减少每次递归调用和返回时产生和释放`nonstatic`对象的开销，而且`static`对象还可以保存递归调用的中间状态，并且可为

   各个调用层所访问。

比如，下面代码就采用了，非递归的方式来实现:

```
//求n的阶乘
int factorial(int n)
{
        int result = 1;
        while (n > 1)
        {
             result *= n ;
             n -= 1;
        }
        return result;
}
//求第n个斐波那契数
int fib(int n)
{
     int result;
     int pre_result;
     int next_older_result;
     result = pre_result = 1;
```

```
    while (n > 2)
    {
        n -= 1;
        next_older_result = pre_result;
        pre_result = result;
        result = pre_result + next_older_result;
    }
    return result;
}
```

提示：

1. 许多问题是以递归的形式进行解释的，这只是因为它比非递归的形式更为清晰。
2. 但是这些问题的迭代实现往往比递归实现效率更高，虽然代码的可读性稍微差些。
3. 当一个问题相当复杂，难以用迭代实现时，此时递归实现的简洁性便可以补偿它所带来的运行时开销。

函数递归的几个经典题目（自主研究）：

> 1. 汉诺塔问题
> 2. 青蛙跳台阶问题

本章完。

BIT-4-数组

1. 一维数组的创建和初始化
2. 一维数组的使用
3. 一维数组在内存中的存储
4. 二维数组的创建和初始化
5. 二维数组的使用
6. 二维数组在内存中的存储
7. 数组越界
8. 数组作为函数参数
9. 数组的应用实例1：三子棋
10. 数组的应用实例2：扫雷游戏

正文开始@比特就业课

1. 一维数组的创建和初始化。

1.1 数组的创建

数组是一组相同类型元素的集合。
数组的创建方式：

```
type_t   arr_name   [const_n];
//type_t 是指数组的元素类型
//const_n 是一个常量表达式，用来指定数组的大小
```

数组创建的实例：

```
//代码1
int arr1[10];

//代码2
int count = 10;
int arr2[count];//数组时候可以正常创建？

//代码3
char arr3[10];
float arr4[1];
double arr5[20];
```

注：数组创建，在C99标准之前，`[]`中要给一个**常量**才可以，不能使用变量。在C99标准支持了变长数组的概念。

1.2 数组的初始化

数组的初始化是指，在创建数组的同时给数组的内容一些合理初始值（初始化）。
看代码：

```
int arr1[10] = {1,2,3};
int arr2[] = {1,2,3,4};
int arr3[5] = {1, 2, 3, 4, 5};
char arr4[3] = {'a',98, 'c'};
char arr5[] = {'a','b','c'};
char arr6[] = "abcdef";
```

数组在创建的时候如果想不指定数组的确定的大小就得初始化。数组的元素个数根据初始化的内容来确定。
但是对于下面的代码要区分，内存中如何分配。

```
char arr1[] = "abc";
char arr2[3] = {'a','b','c'};
```

1.3 一维数组的使用

对于数组的使用我们之前介绍了一个操作符：`[]`，下标引用操作符。它其实就数组访问的操作符。
我们来看代码：

```
#include <stdio.h>
int main()
{
    int arr[10] = {0};//数组的不完全初始化
    //计算数组的元素个数
    int sz = sizeof(arr)/sizeof(arr[0]);
    //对数组内容赋值,数组是使用下标来访问的，下标从0开始。所以：
    int i = 0;//做下标
    for(i=0; i<10; i++)//这里写10，好不好？
    {
        arr[i] = i;
    }
    //输出数组的内容
    for(i=0; i<10; ++i)
    {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

总结:

1. 数组是使用下标来访问的，下标是从0开始。
2. 数组的大小可以通过计算得到。

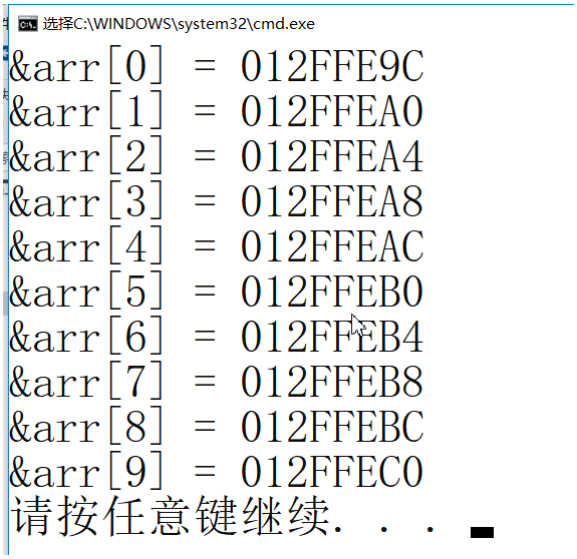
```
int arr[10];
int sz = sizeof(arr)/sizeof(arr[0]);
```

1.4 一维数组在内存中的存储

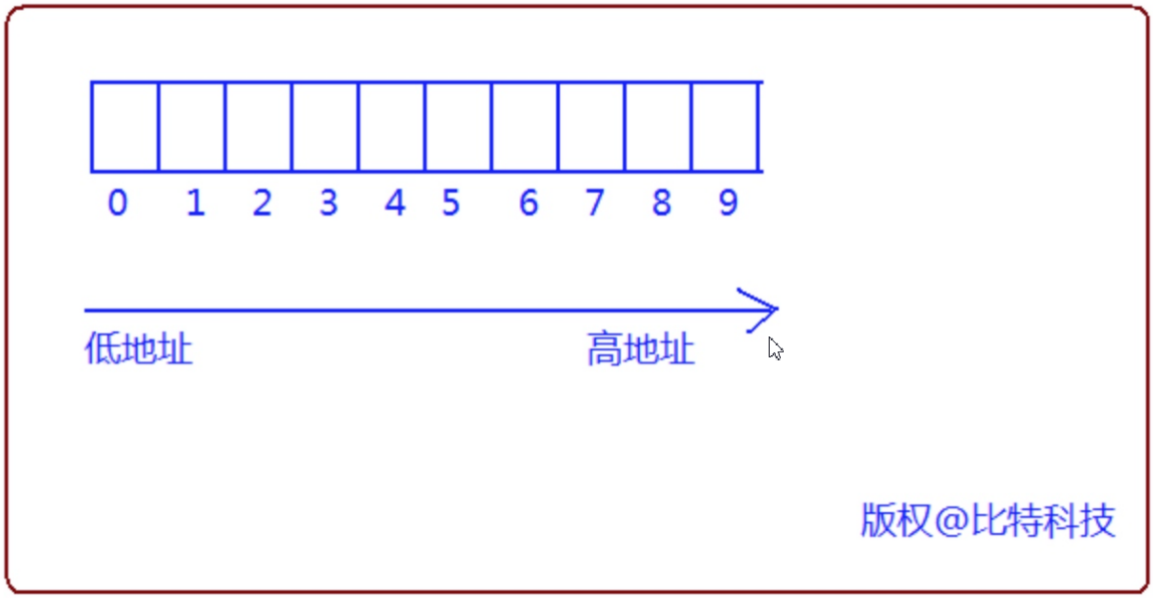
接下来我们探讨数组在内存中的存储。
看代码：

```
#include <stdio.h>

int main()
{
    int arr[10] = {0};
    int i = 0;
    int sz = sizeof(arr)/sizeof(arr[0]);

    for(i=0; i<sz; ++i)
    {
        printf("&arr[%d] = %p\n", i, &arr[i]);
    }
    return 0;
}
```

输出的结果如下：



仔细观察输出的结果，我们知道，随着数组下标的增长，元素的地址，也在有规律的递增。
由此可以得出结论：**数组在内存中是连续存放的。**



2. 二维数组的创建和初始化

2.1 二维数组的创建

```
//数组创建
int arr[3][4];
char arr[3][5];
double arr[2][4];
```

2.2 二维数组的初始化

```
//数组初始化
int arr[3][4] = {1,2,3,4};
int arr[3][4] = {{1,2},{4,5}};
int arr[][4] = {{2,3},{4,5}};//二维数组如果有初始化，行可以省略，列不能省略
```

2.3 二维数组的使用

二维数组的使用也是通过下标的方式。
看代码：

```
#include <stdio.h>
int main()
{
    int arr[3][4] = {0};
    int i = 0;
    for(i=0; i<3; i++)
    {
        int j = 0;
        for(j=0; j<4; j++)
        {
            arr[i][j] = i*4+j;
```

```
        }
    }
    for(i=0; i<3; i++)
    {
        int j = 0;
        for(j=0; j<4; j++)
        {
            printf("%d ", arr[i][j]);
        }
    }
    return 0;
}
```

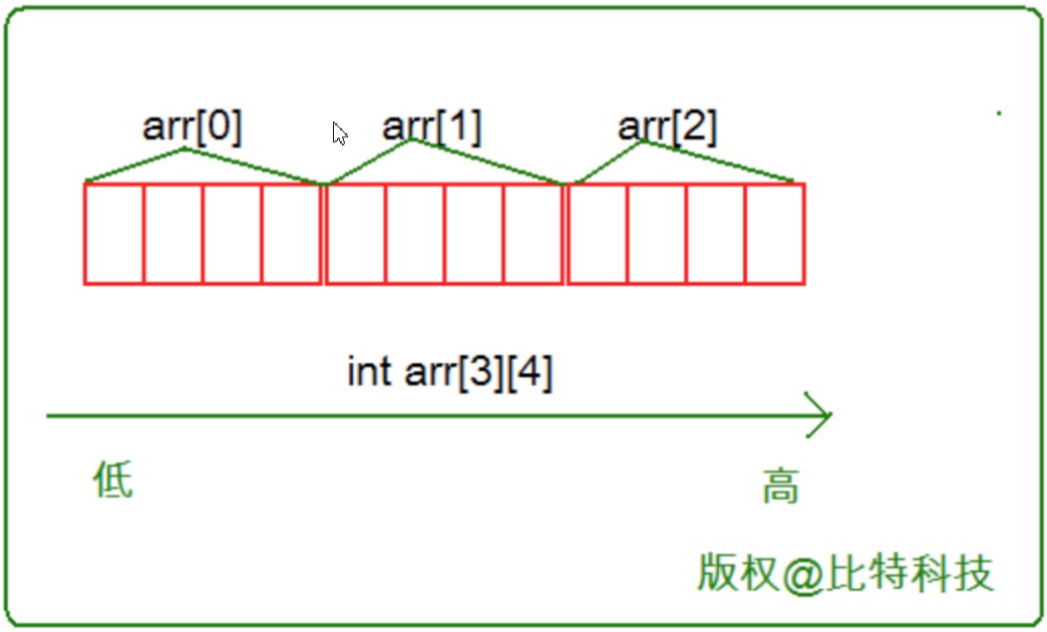
2.4 二维数组在内存中的存储

像一维数组一样，这里我们尝试打印二维数组的每个元素。

```
#include <stdio.h>
int main()
{
    int arr[3][4];
    int i = 0;
    for(i=0; i<3; i++)
    {
        int j = 0;
        for(j=0; j<4; j++)
        {
            printf("&arr[%d][%d] = %p\n", i, j,&arr[i][j]);
        }
    }
    return 0;
}
```

输出的结果是这样的：

通过结果我们可以分析到，其实二维数组在内存中也是连续存储的。



3. 数组越界

数组的下标是有范围限制的。

数组的下规定是从0开始的，如果数组有n个元素，最后一个元素的下标就是n-1。

所以数组的下标如果小于0，或者大于n-1，就是数组越界访问了，超出了数组合法空间的访问。

C语言本身是不做数组下标的越界检查，编译器也不一定报错，但是编译器不报错，并不意味着程序就是正确的，

所以程序员写代码时，最好自己做越界的检查。

```
#include <stdio.h>

int main()
{
    int arr[10] = {1,2,3,4,5,6,7,8,9,10};
    int i = 0;
    for(i=0; i<=10; i++)
    {
        printf("%d\n", arr[i]);//当i等于10的时候，越界访问了
    }
    return 0;
}
```

二维数组的行和列也可能存在越界。

4. 数组作为函数参数

往往我们在写代码的时候，会将数组作为参数传个函数，比如：我要实现一个冒泡排序（这里要讲算法思想）函数

将一个整形数组排序。

那我们将会这样使用该函数：

4.1 冒泡排序函数的错误设计

```
//方法1：
#include <stdio.h>
void bubble_sort(int arr[])
{
    int sz = sizeof(arr)/sizeof(arr[0]);//这样对吗？
    int i = 0;
    for(i=0; i<sz-1; i++)
    {
        int j = 0;
        for(j=0; j<sz-i-1; j++)
        {
            if(arr[j] > arr[j+1])
            {
                int tmp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = tmp;
            }
        }
    }
}

int main()
{
    int arr[] = {3,1,7,5,8,9,0,2,4,6};
    bubble_sort(arr);//是否可以正常排序？
    for(i=0; i<sizeof(arr)/sizeof(arr[0]); i++)
    {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

方法1，出问题，那我们找一下问题，调试之后可以看到`bubble_sort`函数内部的`sz`，是1。

难道数组作为函数参数的时候，不是把整个数组的传递过去？

4.2 数组名是什么？

```
#include <stdio.h>
int main()
{
    int arr[10] = {1,2, 3,4,5};
    printf("%p\n", arr);
    printf("%p\n", &arr[0]);
    printf("%d\n", *arr);
    //输出结果
    return 0;
}
```

结论：

> 数组名是数组首元素的地址。（有两个例外）

如果数组名是首元素地址，那么：

```
int arr[10] = {0};
printf("%d\n", sizeof(arr));
```

为什么输出的结果是：40?

补充：

> 1. sizeof(数组名)，计算整个数组的大小，sizeof内部单独放一个数组名，数组名表示整个数组。
> 2. &数组名，取出的是数组的地址。&数组名，数组名表示整个数组。

除此1,2两种情况之外，所有的数组名都表示数组首元素的地址。

4.3 冒泡排序函数的正确设计

当数组传参的时候，实际上只是把数组的首元素的地址传递过去了。

所以即使在函数参数部分写成数组的形式：`int arr[]`表示的依然是一个指针：`int *arr`。

那么，函数内部的`sizeof(arr)`结果是4。

如果`方法1`错了，该怎么设计?

```
//方法2
void bubble_sort(int arr[], int sz)//参数接收数组元素个数
{
    //代码同上面函数
}
int main()
{
    int arr[] = {3,1,7,5,8,9,0,2,4,6};
    int sz = sizeof(arr)/sizeof(arr[0]);
    bubble_sort(arr, sz);//是否可以正常排序?
    for(i=0; i<sz; i++)
    {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

5. 数据实例：

5.1 数组的应用实例1：三子棋

参考代码，见课堂代码。

5.2 数组的应用实例2：扫雷游戏

参考代码，见课堂代码。

本章完

BIT-5-操作符详解

1. 各种操作符的介绍。
2. 表达式求值

正文开始@比特就业课

1. 操作符分类：

> 算术操作符
>
> 移位操作符
>
> 位操作符
>
> 赋值操作符
>
> 单目操作符
>
> 关系操作符
>
> 逻辑操作符
>
> 条件操作符
>
> 逗号表达式
>
> 下标引用、函数调用和结构成员

2. 算术操作符

```
+   -   *   /   %
```

1. 除了`%`操作符之外，其他的几个操作符可以作用于整数和浮点数。
2. 对于`/`操作符如果两个操作数都为整数，执行整数除法。而只要有浮点数执行的就是浮点数除法。
3. `%`操作符的两个操作数必须为整数。返回的是整除之后的余数。

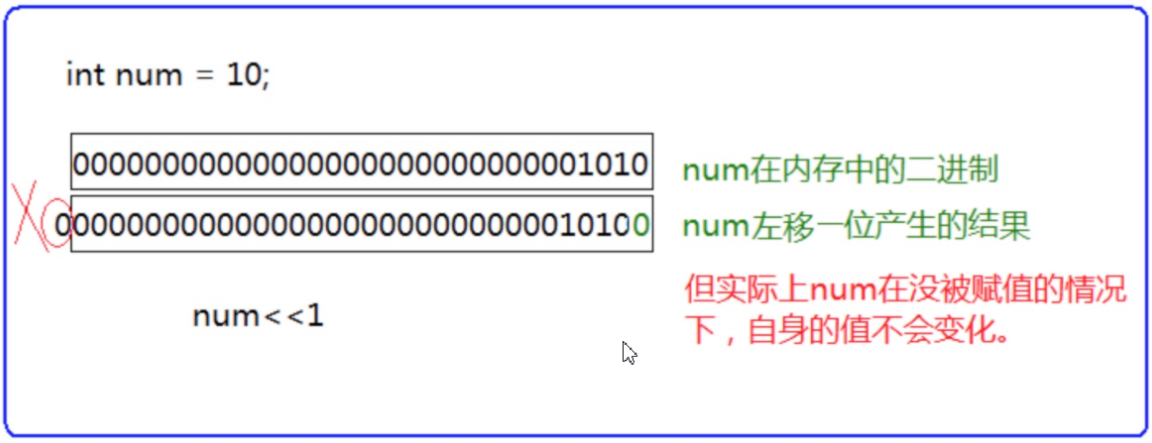
3. 移位操作符

```
<< 左移操作符
>> 右移操作符

注：移位操作符的操作数只能是整数。
```

3.1 左移操作符

移位规则：

> 左边抛弃、右边补0

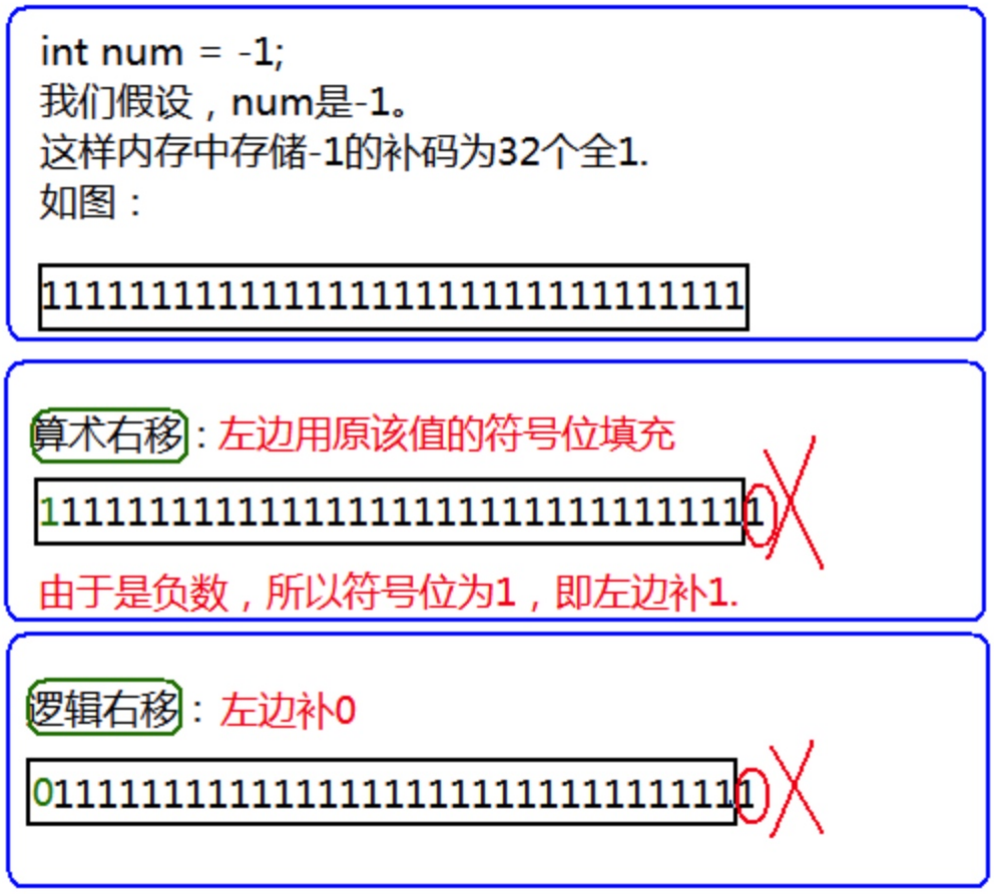


3.2 右移操作符

移位规则：

首先右移运算分两种：

> 1. 逻辑移位
>   左边用0填充，右边丢弃
> 2. 算术移位
>   左边用原该值的符号位填充，右边丢弃



警告⚠：
对于移位运算符，不要移动负数位，这个是标准未定义的。
例如：

```
int num = 10;
num>>-1;//error
```

4. 位操作符

位操作符有：

```
&   //按位与
|   //按位或
^   //按位异或
注：他们的操作数必须是整数。
```

练习一下：

```
#include <stdio.h>
int main()
{
    int num1 = 1;
    int num2 = 2;
    num1 & num2;
    num1 | num2;
    num1 ^ num2;
    return 0;
}
```

一道变态的面试题：

> 不能创建临时变量（第三个变量），实现两个数的交换。

```
#include <stdio.h>
int main()
{
    int a = 10;
    int b = 20;
    a = a^b;
    b = a^b;
    a = a^b;
    printf("a = %d  b = %d\n", a, b);
    return 0;
}
```

练习：

> 编写代码实现：求一个整数存储在内存中的二进制中1的个数。

```
参考代码：
//方法1
#include <stdio.h>
int main()
{
    int num  = 10;
```

```
    int count=  0;//计数
    while(num)
    {
        if(num%2 == 1)
            count++;
        num = num/2;
    }
    printf("二进制中1的个数 = %d\n", count);
    return 0;
}
//思考这样的实现方式有没有问题?

//方法2:
#include <stdio.h>
int main()
{
    int num = -1;
    int i = 0;
    int count = 0;//计数
    for(i=0; i<32; i++)
    {
        if( num & (1 << i) )
            count++;
    }
    printf("二进制中1的个数 = %d\n",count);
    return 0;
}
//思考还能不能更加优化，这里必须循环32次的。

//方法3:
#include <stdio.h>
int main()
{
    int num = -1;
    int i = 0;
    int count = 0;//计数
    while(num)
    {
        count++;
        num = num&(num-1);
    }
    printf("二进制中1的个数 = %d\n",count);
    return 0;
}
//这种方式是不是很好？达到了优化的效果，但是难以想到。
```

5. 赋值操作符

赋值操作符是一个很棒的操作符，他可以让你得到一个你之前不满意的值。也就是你可以给自己重新赋值。

```
int weight = 120;//体重
weight = 89;//不满意就赋值
double salary = 10000.0;
salary = 20000.0;//使用赋值操作符赋值。
```

赋值操作符可以连续使用，比如：

```
int a = 10;
int x = 0;
int y = 20;
a = x = y+1;//连续赋值
```

这样的代码感觉怎么样？

那同样的语义，你看看：

```
x = y+1;
a = x;
```

这样的写法是不是更加清晰爽朗而且易于调试。

复合赋值符

```
+=

-=

*=

/=

%=

>>=

<<=

&=

|=

^=
```

这些运算符都可以写成复合的效果。
比如：

```
int x = 10;
x = x+10;
x += 10;//复合赋值
//其他运算符一样的道理。这样写更加简洁。
```

6. 单目操作符

6.1 单目操作符介绍

```
!           逻辑反操作
-           负值
+           正值
&           取地址
sizeof      操作数的类型长度（以字节为单位）
~           对一个数的二进制按位取反
--          前置、后置--
++          前置、后置++
*           间接访问操作符(解引用操作符)
(类型)      强制类型转换
```

演示代码：

```
#include <stdio.h>
int main()
{
    int a = -10;
    int *p = NULL;
    printf("%d\n", !2);
    printf("%d\n", !0);
    a = -a;
    p = &a;
    printf("%d\n", sizeof(a));
    printf("%d\n", sizeof(int));
    printf("%d\n", sizeof a);//这样写行不行？
    printf("%d\n", sizeof int);//这样写行不行？
    return 0;
}
```

关于sizeof其实我们之前已经见过了，可以求变量（类型）所占空间的大小。

6.2 sizeof 和 数组

```
#include <stdio.h>
void test1(int arr[])
{
    printf("%d\n", sizeof(arr));//(2)
}
void test2(char ch[])
{
    printf("%d\n", sizeof(ch));//(4)
}
int main()
{
    int arr[10] = {0};
    char ch[10] = {0};
    printf("%d\n", sizeof(arr));//(1)
    printf("%d\n", sizeof(ch));//(3)
    test1(arr);
    test2(ch);
    return 0;
}

问：
（1）、（2）两个地方分别输出多少？
（3）、（4）两个地方分别输出多少？
```

```
//++和--运算符

//前置++和--
#include <stdio.h>
int main()
{
    int a = 10;
    int x = ++a;
```

```
    //先对a进行自增，然后对使用a，也就是表达式的值是a自增之后的值。x为11。
    int y = --a;
    //先对a进行自减，然后对使用a，也就是表达式的值是a自减之后的值。y为10;
    return 0;
}

//后置++和--
#include <stdio.h>
int main()
{
    int a = 10;
    int x = a++;
    //先对a先使用，再增加，这样x的值是10；之后a变成11；
    int y = a--;
    //先对a先使用，再自减，这样y的值是11；之后a变成10；
    return 0;
}
```

7. 关系操作符

关系操作符

```
>
>=
<
<=
!=      用于测试“不相等”
==      用于测试“相等”
```

这些关系运算符比较简单，没什么可讲的，但是我们要注意一些运算符使用时候的陷阱。

警告：
在编程的过程中== 和=不小心写错，导致的错误。

8. 逻辑操作符

逻辑操作符有哪些：

```
&&          逻辑与
||          逻辑或
```

区分**逻辑与**和**按位与**
区分**逻辑或**和**按位或**

```
1&2----->0
1&&2---->1

1|2----->3
1||2---->1
```

逻辑与和或的特点：

360笔试题

```
#include <stdio.h>

int main()
{
    int i = 0,a=0,b=2,c =3,d=4;
    i = a++ && ++b && d++;
    //i = a++||++b||d++;
    printf("a = %d\n b = %d\n c = %d\nd = %d\n", a, b, c, d);
    return 0;
}
//程序输出的结果是什么？
```

9. 条件操作符

```
exp1 ? exp2 : exp3
```

练习：

```
1.
if (a > 5)
        b = 3;
else
        b = -3;
转换成条件表达式，是什么样？

2.使用条件表达式实现找两个数中较大值。
```

10. 逗号表达式

```
exp1, exp2, exp3, …expN
```

逗号表达式，就是用逗号隔开的多个表达式。
逗号表达式，从左向右依次执行。整个表达式的结果是最后一个表达式的结果。

```
//代码1
int a = 1;
int b = 2;
int c = (a>b, a=b+10, a, b=a+1);//逗号表达式
c是多少？

//代码2
if (a =b + 1, c=a / 2, d > 0)

//代码3
a = get_val();
count_val(a);
while (a > 0)
{
```

```
        //业务处理
        a = get_val();
        count_val(a);
}

如果使用逗号表达式，改写：
while (a = get_val(), count_val(a), a>0)
{
        //业务处理
}
```

11. 下标引用、函数调用和结构成员

1. [] 下标引用操作符

操作数：一个数组名 + 一个索引值

```
    int arr[10];//创建数组
    arr[9] = 10;//实用下标引用操作符。
    [ ]的两个操作数是arr和9。
```

2. ( ) 函数调用操作符

接受一个或者多个操作数：第一个操作数是函数名，剩余的操作数就是传递给函数的参数。

```
    #include <stdio.h>
    void test1()
    {
        printf("hehe\n");
    }
    void test2(const char *str)
    {
        printf("%s\n", str);
    }
    int main()
    {
        test1();            //实用（）作为函数调用操作符。
        test2("hello bit.");//实用（）作为函数调用操作符。
        return 0;
    }
```

3. 访问一个结构的成员

> . 结构体.成员名
>
> -> 结构体指针->成员名

```
#include <stdio.h>
struct Stu
{
    char name[10];
    int age;
    char sex[5];
    double score;
```

```
};

void set_age1(struct Stu stu)
{
    stu.age = 18;
}
void set_age2(struct Stu* pStu)
{
    pStu->age = 18;//结构成员访问
}
int main()
{
    struct Stu stu;
    struct Stu* pStu = &stu;//结构成员访问

    stu.age = 20;//结构成员访问
    set_age1(stu);

    pStu->age = 20;//结构成员访问
    set_age2(pStu);
    return 0;
}
```

12. 表达式求值

表达式求值的顺序一部分是由操作符的优先级和结合性决定。

同样，有些表达式的操作数在求值的过程中可能需要转换为其他类型。

12.1 隐式类型转换

C的整型算术运算总是至少以缺省整型类型的精度来进行的。

为了获得这个精度，表达式中的字符和短整型操作数在使用之前被转换为普通整型，这种转换称为**整型提升**。

整型提升的意义：

> 表达式的整型运算要在CPU的相应运算器件内执行，CPU内整型运算器(ALU)的操作数的字节长度一般就是int的字节长度，同时也是CPU的通用寄存器的长度。
>
> 因此，即使两个char类型的相加，在CPU执行时实际上也要先转换为CPU内整型操作数的标准长度。
>
> 通用CPU（general-purpose CPU）是难以直接实现两个8比特字节直接相加运算（虽然机器指令中可能有这种字节相加指令）。所以，表达式中各种长度可能小于int长度的整型值，都必须先转换为int或unsigned int，然后才能送入CPU去执行运算。

```
//实例1
char a,b,c;
...
a = b + c;
```

b和c的值被提升为普通整型，然后再执行加法运算。

加法运算完成之后，结果将被截断，然后再存储于a中。

如何进行整体提升呢?

> 整形提升是按照变量的数据类型的符号位来提升的

```
//负数的整形提升
char c1 = -1;
变量c1的二进制位(补码)中只有8个比特位:
1111111
因为 char 为有符号的 char
所以整形提升的时候，高位补充符号位，即为1
提升之后的结果是:
11111111111111111111111111111111

//正数的整形提升
char c2 = 1;
变量c2的二进制位(补码)中只有8个比特位:
00000001
因为 char 为有符号的 char
所以整形提升的时候，高位补充符号位，即为0
提升之后的结果是:
00000000000000000000000000000001

//无符号整形提升，高位补0
```

整形提升的例子:

```
//实例1
int main()
{
    char a = 0xb6;
    short b = 0xb600;
    int c = 0xb6000000;
    if(a==0xb6)
        printf("a");
    if(b==0xb600)
        printf("b");
    if(c==0xb6000000)
        printf("c");
    return 0;
}
```

实例1中的a,b要进行整形提升,但是c不需要整形提升
a,b整形提升之后,变成了负数,所以表达式 `a==0xb6` , `b==0xb600` 的结果是假,但是c不发生整形提升,则表达式 `c==0xb6000000` 的结果是真.

所程序输出的结果是:

```
c
```

```c
//实例2
int main()
{
    char c = 1;
    printf("%u\n", sizeof(c));
    printf("%u\n", sizeof(+c));
    printf("%u\n", sizeof(-c));
    return 0;
}
```

实例2中的,c只要参与表达式运算,就会发生整形提升,表达式`+c`,就会发生提升,所以`sizeof(+c)`是4个字节.

表达式`-c`也会发生整形提升,所以`sizeof(-c)`是4个字节,但是`sizeof(c)`,就是1个字节.

12.2 算术转换

如果某个操作符的各个操作数属于不同的类型，那么除非其中一个操作数的转换为另一个操作数的类型，否则操作就无法进行。下面的层次体系称为**寻常算术转换。**

```
long double
double
float
unsigned long int
long int
unsigned int
int
```

如果某个操作数的类型在上面这个列表中排名较低，那么首先要转换为另外一个操作数的类型后执行运算。

警告:
但是算术转换要合理，要不然会有一些潜在的问题。

```c
float f = 3.14;
int num = f;//隐式转换，会有精度丢失
```

12.3 操作符的属性

复杂表达式的求值有三个影响的因素。

1. 操作符的优先级
2. 操作符的结合性
3. 是否控制求值顺序。

两个相邻的操作符先执行哪个？取决于他们的优先级。如果两者的优先级相同，取决于他们的结合性。

操作符优先级

操作符	描述	用法示例	结果类型	结合性	是否控制求值顺序
()	聚组	（表达式）	与表达式同	N/A	否
()	函数调用	rexp（rexp，...,rexp）	rexp	L-R	否
[ ]	下标引用	rexp[rexp]	lexp	L-R	否
.	访问结构成员	lexp.member_name	lexp	L-R	否
->	访问结构指针成员	rexp->member_name	lexp	L-R	否
++	后缀自增	lexp ++	rexp	L-R	否
--	后缀自减	lexp --	rexp	L-R	否
!	逻辑反	! rexp	rexp	R-L	否
~	按位取反	~ rexp	rexp	R-L	否
+	单目，表示正值	+ rexp	rexp	R-L	否
-	单目，表示负值	- rexp	rexp	R-L	否
++	前缀自增	++ lexp	rexp	R-L	否
--	前缀自减	-- lexp	rexp	R-L	否
*	间接访问	* rexp	lexp	R-L	否
&	取地址	& lexp	rexp	R-L	否
sizeof	取其长度，以字节表示	sizeof rexp sizeof(类型)	rexp	R-L	否
(类型)	类型转换	(类型) rexp	rexp	R-L	否
*	乘法	rexp * rexp	rexp	L-R	否
/	除法	rexp / rexp	rexp	L-R	否
%	整数取余	rexp % rexp	rexp	L-R	否
+	加法	rexp + rexp	rexp	L-R	否
-	减法	rexp - rexp	rexp	L-R	否
<<	左移位	rexp << rexp	rexp	L-R	否
>>	右移位	rexp >> rexp	rexp	L-R	否
>	大于	rexp > rexp	rexp	L-R	否
>=	大于等于	rexp >= rexp	rexp	L-R	否
<	小于	rexp < rexp	rexp	L-R	否
<=	小于等于	rexp <= rexp	rexp	L-R	否

操作符	描述	用法示例	结果类型	结合性	是否控制求值顺序
==	等于	rexp == rexp	rexp	L-R	否
!=	不等于	rexp != rexp	rexp	L-R	否
&	位与	rexp & rexp	rexp	L-R	否
^	位异或	rexp ^ rexp	rexp	L-R	否
|	位或	rexp | rexp	rexp	L-R	否
&&	逻辑与	rexp && rexp	rexp	L-R	是
||	逻辑或	rexp || rexp	rexp	L-R	是
? :	条件操作符	rexp ? rexp : rexp	rexp	N/A	是
=	赋值	lexp = rexp	rexp	R-L	否
+=	以...加	lexp += rexp	rexp	R-L	否
-=	以...减	lexp -= rexp	rexp	R-L	否
*=	以...乘	lexp *= rexp	rexp	R-L	否
/=	以...除	lexp /= rexp	rexp	R-L	否
%=	以...取模	lexp %= rexp	rexp	R-L	否
<<=	以...左移	lexp <<= rexp	rexp	R-L	否
>>=	以...右移	lexp >>= rexp	rexp	R-L	否
&=	以...与	lexp &= rexp	rexp	R-L	否
^=	以...异或	lexp ^= rexp	rexp	R-L	否
|=	以...或	lexp |= rexp	rexp	R-L	否
,	逗号	rexp，rexp	rexp	L-R	是

一些问题表达式

```
//表达式的求值部分由操作符的优先级决定。
//表达式1
a*b + c*d + e*f
```

> 注释：代码1在计算的时候，由于*比+的优先级高，只能保证，*的计算是比+早，但是优先级并不能决定第三个*比第一个+早执行。

所以表达式的计算机顺序就可能是：

```
a*b
c*d
a*b + c*d
e*f
a*b + c*d + e*f
```

```
或者：
a*b
c*d
e*f

a*b + c*d
a*b + c*d + e*f
```

```
//表达式2
c + --c;
```

> 注释：同上，操作符的优先级只能决定自减--的运算在+的运算的前面，但是我们并没有办法得知，+操作符的左操作数的获取在右操作数之前还是之后求值，所以结果是不可预测的，是有歧义的。

```
//代码3-非法表达式
int main()
{
    int i = 10;
    i = i-- - --i * ( i = -3 ) * i++ + ++i;
    printf("i = %d\n", i);
    return 0;
}
```

表达式3在不同编译器中测试结果：非法表达式程序的结果

值	编译器
—128	Tandy 6000 Xenix 3.2
—95	Think C 5.02(Macintosh)
—86	IBM PowerPC AIX 3.2.5
—85	Sun Sparc cc(K&C编译器)
—63	gcc，HP_UX 9.0，Power C 2.0.0
4	Sun Sparc acc(K&C编译器)
21	Turbo C/C++ 4.5
22	FreeBSD 2.1 R
30	Dec Alpha OSF1 2.0
36	Dec VAX/VMS
42	Microsoft C 5.1

```c
//代码4
int fun()
{
     static int count = 1;
     return ++count;
}
int main()
{
     int answer;
     answer = fun() - fun() * fun();
     printf( "%d\n", answer);//输出多少？
     return 0;
}
```

这个代码有没有实际的问题？

有问题！

虽然在大多数的编译器上求得结果都是相同的。

但是上述代码`answer = fun() - fun() * fun();`中我们只能通过操作符的优先级得知：先算乘法，再算减法。

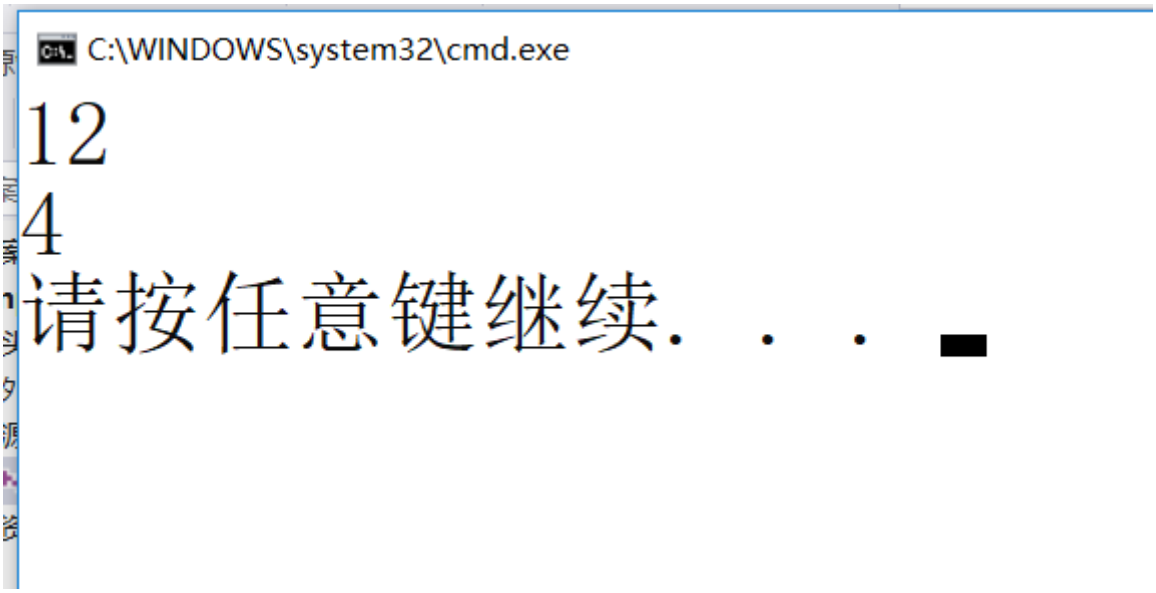
函数的调用先后顺序无法通过操作符的优先级确定。

```c
//代码5
#include <stdio.h>
int main()
{
    int i = 1;
    int ret = (++i) + (++i) + (++i);
    printf("%d\n", ret);
    printf("%d\n", i);
    return 0;
}
//尝试在linux 环境gcc编译器，VS2013环境下都执行，看结果。
```

Linux环境的结果：

```
[root@centos7net test]# ./a.out
10
4
```

VS2013环境的结果：



看看同样的代码产生了不同的结果，这是为什么？

简单看一下汇编代码.就可以分析清楚.

这段代码中的第一个`+`在执行的时候，第三个++是否执行，这个是不确定的，因为依靠操作符的优先级和结合性是无法决定第一个`+`和第

三个前置`++`的先后顺序。

总结：我们写出的表达式如果不能通过操作符的属性确定唯一的计算路径，那这个表达式就是存在问题的。

本章完

BIT-6-指针

1. 指针是什么
2. 指针和指针类型
3. 野指针
4. 指针运算
5. 指针和数组
6. 二级指针
7. 指针数组

正文开始@比特就业课

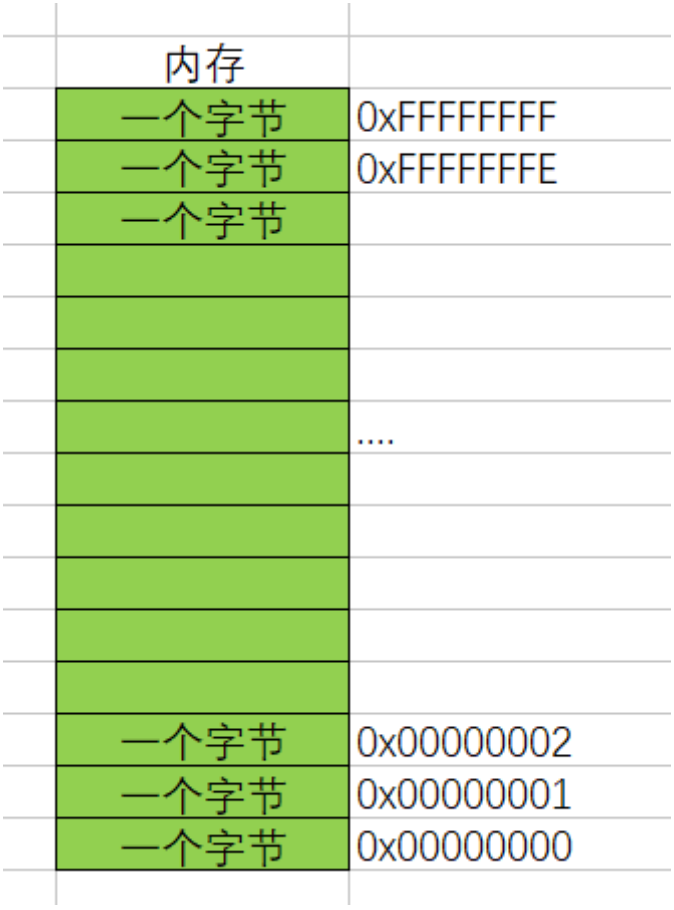
1. 指针是什么?

> 指针是什么?
>
> 指针理解的2个要点:
>
> 1. 指针是内存中一个最小单元的编号，也就是地址
> 2. 平时口语中说的指针，通常指的是指针变量，是用来存放内存地址的变量

总结：指针就是地址，口语中说的指针通常指的是指针变量。

那我们就可以这样理解：

> 内存



> 指针变量
>
> 我们可以通过&（取地址操作符）取出变量的内存其实地址，把地址可以存放到一个变量中，这个变量就是指针变量

```
#include <stdio.h>
int main()
{
    int a = 10;//在内存中开辟一块空间
    int *p = &a;//这里我们对变量a，取出它的地址，可以使用&操作符。
                //a变量占用4个字节的空间，这里是将a的4个字节的第一个字节的地址存放在p变量中，p就是一个之指针变量。
    return 0;
}
```

总结:

指针变量，用来存放地址的变量。（存放在指针中的值都被当成地址处理）。

那这里的问题是:

- 一个小的单元到底是多大？（1个字节）
- 如何编址?

经过仔细的计算和权衡我们发现一个字节给一个对应的地址是比较合适的。

对于32位的机器，假设有32根地址线，那么假设每根地址线在寻址的时候产生高电平（高电压）和低电平（低电压）就是（1或者0）；

那么32根地址线产生的地址就会是:

> 00000000 00000000 00000000 00000000
>
> 00000000 00000000 00000000 00000001
>
> ...
>
> 11111111 11111111 11111111 11111111

这里就有2的32次方个地址。

每个地址标识一个字节，那我们就可以给`（2^32Byte == 2^32/1024KB == 2^32/1024/1024MB==2^32/1024/1024/1024GB == 4GB）`4G的空闲进行编址。

同样的方法，那64位机器，如果给64根地址线，那能编址多大空间，自己计算。

这里我们就明白：

- 在32位的机器上，地址是32个0或者1组成二进制序列，那地址就得用4个字节的空间来存储，所以一个指针变量的大小就应该是4个字节。
- 那如果在64位机器上，如果有64个地址线，那一个指针变量的大小是8个字节，才能存放一个地址。

总结：

- 指针是用来存放地址的，地址是唯一标示一块地址空间的。
- **指针的大小在32位平台是4个字节，在64位平台是8个字节。**

2. 指针和指针类型

这里我们在讨论一下：指针的类型
我们都知道，变量有不同的类型，整形，浮点型等。那指针有没有类型呢？
准确的说：有的。

当有这样的代码：

```
int num = 10;
p = &num;
```

要将&num（num的地址）保存到p中，我们知道p就是一个指针变量，那它的类型是怎样的呢？
我们给指针变量相应的类型。

```
char  *pc = NULL;
int   *pi = NULL;
short *ps = NULL;
long  *pl = NULL;
float *pf = NULL;
double *pd = NULL;
```

这里可以看到，指针的定义方式是：`type + *`。
其实：
`char*`类型的指针是为了存放`char`类型变量的地址。
`short*`类型的指针是为了存放`short`类型变量的地址。
`int*`类型的指针是为了存放`int`类型变量的地址。

那指针类型的意义是什么？

2.1 指针+-整数

```
#include <stdio.h>
//演示实例
int main()
{
    int n = 10;
    char *pc = (char*)&n;
    int *pi = &n;

    printf("%p\n", &n);
    printf("%p\n", pc);
    printf("%p\n", pc+1);
    printf("%p\n", pi);
    printf("%p\n", pi+1);
    return  0;
}
```

总结：指针的类型决定了指针向前或者向后走一步有多大（距离）。

2.2 指针的解引用

```
//演示实例
#include <stdio.h>

int main()
{
    int n = 0x11223344;
    char *pc = (char *)&n;
    int *pi = &n;
    *pc = 0;    //重点在调试的过程中观察内存的变化。
    *pi = 0;    //重点在调试的过程中观察内存的变化。
    return 0;
}
```

总结：
指针的类型决定了，对指针解引用的时候有多大的权限（能操作几个字节）。

比如：`char*`的指针解引用就只能访问一个字节，而`int*`的指针的解引用就能访问四个字节。

3. 野指针

> 概念： 野指针就是指针指向的位置是不可知的（随机的、不正确的、没有明确限制的）

3.1 野指针成因

1. 指针未初始化

```c
#include <stdio.h>
int main()
{
    int *p;//局部变量指针未初始化，默认为随机值
    *p = 20;
    return 0;
}
```

2. 指针越界访问

```c
#include <stdio.h>
int main()
{
    int arr[10] = {0};
    int *p = arr;
    int i = 0;
    for(i=0; i<=11; i++)
    {
        //当指针指向的范围超出数组arr的范围时，p就是野指针
        *(p++) = i;
    }
    return 0;
}
```

3. 指针指向的空间释放

   这里放在动态内存开辟的时候讲解，这里可以简单提示一下。

3.2 如何规避野指针

1. 指针初始化
2. 小心指针越界
3. 指针指向空间释放即使置NULL
4. 避免返回局部变量的地址
5. 指针使用之前检查有效性

```c
#include <stdio.h>
int main()
{
    int *p = NULL;
    //....
    int a = 10;
    p = &a;
    if(p != NULL)
    {
        *p = 20;
    }
    return 0;
}
```

4. 指针运算

- 指针+- 整数
- 指针-指针
- 指针的关系运算

4.1 指针+-整数

```
#define N_VALUES 5
float values[N_VALUES];
float *vp;
//指针+-整数；指针的关系运算
for (vp = &values[0]; vp < &values[N_VALUES];)
{
     *vp++ = 0;
}
```

4.2 指针-指针

```
int my_strlen(char *s)
{
        char *p = s;
        while(*p != '\0' )
                p++;
        return p-s;
}
```

4.3 指针的关系运算

```
for(vp = &values[N_VALUES]; vp > &values[0];)
{
    *--vp = 0;
}
```

代码简化, 这将代码修改如下:

```
for(vp = &values[N_VALUES-1]; vp >= &values[0];vp--)
{
    *vp = 0;
}
```

实际在绝大部分的编译器上是可以顺利完成任务的，然而我们还是应该避免这样写，因为标准并不保证它可行。

标准规定:

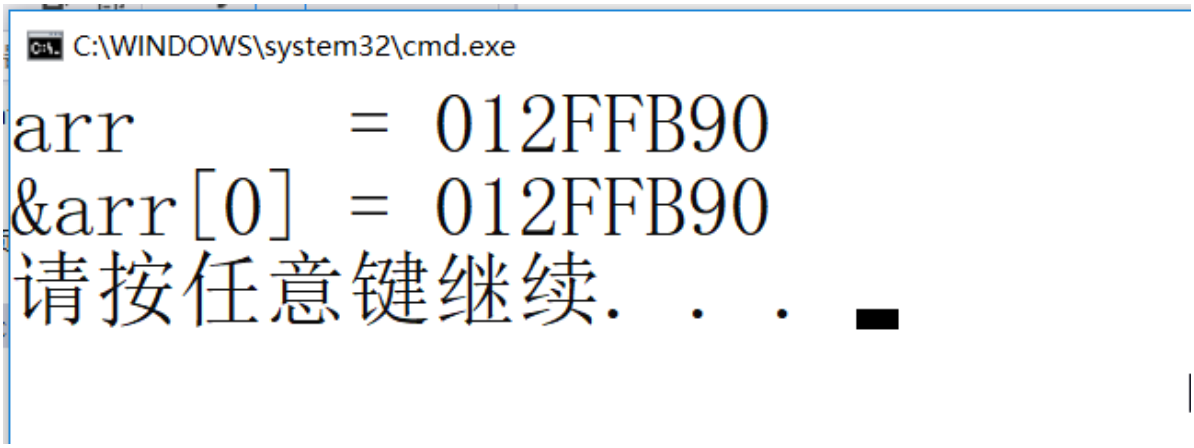
> 允许指向数组元素的指针与指向数组最后一个元素后面的那个内存位置的指针比较，但是不允许与指向第一个元素之前的那个内存位置的指针进行比较。

5. 指针和数组

我们看一个例子：

```
#include <stdio.h>
int main()
{
    int arr[10] = {1,2,3,4,5,6,7,8,9,0};
    printf("%p\n", arr);
    printf("%p\n", &arr[0]);
    return 0;
}
```

运行结果：



可见数组名和数组首元素的地址是一样的。

结论：**数组名表示的是数组首元素的地址。**（2种情况除外，数组章节讲解了）

那么这样写代码是可行的：

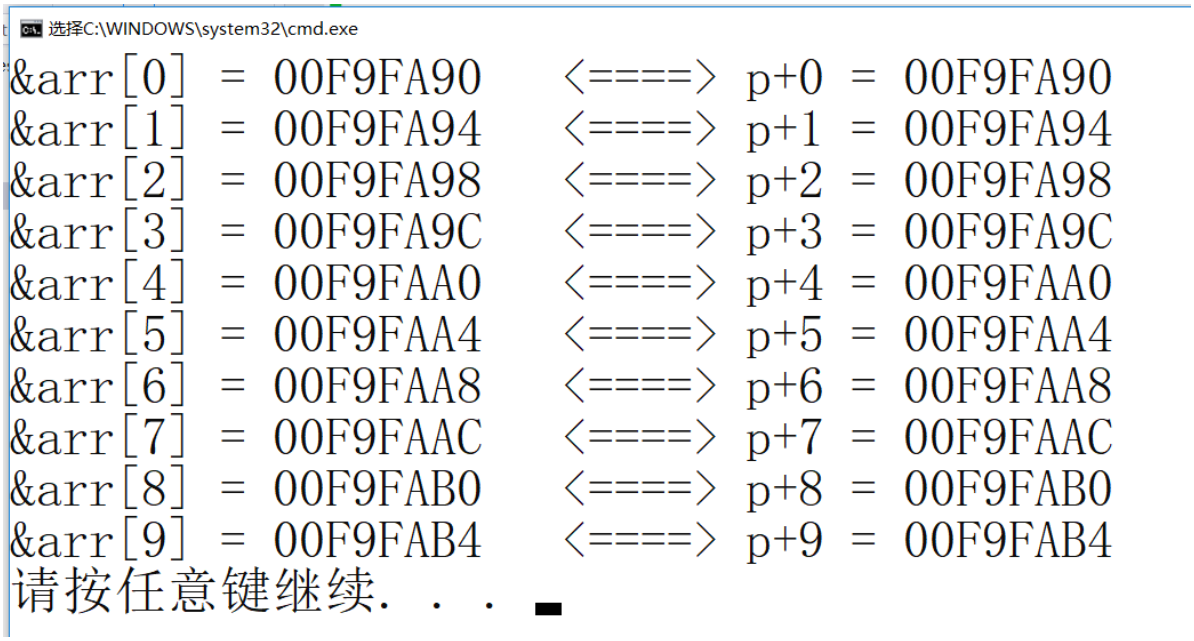
```
int arr[10] = {1,2,3,4,5,6,7,8,9,0};
int *p = arr;//p存放的是数组首元素的地址
```

既然可以把数组名当成地址存放到一个指针中，我们使用指针来访问一个就成为可能。

例如：

```
#include <stdio.h>

int main()
{
    int arr[] = {1,2,3,4,5,6,7,8,9,0};
    int *p = arr; //指针存放数组首元素的地址
    int sz = sizeof(arr)/sizeof(arr[0]);
    for(i=0; i<sz; i++)
    {
        printf("&arr[%d] = %p    <====> p+%d = %p\n", i, &arr[i], i, p+i);
    }
    return 0;
}
```

运行结果：



所以`p+i`其实计算的是数组`arr`下标为i的地址。

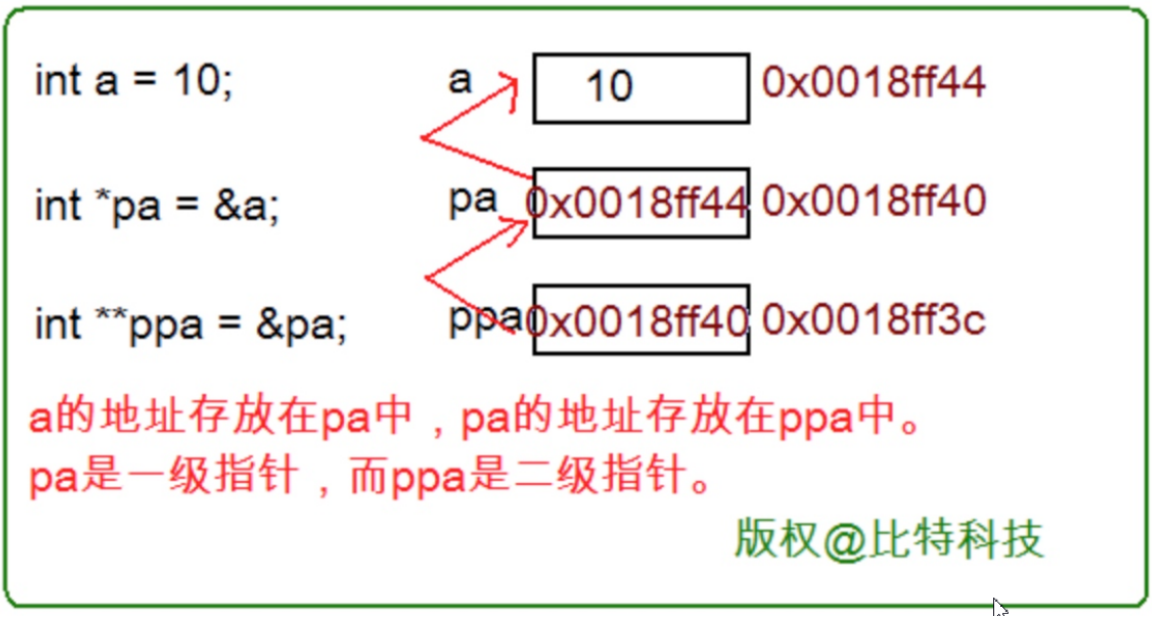
那我们就可以直接通过指针来访问数组。

如下：

```
int main()
{
    int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
    int *p = arr; //指针存放数组首元素的地址
    int sz = sizeof(arr) / sizeof(arr[0]);
    int i = 0;
    for (i = 0; i<sz; i++)
    {
        printf("%d ", *(p + i));
    }
    return 0;
}
```

6. 二级指针

指针变量也是变量，是变量就有地址，那指针变量的地址存放在哪里？
这就是`二级指针`。



对于二级指针的运算有：

- `*ppa`通过对ppa中的地址进行解引用，这样找到的是`pa`，`*ppa`其实访问的就是`pa`.

```
int b = 20;
*ppa = &b;//等价于 pa = &b;
```

- `**ppa`先通过`*ppa`找到`pa`,然后对`pa`进行解引用操作：`*pa`，那找到的是`a`.

```
**ppa = 30;
//等价于*pa = 30;
//等价于a = 30;
```

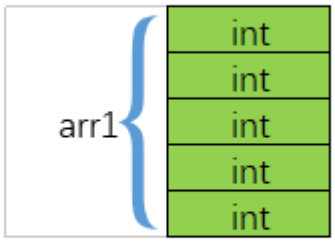
7. 指针数组

指针数组是指针还是数组？

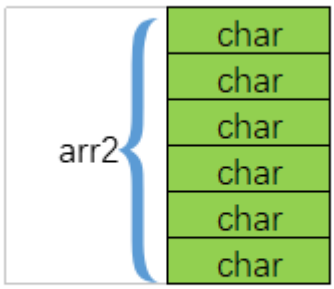
答案：是数组。是存放指针的数组。

数组我们已经知道整形数组，字符数组。

```
int arr1[5];
char arr2[6];
```



整形数组

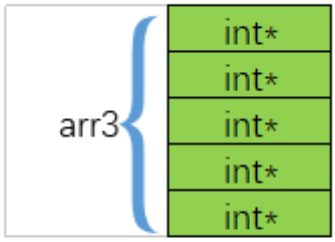


字符数组

那指针数组是怎样的?

```
int* arr3[5];//是什么?
```

arr3是一个数组，有五个元素，每个元素是一个整形指针。



整形指针数组

本章完

BIT-7-结构体

- 结构体类型的声明
- 结构体初始化
- 结构体成员访问
- 结构体传参

正文开始@比特就业课

<struct关键字> [标签名称]{
[成员列表]
}结构体变量;

1. 结构体的声明

1.1 结构的基础知识

> 结构是一些值的集合，这些值称为成员变量。结构的每个成员可以是不同类型的变量。

1.2 结构的声明

```
struct tag
{
    member-list;
}variable-list;
```

例如描述一个学生：

```
typedef struct Stu
{
    char name[20];//名字
    int age;//年龄
    char sex[5];//性别
    char id[20];//学号
}Stu; //分号不能丢
```

1.3 结构成员的类型

结构的成员可以是标量、数组、指针，甚至是其他结构体。

1.4 结构体变量的定义和初始化

有了结构体类型，那如何定义变量，其实很简单。

在结构体声明以后, 可以在main函数中定义变量, 如何定义?
struct tag 变量名称;

```
struct Point
{
    int x;
    int y;
}p1;                //声明类型的同时定义变量p1
struct Point p2;    //定义结构体变量p2


//初始化：定义变量的同时赋初值。
```

```
struct Point p3 = {x, y};

struct Stu        //类型声明
{
    char name[15];//名字
    int age;      //年龄
};
struct Stu s = {"zhangsan", 20};//初始化

struct Node
{
    int data;
    struct Point p;
    struct Node* next;
}n1 = {10, {4,5}, NULL};            //结构体嵌套初始化

struct Node n2 = {20, {5, 6}, NULL};//结构体嵌套初始化
```

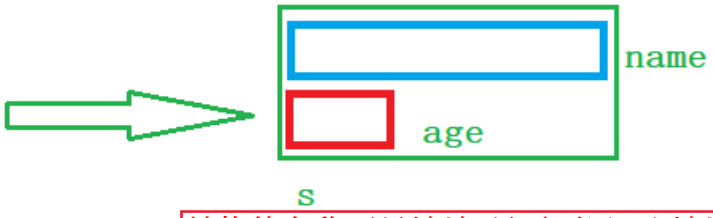
{}表示初始化, 一一对应的填入数值即可, 和数组初始化类似, 注意';'表示该语句结束, 故而{}是在;前

这里可以理解为多维数组, 若没有一一对应, 则是不完全初始化

2. 结构体成员的访问

- 结构体变量访问成员
  结构变量的成员是通过点操作符（.）访问的。点操作符接受两个操作数。
  例如：

```
struct Stu
{
    char name[20];
    int age;
};

struct Stu s;
```



结构体名称不是地址, 是需要用&取地址.
若将p1结构体传递个某个函数, 可以function(&p1);
void function(struct Peo* sp){
  printf("%s %s\n",sp->name,sp->tele);
}

我们可以看到 s 有成员 name 和 age；
那我们如何访问s的成员?

```
struct S s;
strcpy(s.name, "zhangsan");//使用.访问name成员
s.age = 20;//使用.访问age成员
```

- 结构体指针访问指向变量的成员
  有时候我们得到的不是一个结构体变量，而是指向一个结构体的指针。

  那该如何访问成员。
  如下：

点操作数是对非地址处理的, 若是指针, 得用->箭头操作数

```
struct Stu
{
    char name[20];
    int age;
};
```

```c
void print(struct Stu* ps)
{
    printf("name = %s   age = %d\n", (*ps).name, (*ps).age);
    //使用结构体指针访问指向对象的成员
    printf("name = %s   age = %d\n", ps->name, ps->age);
}
int main()
{
    struct Stu s = {"zhangsan", 20};
    print(&s);//结构体地址传参
    return 0;
}
```

3. 结构体传参

要么传结构体变量本身(相当于拷贝), 要么传递结构体地址(相当于链接)

直接上代码:

```c
struct S
{
    int data[1000];
    int num;
};

struct S s = {{1,2,3,4}, 1000};
//结构体传参
void print1(struct S s)
{
    printf("%d\n", s.num);
}
//结构体地址传参
void print2(struct S* ps)
{
    printf("%d\n", ps->num);
}

int main()
{
    print1(s);  //传结构体
    print2(&s); //传地址
    return 0;
}
```

上面的 `print1` 和 `print2` 函数哪个好些?

答案是: 首选print2函数。
原因:

> 函数传参的时候, 参数是需要压栈的。
> 如果传递一个结构体对象的时候, 结构体过大, 参数压栈的的系统开销比较大, 所以会导致性能的下降。

结论:
结构体传参的时候, 要传结构体的地址。

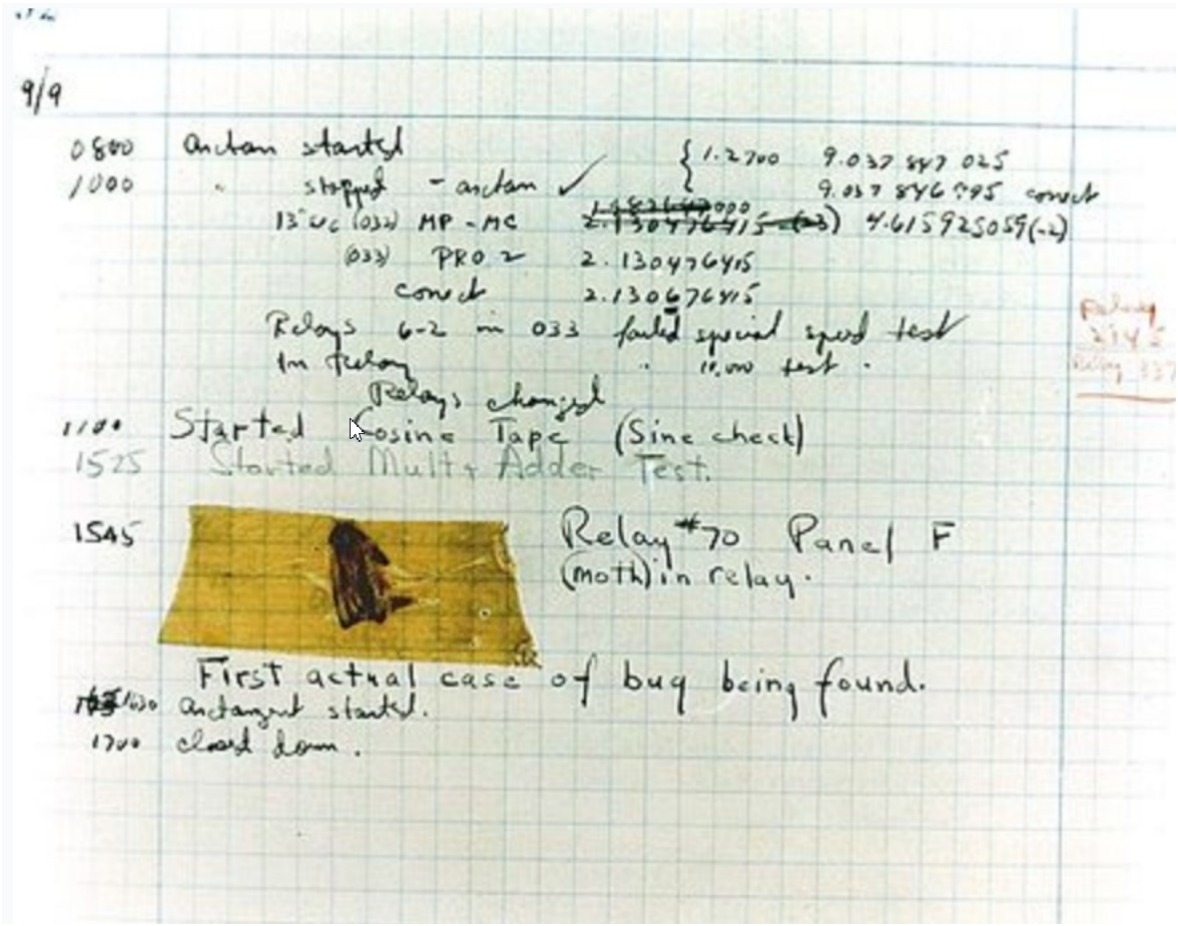
本章完

BIT-8-实用调试技巧

- 什么是bug?
- 调试是什么？有多重要?
- debug和release的介绍。
- windows环境调试介绍。
- 一些调试的实例。
- 如何写出好（易于调试）的代码。
- 编程常见的错误。

正文开始@比特就业课

1. 什么是bug?



第一次被发现的导致计算机错误的飞蛾，也是第一个计算机程序错误。

注：参考资料

2. 调试是什么？有多重要?

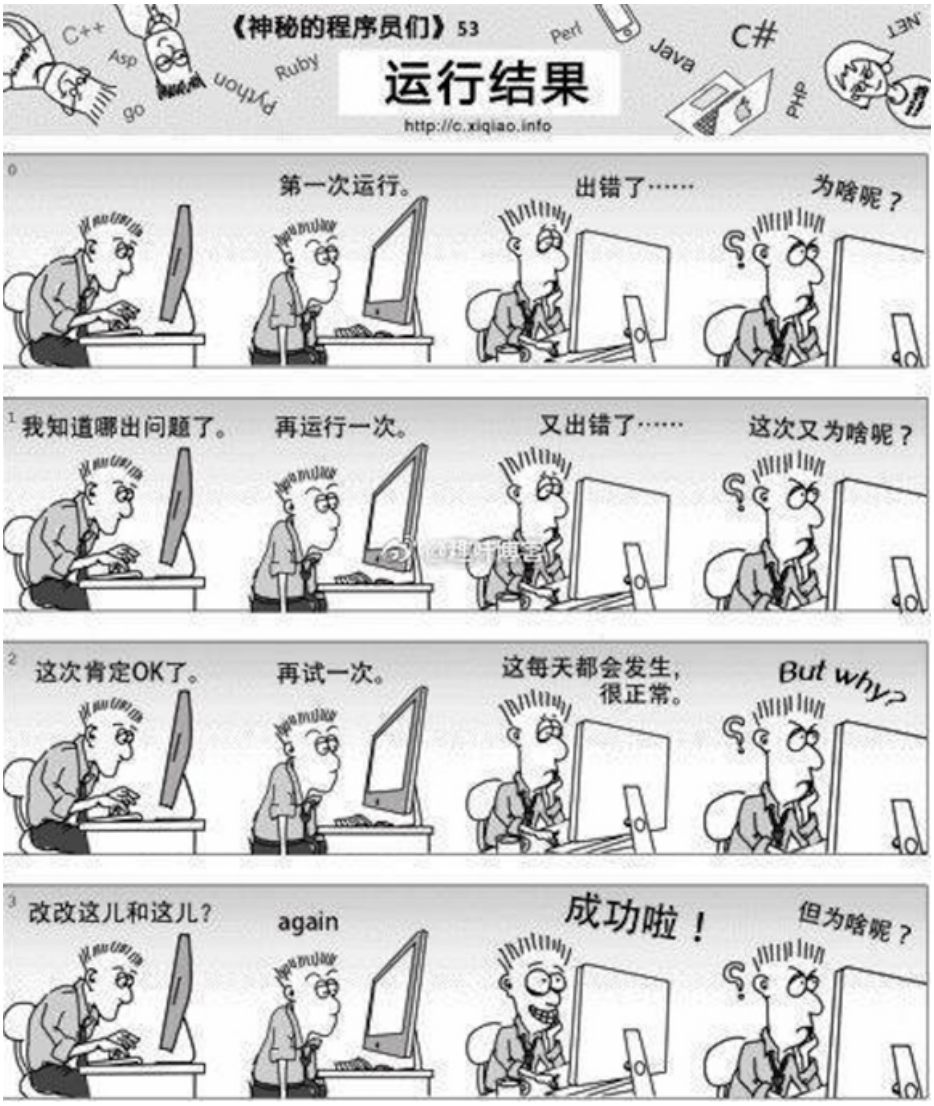
> 所有发生的事情都一定有迹可循，如果问心无愧，就不需要掩盖也就没有迹象了，如果问心有愧，就必然需要掩盖，那就一定会有迹象，**迹象越多就越容易顺藤而上，这就是推理的途径。**
>
> 顺着这条途径顺流而下就是犯罪，逆流而上，就是真相。

一名优秀的程序员是一名出色的侦探。

> 每一次调试都是尝试破案的过程。

我们是如何写代码的?

又是如何排查出现的问题的呢?



拒绝-迷信式调试！！！！

2.1 调试是什么?

> **调试**（英语：Debugging / Debug），又称除错，是发现和减少计算机程序或电子仪器设备中程序错误的一个过程。

2.2 调试的基本步骤

- 发现程序错误的存在
- 以隔离、消除等方式对错误进行定位
- 确定错误产生的原因
- 提出纠正错误的解决办法
- 对程序错误予以改正，重新测试

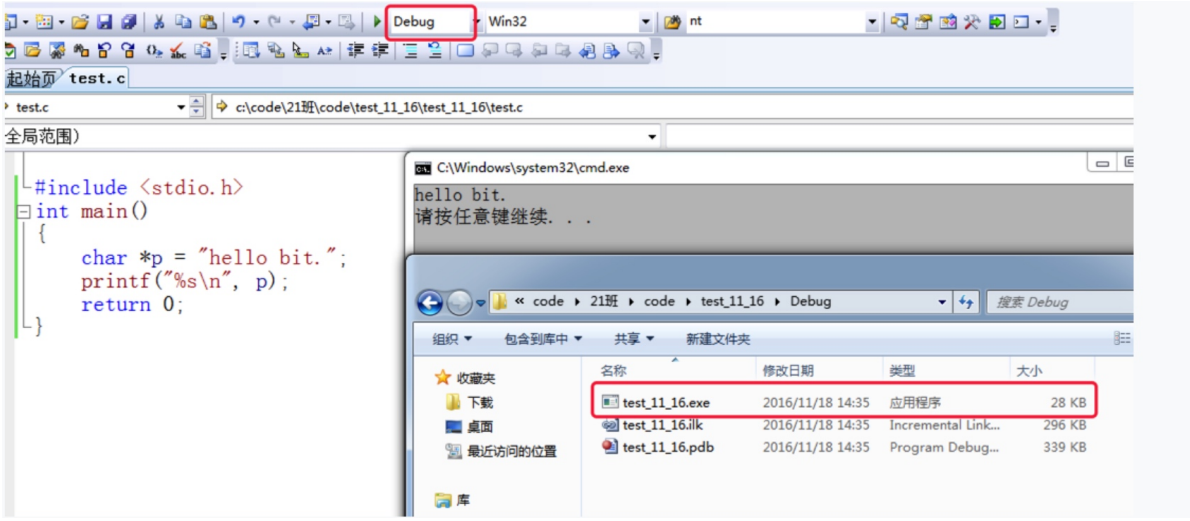
2.3 Debug和Release的介绍。

> **Debug 通常称为调试版本**，它包含调试信息，并且不作任何优化，便于程序员调试程序。程序员版本
>
> **Release 称为发布版本**，它往往是进行了各种优化，使得程序在代码大小和运行速度上都是最优的，以便用户很好地使用。用户版本

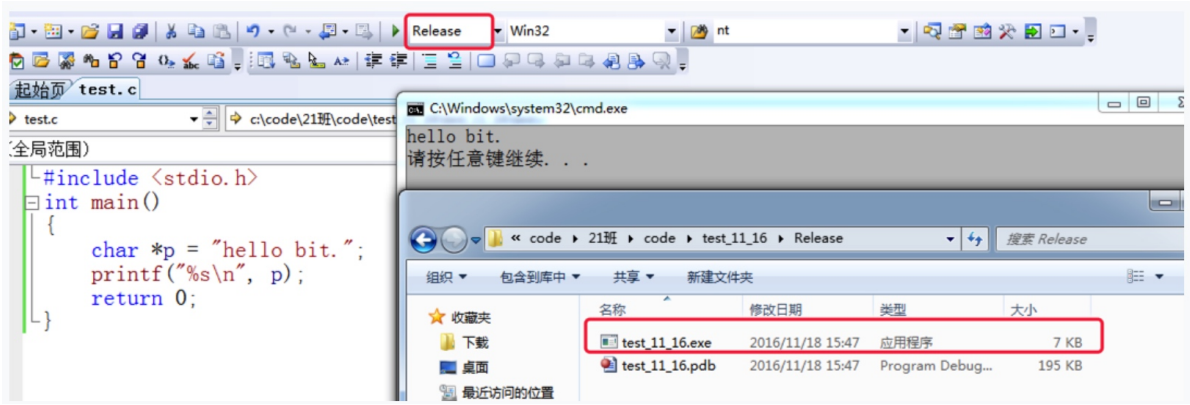
代码：

```
#include <stdio.h>
int main()
{
    char *p = "hello bit.";
    printf("%s\n", p);
    return 0;
}
```

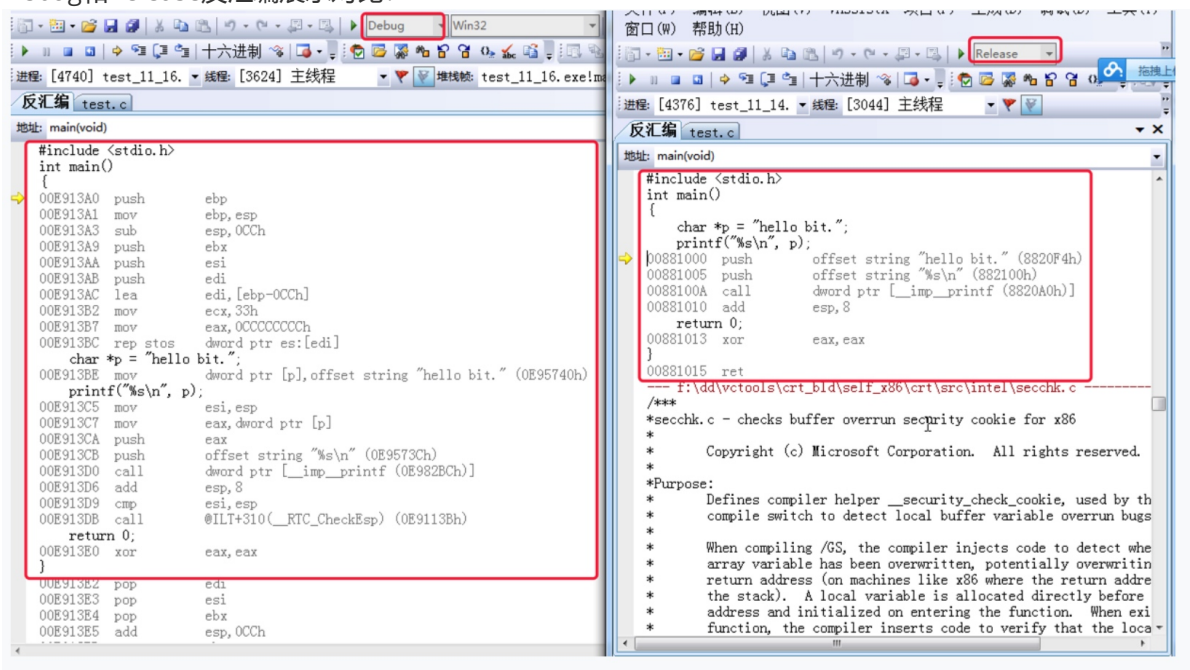
上述代码在Debug环境的结果展示：



上述代码在Release环境的结果展示：



Debug和Release反汇编展示对比：

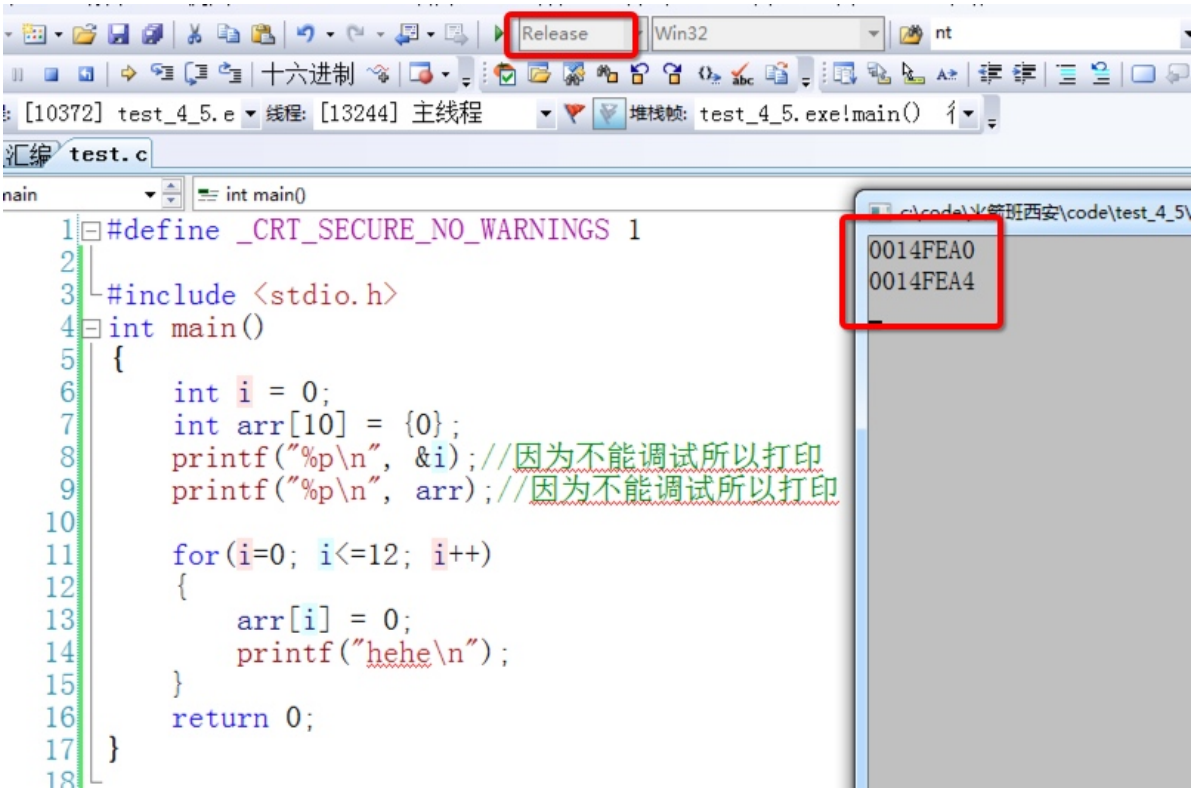


所以我们说调试就是在**Debug版本**的环境中，找代码中**潜伏的问题**的一个过程。

那编译器进行了哪些优化呢？
请看如下代码：

```
#include <stdio.h>
int main()
{
    int i = 0;
    int arr[10] = {0};
    for(i=0; i<=12; i++)
    {
        arr[i] = 0;
        printf("hehe\n");
    }
    return 0;
}
```

如果是`debug`模式去编译，程序的结果是死循环。
如果是`release`模式去编译，程序没有死循环。
那他们之间有什么区别呢？
就是因为优化导致的。

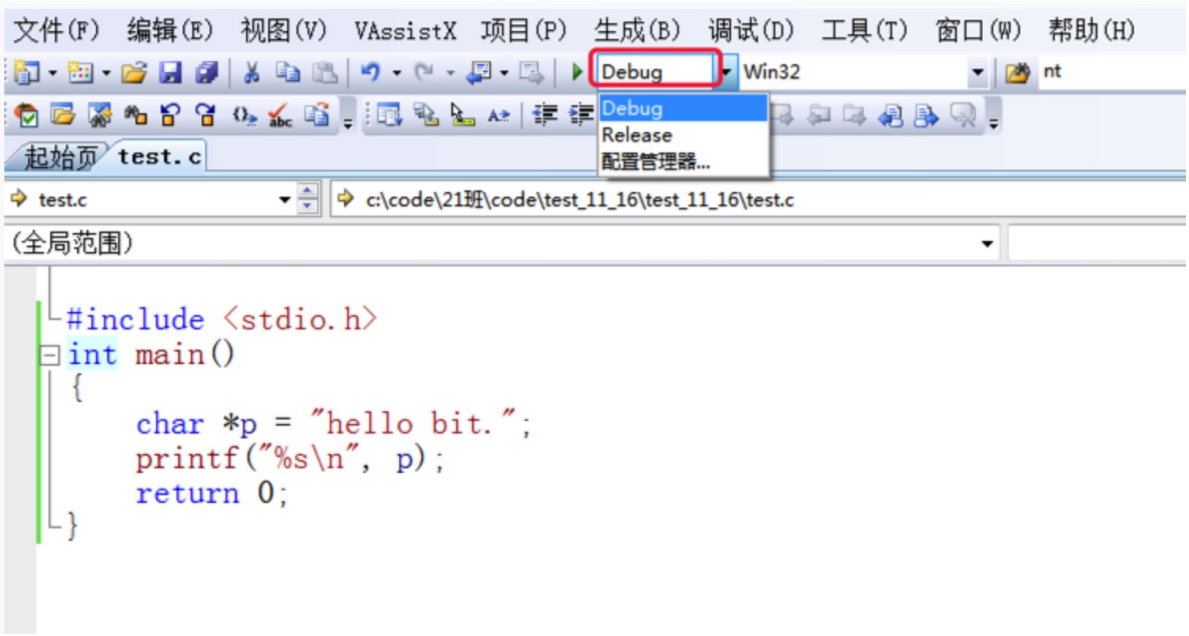


变量在内存中开辟的顺序发生了变化，影响到了程序执行的结果。

3. Windows环境调试介绍

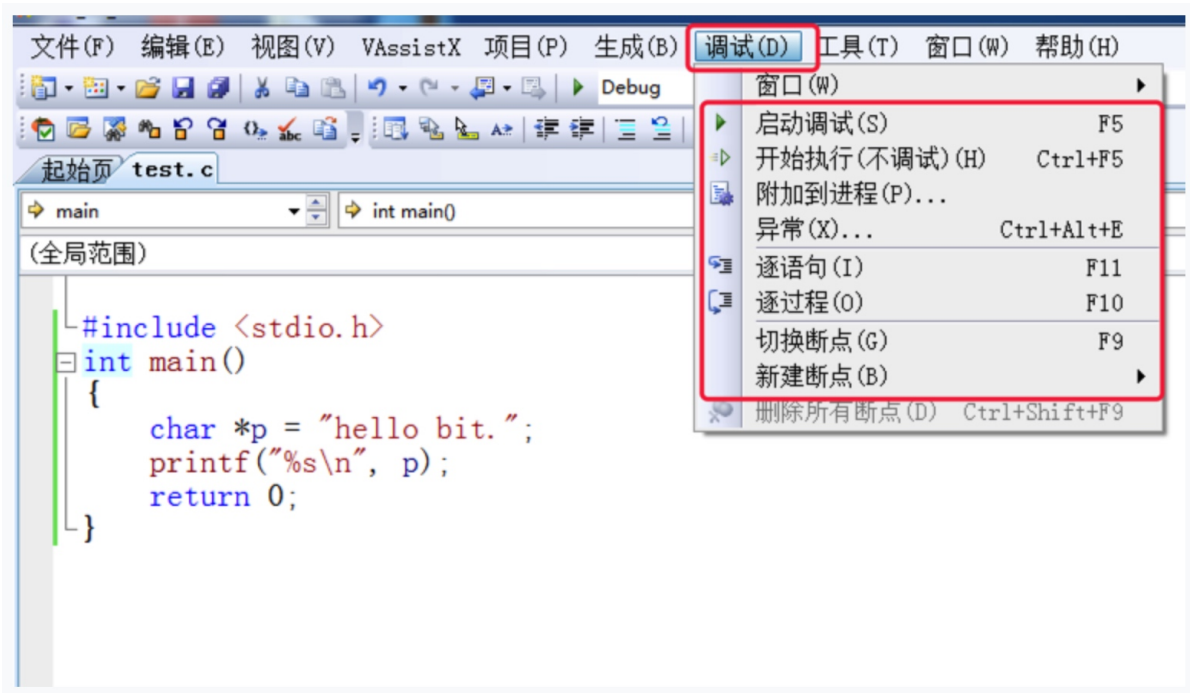
> 注：linux开发环境调试工具是gdb，后期课程会介绍。

3.1 调试环境的准备



在环境中选择 debug 选项，才能使代码正常调试。

3.2 学会快捷键



最常使用的几个快捷键：

F5

> 启动调试，经常用来直接跳到下一个断点处。这里是逻辑上的下一个断点

F9

鼠标先点击你要设置断点的地方，然后按下F9设置断点，注意scanf()是需要停下来输入数值的

> 创建断点和取消断点
>
> **断点**的重要作用，可以在程序的任意位置设置断点。
>
> 这样就可以使得程序在想要的位置随意停止执行，继而一步步执行下去。

F10

> 逐过程，通常用来处理一个过程，一个过程可以是一次函数调用，或者是一条语句。

F11

> 逐语句，就是每次都执行一条语句，但是这个快捷键可以使我们的执行逻辑**进入函数内部**（这是最长用的）。

CTRL + F5

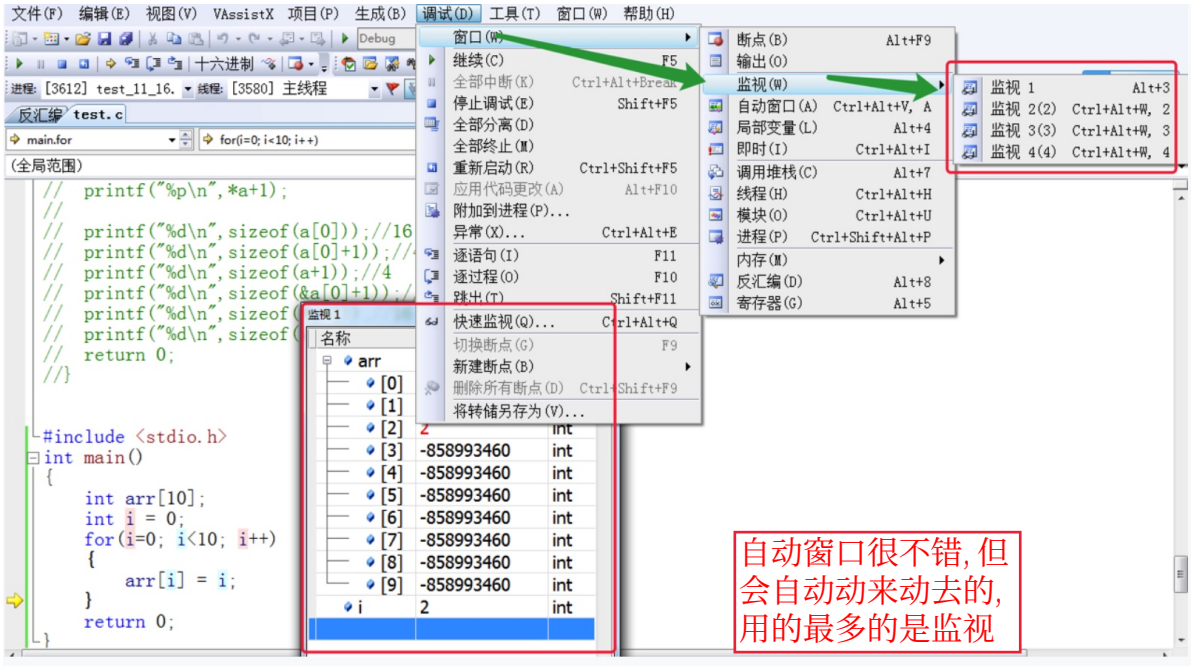
> 开始执行不调试，如果你想让程序直接运行起来而不调试就可以直接使用。

想知道更多快捷键？点我

3.3 调试的时候查看程序当前信息

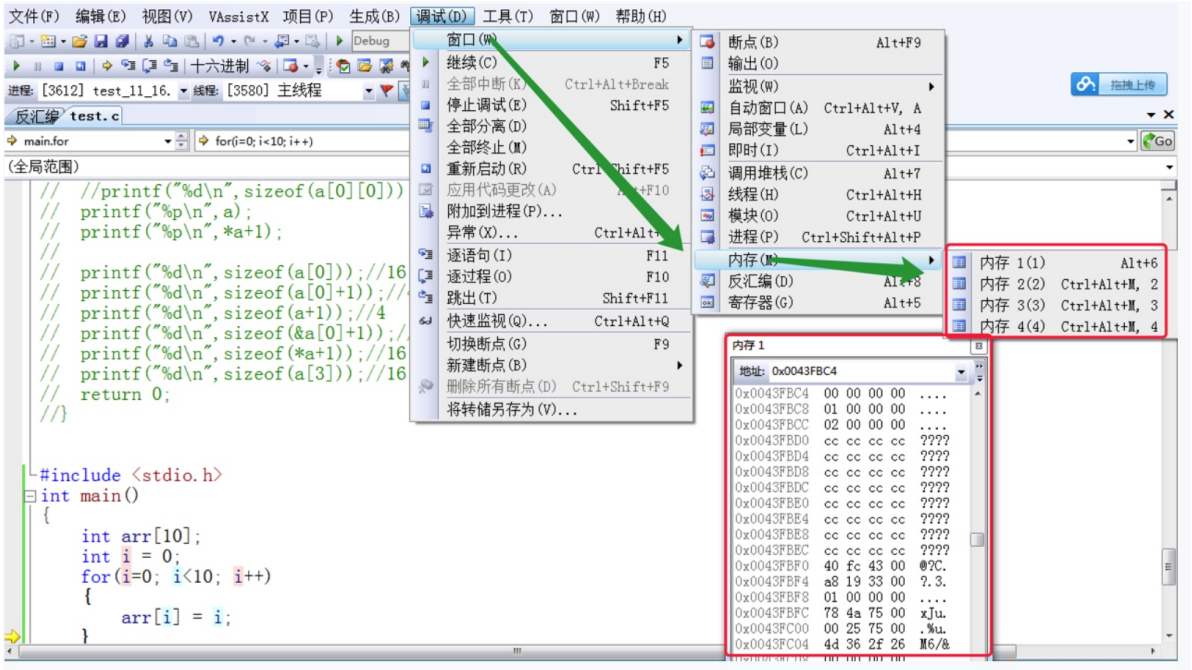
3.3.1 查看临时变量的值

在调试开始之后，用于观察变量的值。

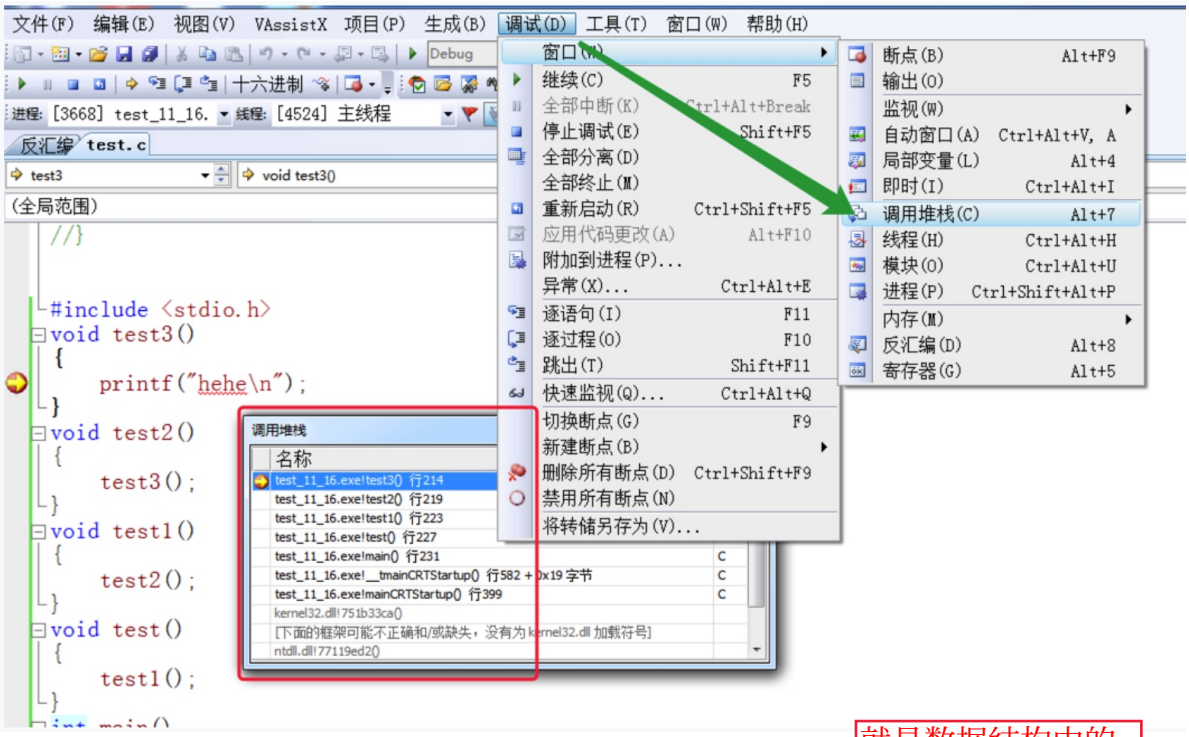


3.3.2 查看内存信息

在调试开始之后，用于观察内存信息。



3.3.3 查看调用堆栈



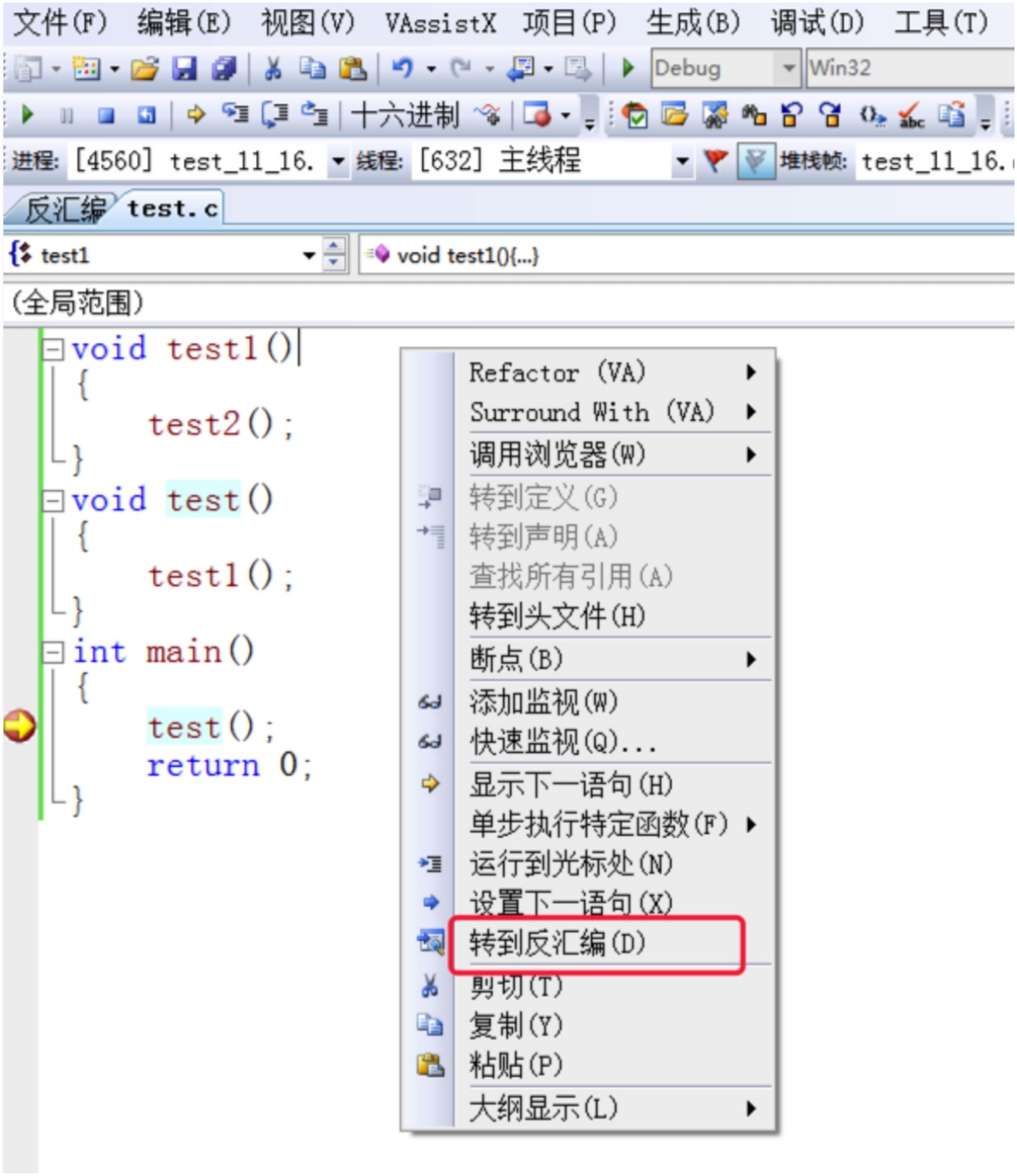
通过调用堆栈，可以清晰的反应函数的调用关系以及当前调用所处的位置。

就是数据结构中的压栈出栈

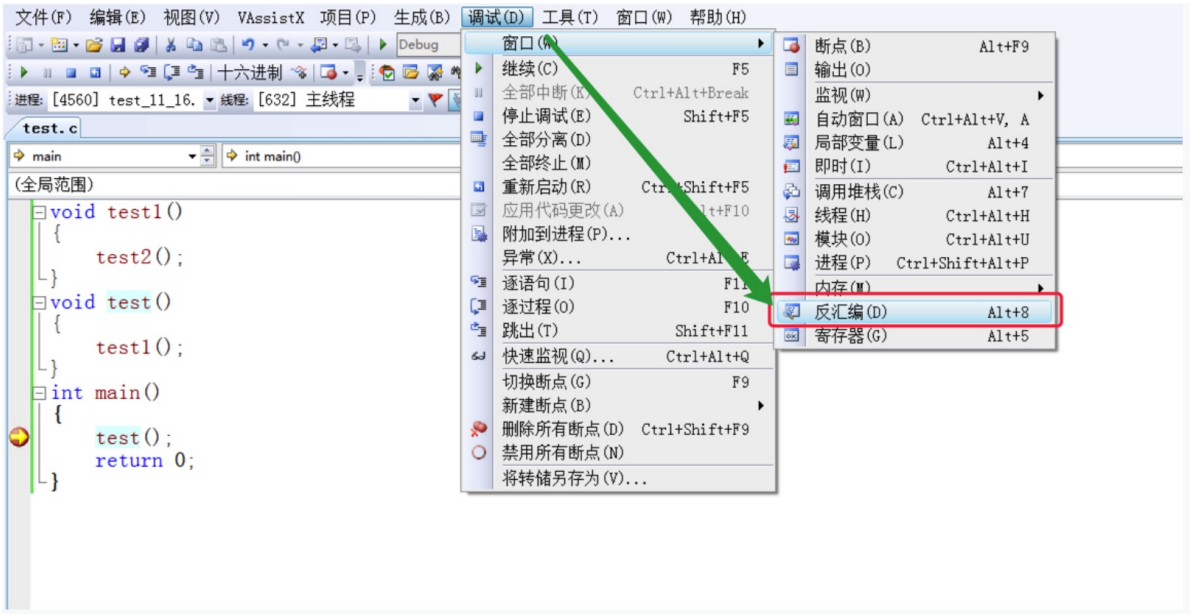
3.3.4 查看汇编信息

在调试开始之后，有两种方式转到汇编：

（1）第一种方式：右击鼠标，选择【转到反汇编】：

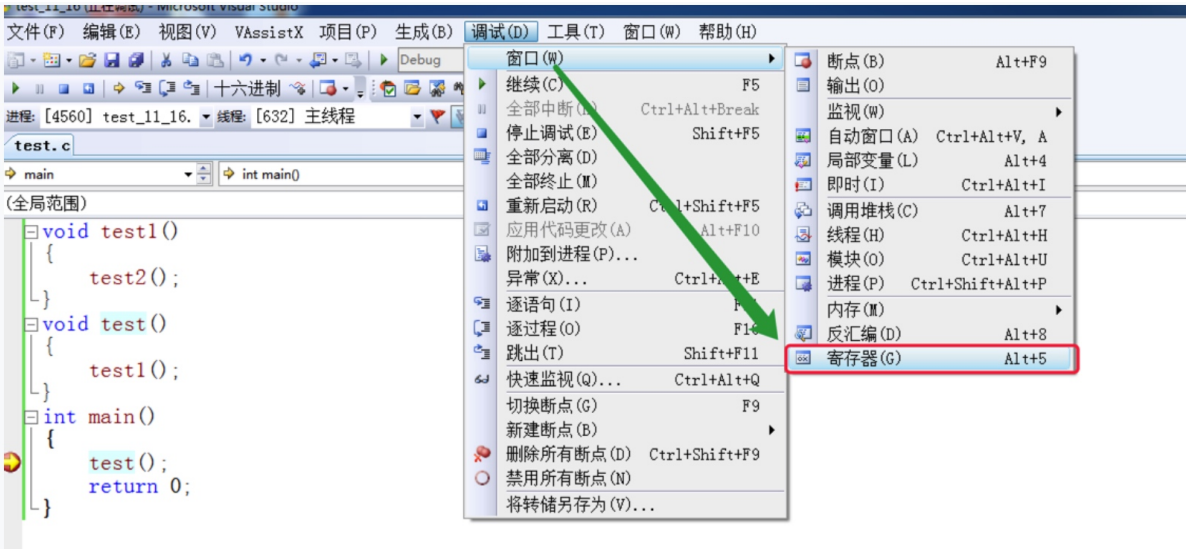


（2）第二种方式：



可以切换到汇编代码。

3.3.5 查看寄存器信息



可以查看当前运行环境的寄存器的使用信息。

4.多多动手，尝试调试，才能有进步。

> - 一定要熟练掌握调试技巧。
> - 初学者可能80%的时间在写代码，20%的时间在调试。但是一个程序员可能20%的时间在写程序，但是80%的时间在调试。
> - 我们所讲的都是一些简单的调试。
>   以后可能会出现很复杂调试场景：多线程程序的调试等。
> - 多多使用快捷键，提升效率。

5. 一些调试的实例

5.1 实例一

实现代码：求`1! +2! +3! ...+ n!`；不考虑溢出。

```
int main()
{
    int i = 0;
    int sum = 0;//保存最终结果
    int n = 0;
    int ret = 1;//保存n的阶乘
    scanf("%d", &n);
    for(i=1; i<=n; i++)
    {
        int j = 0;
        for(j=1; j<=i; j++)
        {
            ret *= j;
        }
        sum += ret;
    }
    printf("%d\n", sum);
    return 0;
}
```

这时候我们如果3，期待输出9，但实际输出的是15。

why?
这里我们就得找我们问题。

> 1. 首先推测问题出现的原因。初步确定问题可能的原因最好。
> 2. 实际上手调试很有必要。
> 3. 调试的时候我们心里有数。

5.2 实例二

```
#include <stdio.h>
int main()
{
    int i = 0;
    int arr[10] = {0};
    for(i=0; i<=12; i++)
    {
        arr[i] = 0;
        printf("hehe\n");
    }
    return 0;
}
```

研究程序死循环的原因。
注:
带学生看【nice公司的笔试题中的有关的题目】，讲解题目的重要性。

演示调试

6. 如何写出好（易于调试）的代码。

6.1 优秀的代码：

> 1. 代码运行正常
> 2. bug很少
> 3. 效率高
> 4. 可读性高
> 5. 可维护性高
> 6. 注释清晰
> 7. 文档齐全

常见的coding技巧：

> 1. 使用assert
> 2. 尽量使用const
> 3. 养成良好的编码风格
> 4. 添加必要的注释
> 5. 避免编码的陷阱。

const 可以对指针进行相关的修饰

6.2 示范：

> 模拟实现库函数：strcpy

```
/***
*char *strcpy(dst, src) - copy one string over another
*
*Purpose:
*       Copies the string src into the spot specified by
*       dest; assumes enough room.
*
*Entry:
*       char * dst - string over which "src" is to be copied
*       const char * src - string to be copied over "dst"
*
*Exit:
*       The address of "dst"
*
*Exceptions:
*******************************************************************************/

char * strcpy(char * dst, const char * src)
{
        char * cp = dst;
        assert(dst && src);

        while( *cp++ = *src++ )
                ;       /* Copy src over
        return( dst );
}
```

这是赋值语句, 其结果是字符的ASCII值, 当到了最后, \0的ASCII值为0

注意：

> 1. 分析参数的设计（命名，类型），返回值类型的设计
> 2. 这里讲解野指针，空指针的危害。
> 3. assert的使用，这里介绍assert的作用
> 4. 参数部分 const 的使用，这里讲解const修饰指针的作用

> 5. 注释的添加

6.3 const的作用

```
#include <stdio.h>
//代码1
void test1()
{
    int n = 10;
    int m = 20;
    int *p = &n;
    *p = 20;//ok?
    p = &m; //ok?
}
void test2()
{
     //代码2
    int n = 10;
    int m = 20;
    const int* p = &n;
    *p = 20;//ok?
    p = &m; //ok?
}
void test3()
{
    int n = 10;
    int m = 20;
    int *const p = &n;
    *p = 20; //ok?
    p = &m;  //ok?
}
int main()
{
    //测试无cosnt的
    test1();
    //测试const放在*的左边
    test2();
    //测试const放在*的右边
    test3();
    return 0;
}
```

结论：

const修饰指针变量的时候：

> 1. const如果放在*的左边，修饰的是指针指向的内容，保证指针指向的内容不能通过指针来改变。但是指针变量本身的内容可变。
> 2. const如果放在*的右边，修饰的是指针变量本身，保证了指针变量的内容不能修改，但是指针指向的内容，可以通过指针改变。

注:
介绍《高质量C/C++编程》一书中最后章节试卷中有关 `strcpy` 模拟实现的题目。

练习：

> 模拟实现一个strlen函数
> 参考代码：

```c
#include <stdio.h>
int my_strlen(const char *str)
{
    int count = 0;
    assert(str != NULL);
    while(*str)//判断字符串是否结束
    {
        count++;
        str++;
    }
    return count;
}
int main()
{
    const char* p = "abcdef";
    //测试
    int len = my_strlen(p);
    printf("len = %d\n", len);
    return 0;
}
```

7. 编程常见的错误

7.1 编译型错误

语法错误

> 直接看错误提示信息（双击），解决问题。或者凭借经验就可以搞定。相对来说简单。

7.2 链接型错误

标识符错误

> 看错误提示信息，主要在代码中找到错误信息中的标识符，然后定位问题所在。一般是**标识符名不存在**或者**拼写错误**。

7.3 运行时错误

> 借助调试，逐步定位问题。最难搞。

温馨提示：

> 做一个有心人，积累排错经验。

讲解重点：
介绍每种错误怎么产生，出现之后如何解决。

本章完。


比特科技

Git和Github的基本用法

背景

git是一个版本控制工具. 主要解决三个问题

1. 代码被喵星人吃掉了.
2. 产品经理反复修改需求, 需要同时维护多个版本代码.
3. 多人协同开发.

Github 是 "全球最大的同性社交网站". 通过 git 可以把代码上传到 Github 上给全球的用户分享.

下载安装

安装 git for windows

这个是一个git的windows系统的命令行版本

https://git-scm.com/downloads

或者

https://pan.baidu.com/s/1kU5OCOB#list/path=%2Fpub%2Fgit

安装 tortoise git

这个是git的图形界面.

https://tortoisegit.org/download/

注意:

1. 先安装 git for windows, 再安装 tortoise git
2. 安装 git for windows 一路 next 即可.
3. 安装 tortoise git 中需要配置 git.exe, 这个是 git for windows 包含的部分. 如果 git for windows 安装成功, 这一步使用默认结果即可.
4. 安装 tortoise git 还需要配置姓名和邮箱, 这个尽量和 Github 的邮箱填成一致.
5. 两个工具安装完毕后, 需要重启电脑才能正确使用.

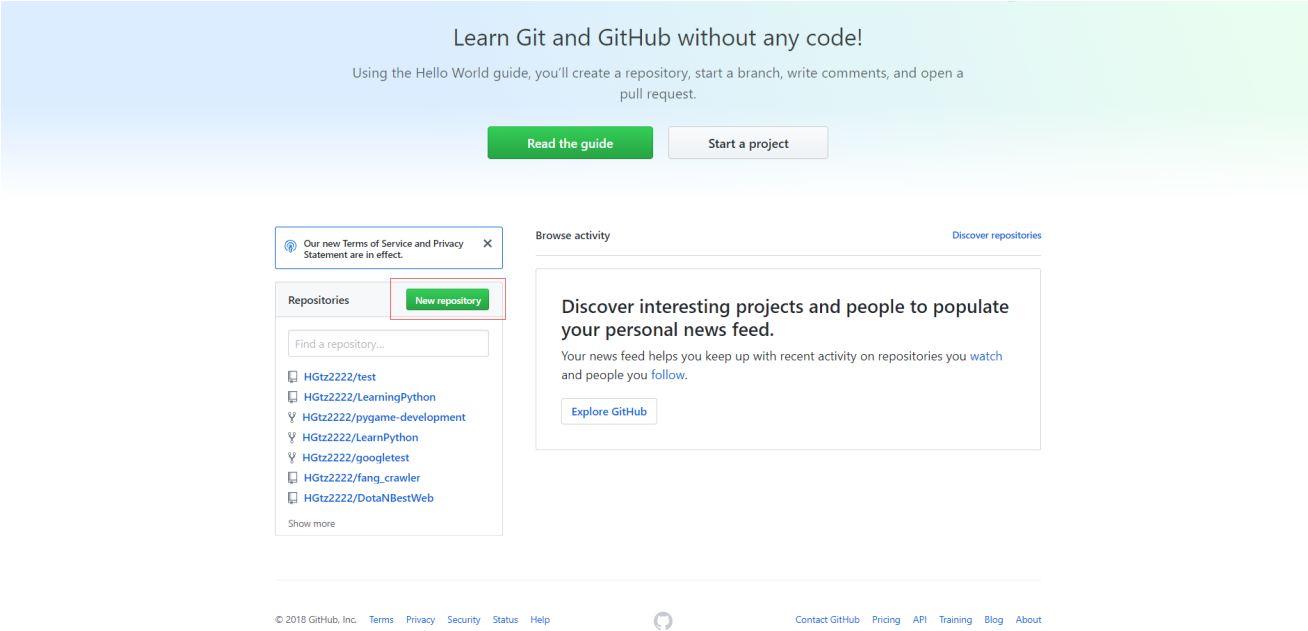
使用 Github 创建项目

注册账号

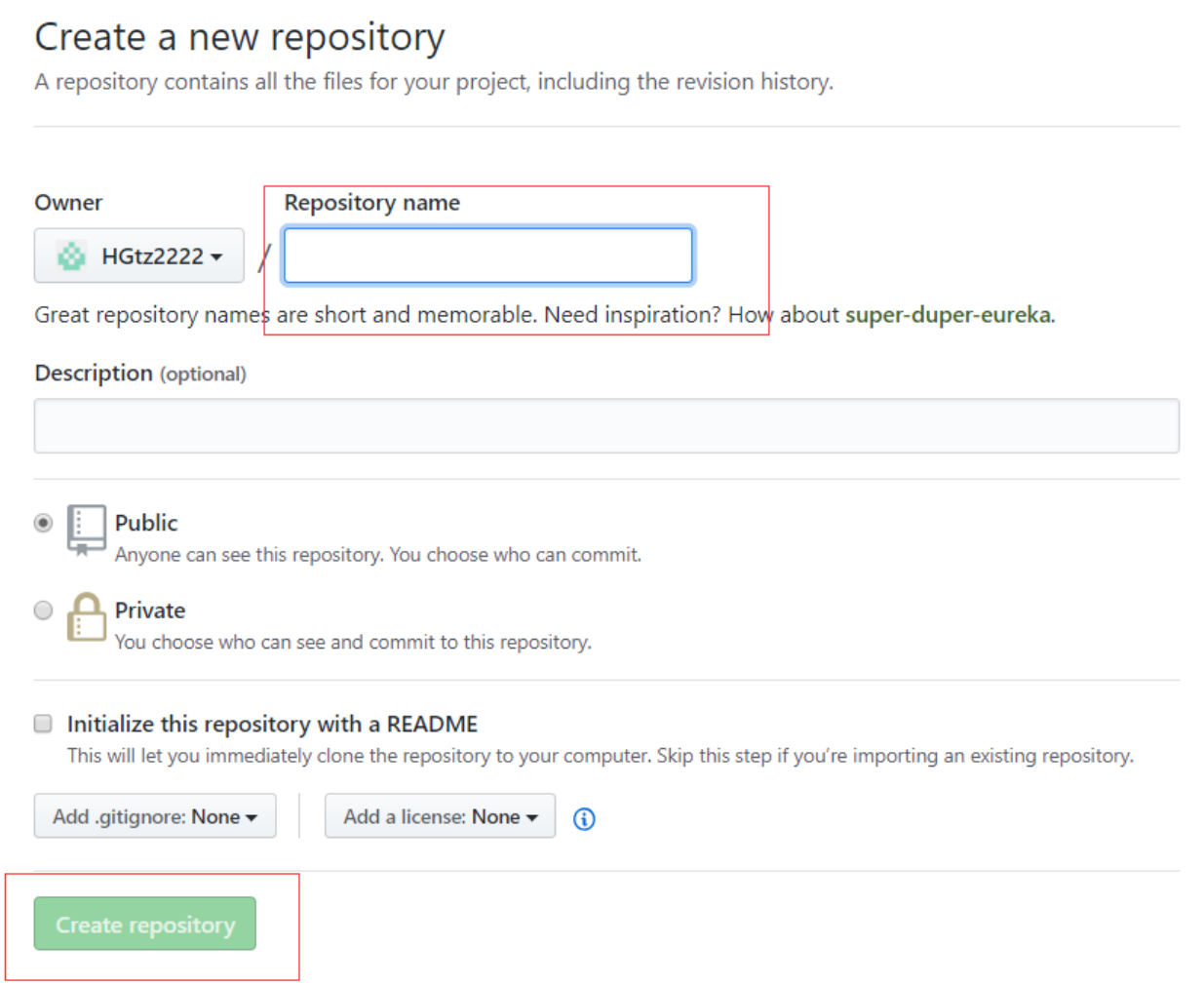
这个比较简单, 参考着官网提示即可. 需要进行邮箱校验.

创建项目

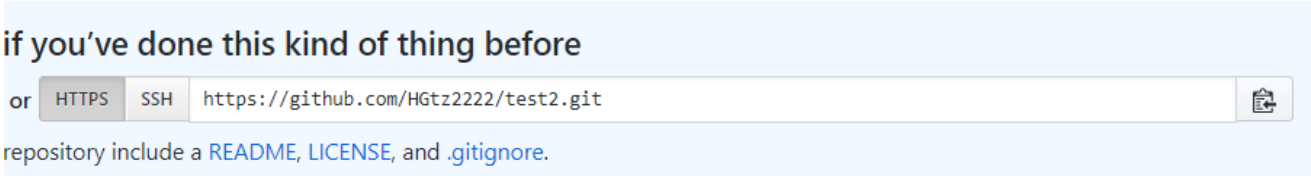
1. 登陆成功后, 进入个人主页, 点击左下方的 New repository 按钮新建项目



2. 然后跳转到的新页面中输入项目名称(注意, 名称不能重复, 系统会自动校验. 校验过程可能会花费几秒钟). 校验完毕后, 点击下方的 Create repository 按钮确认创建.

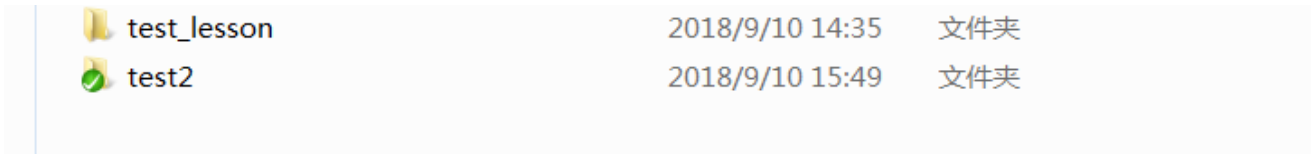


3. 在创建好的项目页面中复制项目的链接, 以备接下来进行下载.



下载项目到本地

1. 复制刚才创建好的项目的链接.
2. 打开指定的需要放置项目的目录
3. 右击目录, 点击 Git Clone
4. 在弹出的对话框中输入刚才复制的项目链接即可.



下载成功, 会出现 绿色 图标.

Git 操作的三板斧

放入代码

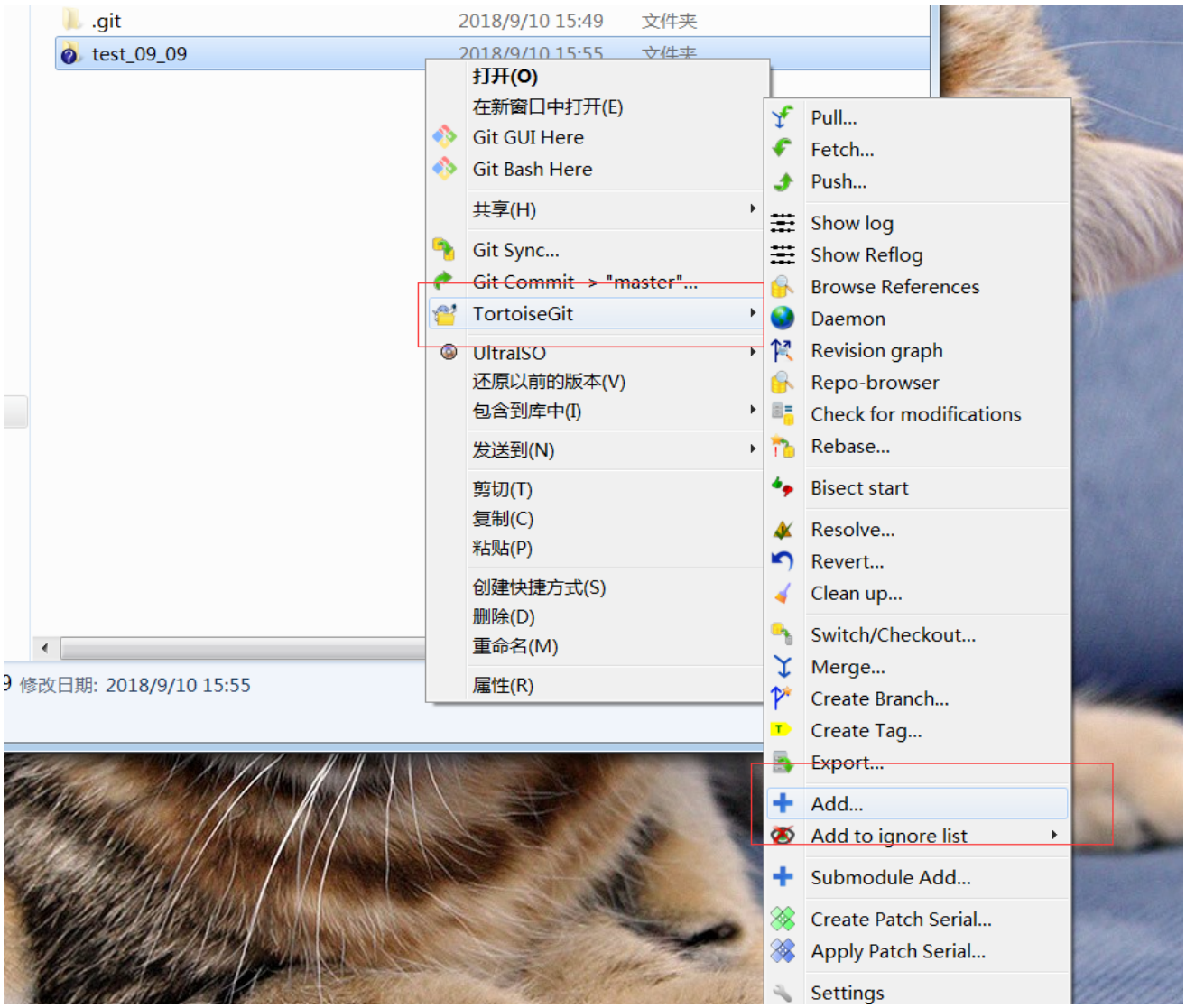
使用 VS 创建工程, 并把工程放在刚才下载到本地的项目路径中.

或者将曾经写过的代码的工程目录直接拷贝到项目目录中.

三板斧第一招: git add

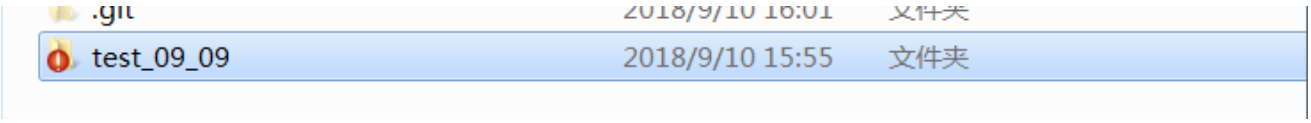
告知 git 工具哪些文件需要进行版本管理

此时右击标记为 蓝色 ? (表示该文件未使用 git 管理) 的目录, 选择 add



弹出的对话框中勾选具体需要管理的文件. 勾选完毕点击 ok 即可.

此时图标变成红色感叹号(表示该文件被git管理, 但是未提交内容)



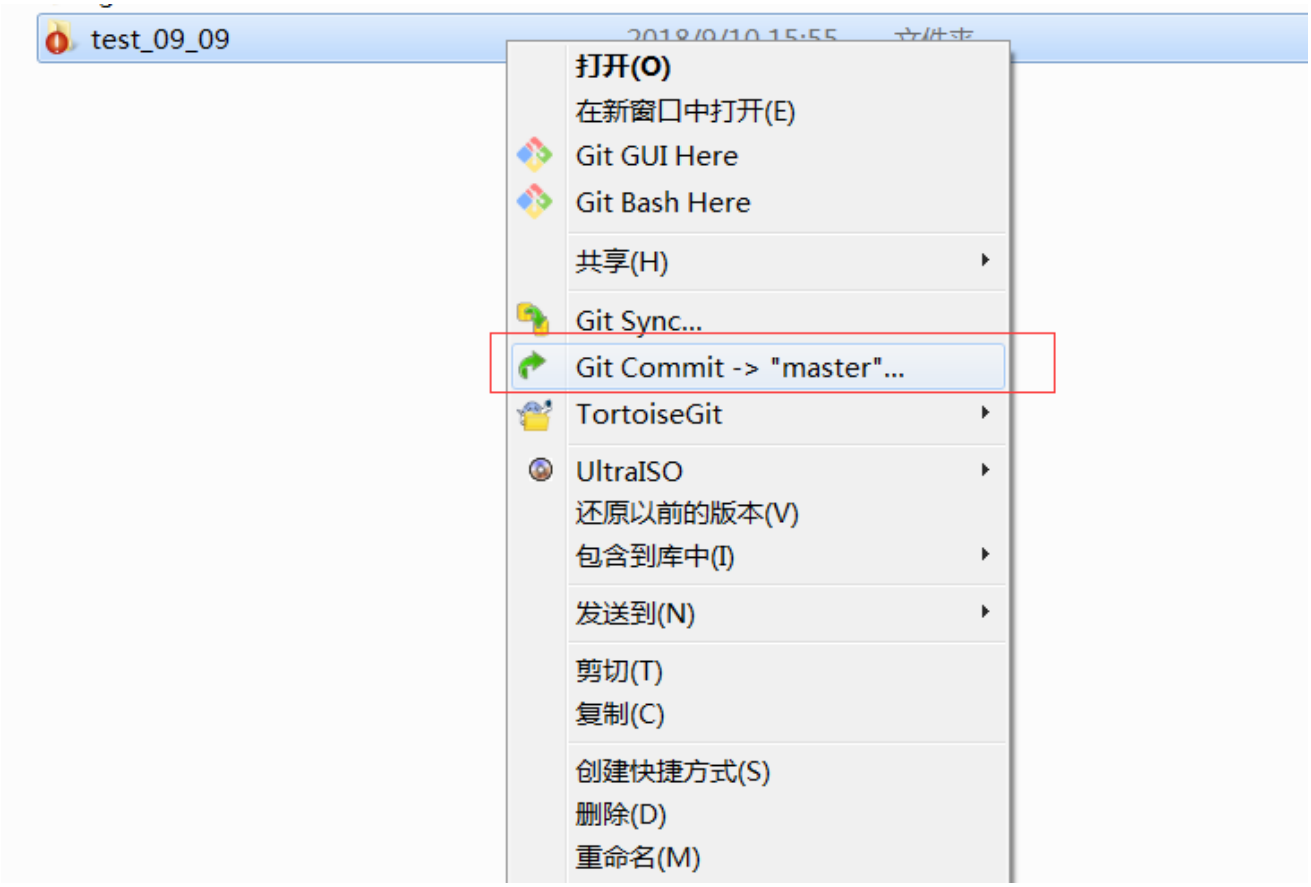
三板斧第二招: git commit

将修改内容提交到本地

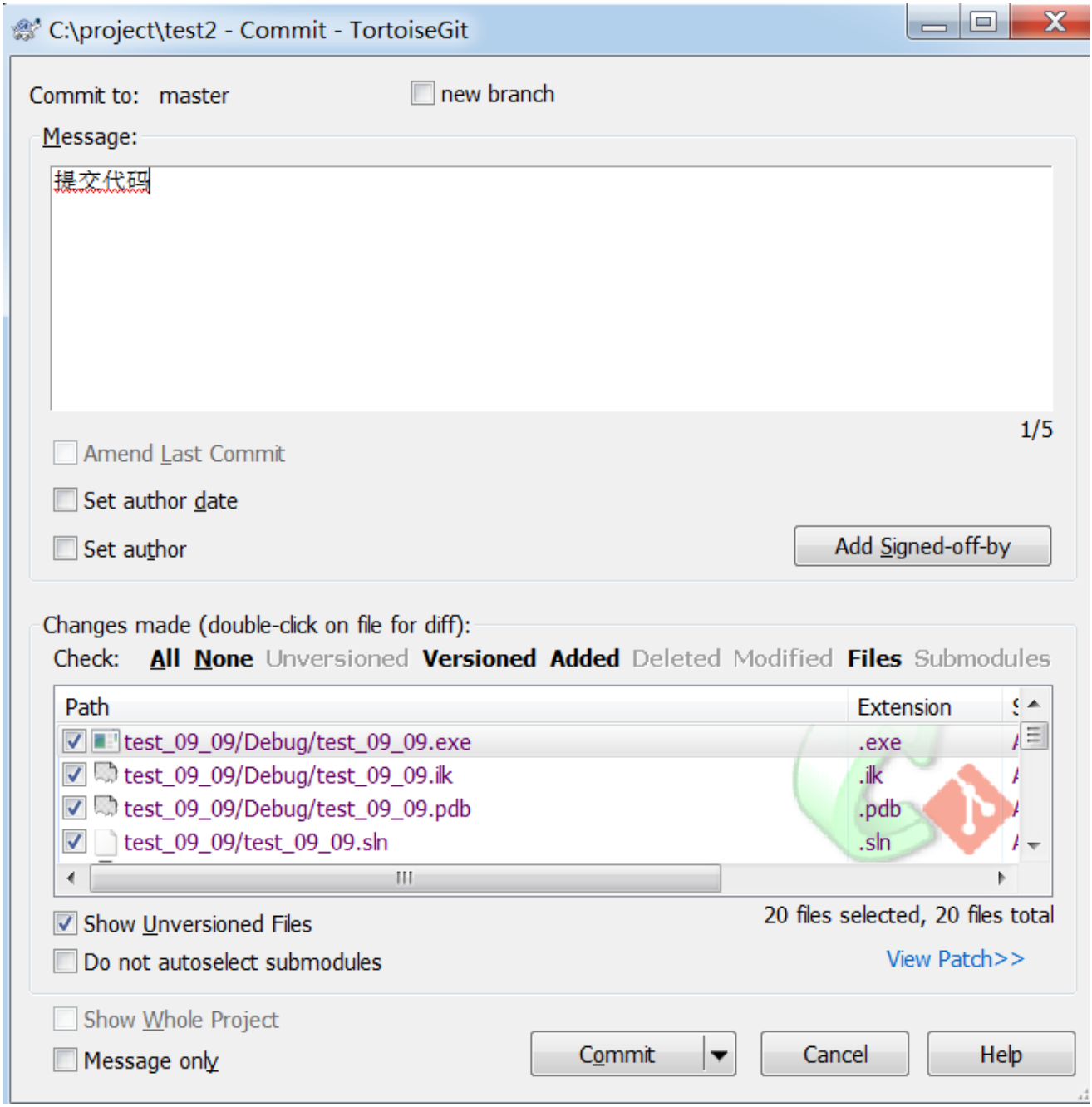
每提交一次, 就是一个版本. 比如开发完某个功能模块, 就可以提交一次了. 后续进行版本回退都是以提交为准.

注意: 此时只是提交到本地, Github 上还看不到代码变更.

右键选择 红色感叹号 目录, 选择 Git commit -> master



此时弹出了一个对话框. 可以在此处看到都需要提交哪些文件, 以及每个文件的具体改动情况. 并且需要输入提交日志. 描述这次提交的具体改动原因是什么. 这个日志是后续进行版本回退的重要参考依据.

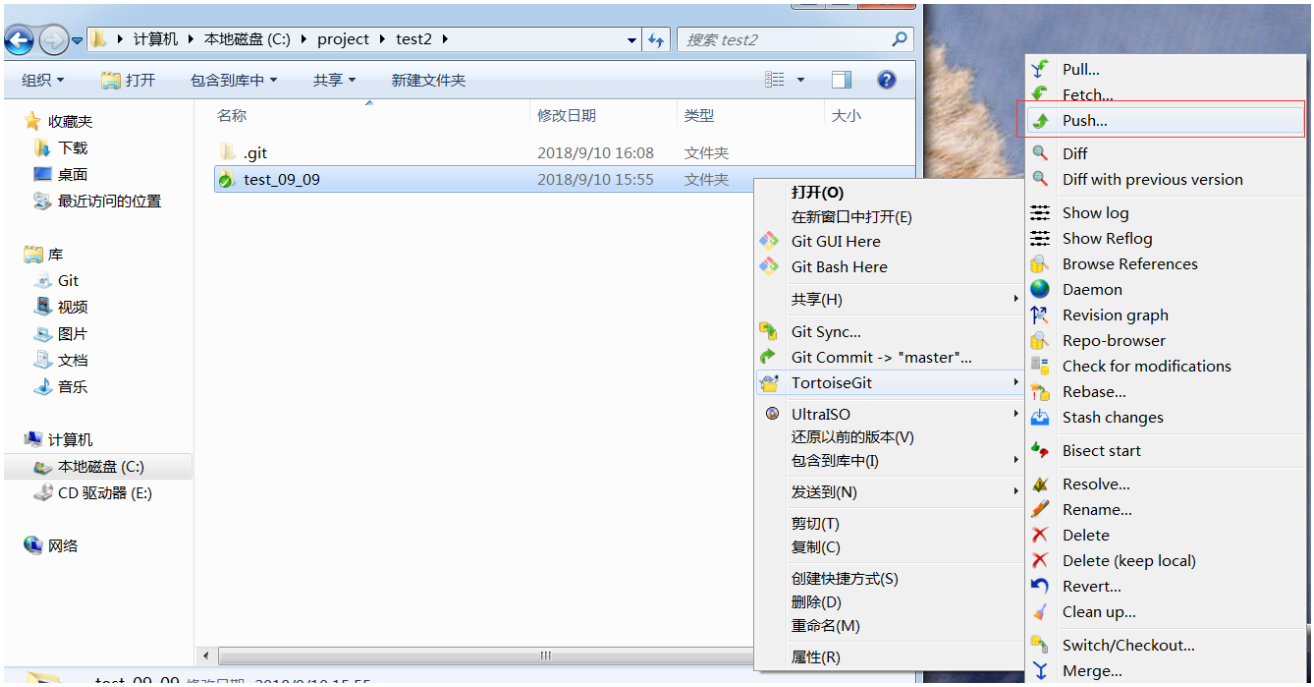


点击下方的 Commit 按钮完成提交.

三板斧第三招: git push

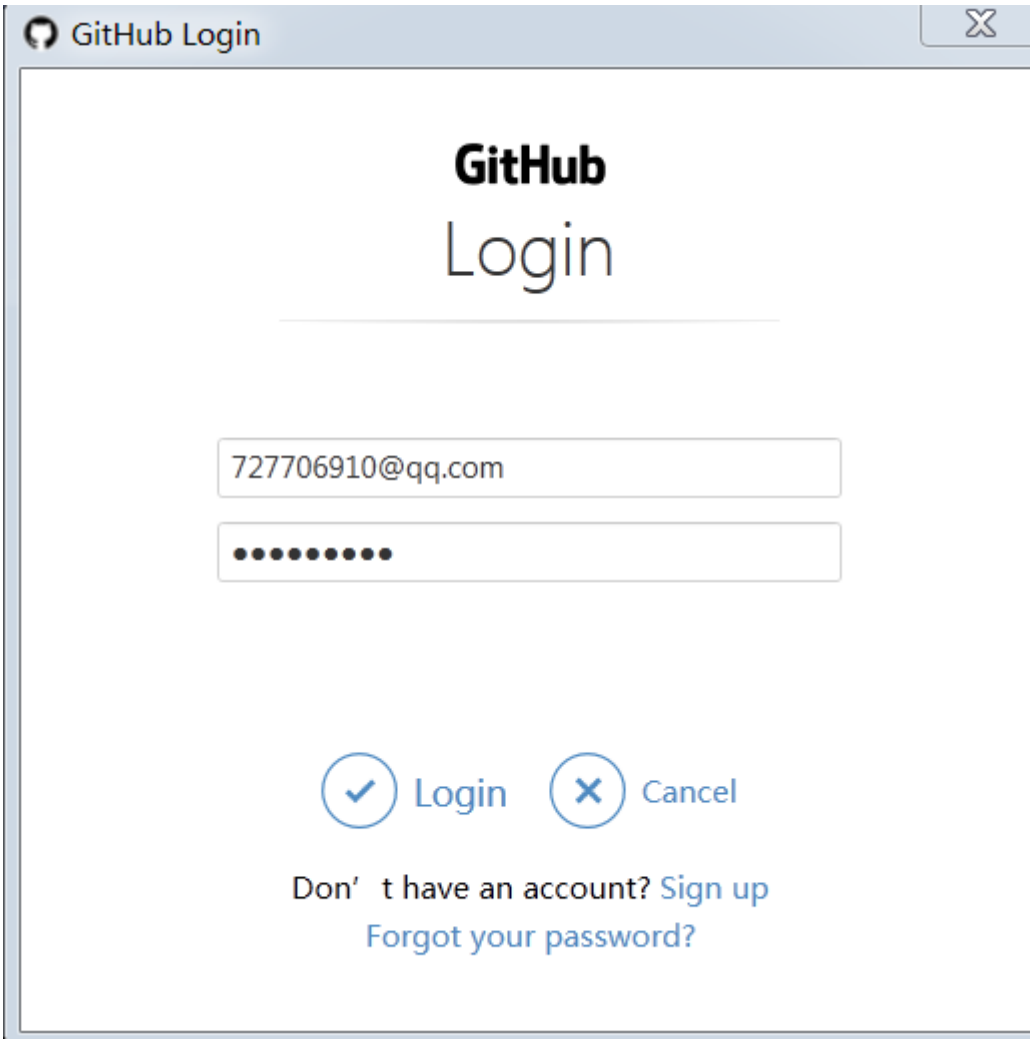
提交的内容需要同步到服务器上, 才能让其他人看到改动. 使用 push 即可.

右键需要 push 的目录, 点击 push



弹出的对话框确认 push. 不需要修改, 直接确认即可.

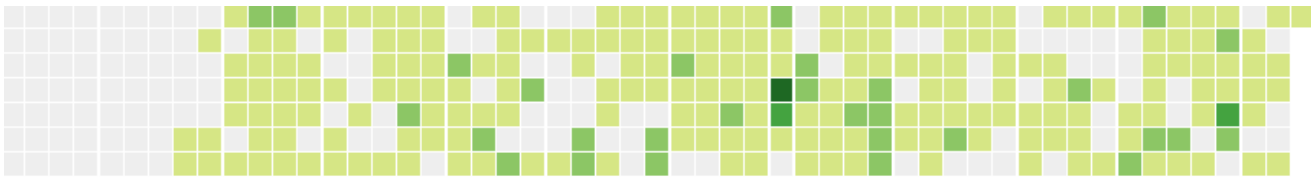
然后会弹出对话框提示输入 Github 的账户和密码.



用户名密码输入正确, 点击 Login , 即可完成 push. 此时刷新 Github 的界面, 就能看到新版本的代码了.

小结

掌握了以上 Git & Github 基本操作, 希望同学们都能够把 Github 用起来, 记录自己的学习过程, 打造自己的专属名片. 如果自己的 Github 日历能够一片绿色, 这是向面试官证明自己靠谱的最有力的证据.



BIT-1-深度剖析数据在内存中的存储

本章重点

1. 数据类型详细介绍
2. 整形在内存中的存储：原码、反码、补码
3. 大小端字节序介绍及判断
4. 浮点型在内存中的存储解析

正文开始©比特就业课

1. 数据类型介绍

前面我们已经学习了基本的内置类型：

```
char        //字符数据类型
short       //短整型
int         //整形
long        //长整型
long long   //更长的整形
float       //单精度浮点数
double      //双精度浮点数
//C语言有没有字符串类型？
```

以及他们所占存储空间的大小。
类型的意义：

1. 使用这个类型开辟内存空间的大小（大小决定了使用范围）。
2. 如何看待内存空间的视角。

1.1 类型的基本归类：

整形家族：

```
char
    unsigned char
    signed char
short
    unsigned short [int]
    signed short [int]
int
    unsigned int
    signed int
long
    unsigned long [int]
    signed long [int]
```

浮点数家族：

```
float
double
```

构造类型：

```
> 数组类型
> 结构体类型 struct
> 枚举类型 enum
> 联合类型 union
```

指针类型

```
int *pi;
char *pc;
float* pf;
void* pv;
```

空类型：

> void 表示空类型（无类型）
>
> 通常应用于函数的返回类型、函数的参数、指针类型。

2. 整形在内存中的存储

我们之前讲过一个变量的创建是要在内存中开辟空间的。空间的大小是根据不同的类型而决定的。

> 那接下来我们谈谈数据在所开辟内存中到底是如何存储的?

比如：

```
int a = 20;
int b = -10;
```

我们知道为 a 分配四个字节的空间。
那如何存储?

下来了解下面的概念：

2.1 原码、反码、补码

计算机中的整数有三种表示方法，即原码、反码和补码。

三种表示方法均有**符号位**和**数值位**两部分，符号位都是用0表示“正”，用1表示“负”，而数值位

负整数的三种表示方法各不相同。

> **原码**
>
> 直接将二进制按照正负数的形式翻译成二进制就可以。

> **反码**
>
> 将原码的符号位不变，其他位依次按位取反就可以得到了。

> **补码**

> 反码+1就得到补码。

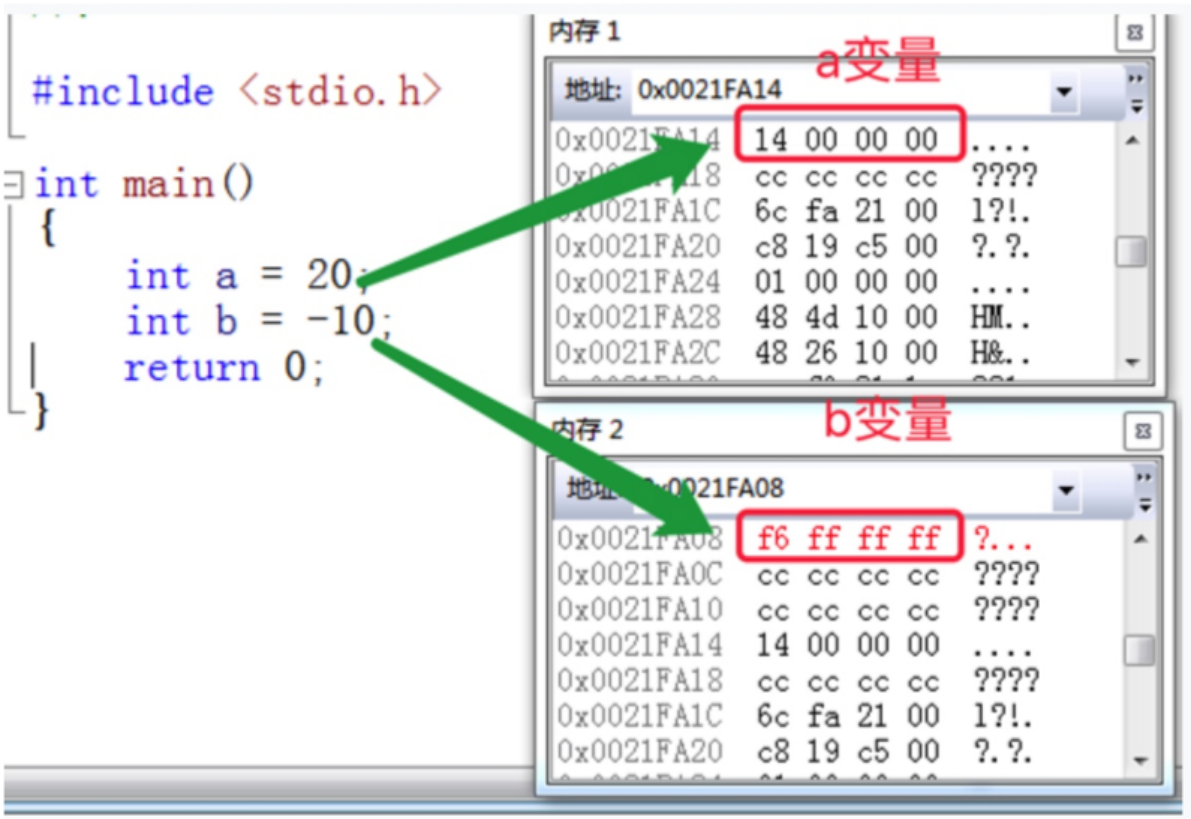
正数的原、反、补码都相同。

对于整形来说：数据存放内存中其实存放的是补码。

为什么呢?

> 在计算机系统中，数值一律用补码来表示和存储。原因在于，使用补码，可以将符号位和数值域统一处理；
> 同时，加法和减法也可以统一处理（**CPU只有加法器**）此外，补码与原码相互转换，其运算过程是相同的，不需要额外的硬件电路。

我们看看在内存中的存储：



我们可以看到对于a和b分别存储的是补码。但是我们发现顺序有点**不对劲**。
这是又为什么?

2.2 大小端介绍

什么大端小端：

> 大端（存储）模式，是指数据的低位保存在内存的高地址中，而数据的高位，保存在内存的低地址中；
>
> 小端（存储）模式，是指数据的低位保存在内存的低地址中，而数据的高位，，保存在内存的高地址中。

为什么有大端和小端：

> 为什么会有大小端模式之分呢？这是因为在计算机系统中，我们是以字节为单位的，每个地址单元都对应着一个字节，一个字节为8

> bit。但是在C语言中除了8 bit的char之外，还有16 bit的short型，32 bit的long型（要看具体的编译器），另外，对于位数大于8位
>
> 的处理器，例如16位或者32位的处理器，由于寄存器宽度大于一个字节，那么必然存在着一个如何将多个字节安排的问题。因此就
>
> 导致了大端存储模式和小端存储模式。
>
> 例如：一个`16bit`的`short`型`x`，在内存中的地址为`0x0010`，`x`的值为`0x1122`，那么`0x11`为高字节，`0x22`为低字节。对于大端
>
> 模式，就将`0x11`放在低地址中，即`0x0010`中，`0x22`放在高地址中，即`0x0011`中。小端模式，刚好相反。我们常用的`X86`结构是
>
> 小端模式，而`KEIL C51`则为大端模式。很多的ARM，DSP都为小端模式。有些ARM处理器还可以由硬件来选择是大端模式还是小端
>
> 模式。

百度2015年系统工程师笔试题：

> 请简述大端字节序和小端字节序的概念，设计一个小程序来判断当前机器的字节序。（10分）

```
//代码1
#include <stdio.h>
int check_sys()
{
    int i = 1;
    return (*(char *)&i);
}
int main()
{
    int ret = check_sys();
    if(ret == 1)
    {
        printf("小端\n");
    }
    else
    {
        printf("大端\n");
    }
    return 0;
}

//代码2
int check_sys()
{
    union
    {
        int i;
        char c;
    }un;
    un.i = 1;
    return un.c;
}
```

2.3 练习

```
1.
//输出什么？
#include <stdio.h>
int main()
{
    char a= -1;
    signed char b=-1;
    unsigned char c=-1;
    printf("a=%d,b=%d,c=%d",a,b,c);
    return 0;
}
```

下面程序输出什么?

```
2.
#include <stdio.h>
int main()
{
    char a = -128;
    printf("%u\n",a);
    return 0;
}
```

```
3.
#include <stdio.h>
int main()
{
    char a = 128;
    printf("%u\n",a);
    return 0;
}
```

```
4.
int i= -20;
unsigned  int  j = 10;
printf("%d\n", i+j);
//按照补码的形式进行运算，最后格式化成为有符号整数
```

```
5.
unsigned int i;
for(i = 9; i >= 0; i--)
{
    printf("%u\n",i);
}
```

```
6.
int main()
{
    char a[1000];
    int i;
    for(i=0; i<1000; i++)
    {
        a[i] = -1-i;
    }
    printf("%d",strlen(a));
    return 0;
}
```

```
7.
#include <stdio.h>

unsigned char i = 0;
int main()
{
    for(i = 0;i<=255;i++)
    {
        printf("hello world\n");
    }
    return 0;
}
```

3. 浮点型在内存中的存储

常见的浮点数：

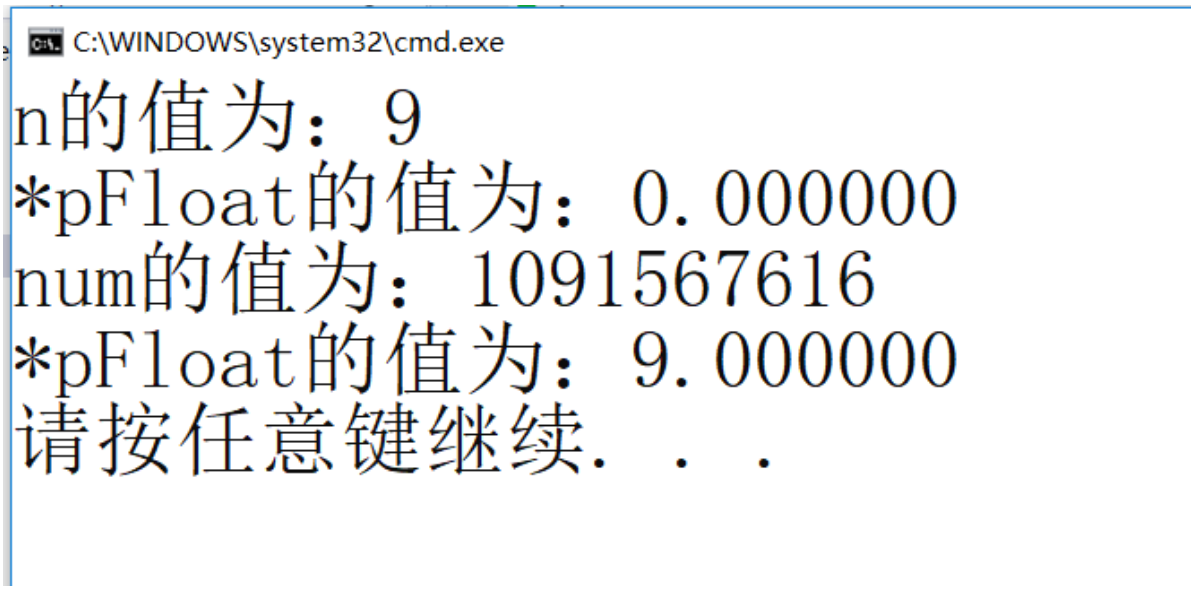
> 3.14159
>
> 1E10
>
> 浮点数家族包括：`float、double、long double`类型。
>
> 浮点数表示的范围：float.h中定义

3.1 一个例子

浮点数存储的例子：

```
int main()
{
    int n = 9;
    float *pFloat = (float *)&n;
    printf("n的值为：%d\n",n);
    printf("*pFloat的值为：%f\n",*pFloat);

    *pFloat = 9.0;
    printf("num的值为：%d\n",n);
    printf("*pFloat的值为：%f\n",*pFloat);
    return 0;
}
```

输出的结果是什么呢?



3.2 浮点数存储规则

`num`和`*pFloat`在内存中明明是同一个数，为什么浮点数和整数的解读结果会差别这么大?

要理解这个结果，一定要搞懂浮点数在计算机内部的表示方法。

详细解读:

根据国际标准IEEE（电气和电子工程协会） 754，任意一个二进制浮点数V可以表示成下面的形式:

> - (-1)^S * M * 2^E
> - (-1)^s表示符号位，当s=0，V为正数；当s=1，V为负数。
> - M表示有效数字，大于等于1，小于2。
> - 2^E表示指数位。

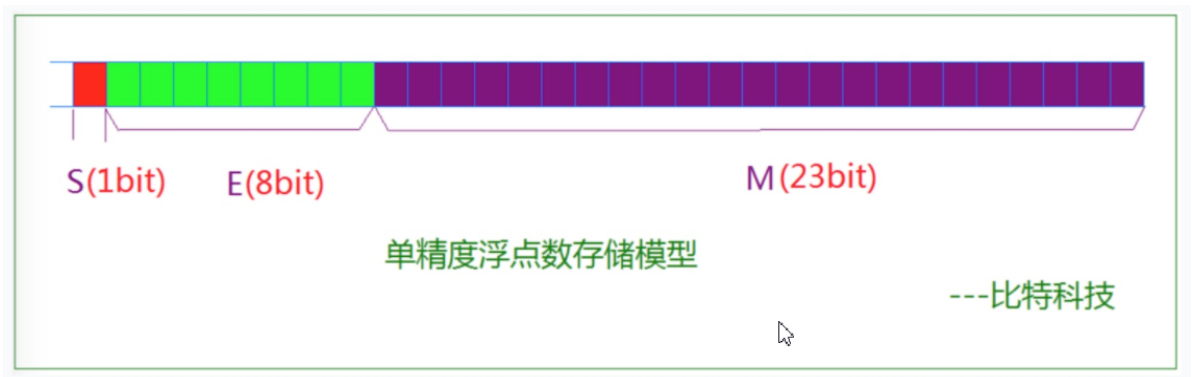
举例来说:

十进制的5.0，写成二进制是`101.0`，相当于`1.01×2^2`。

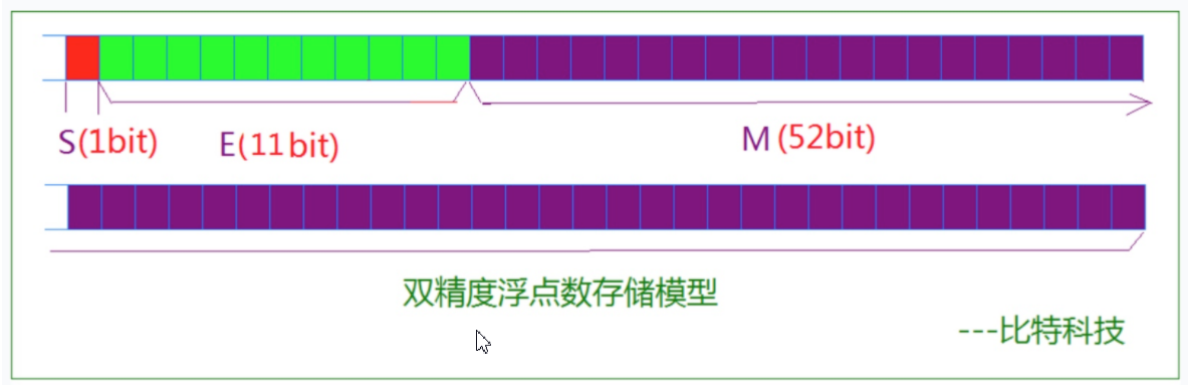
那么，按照上面V的格式，可以得出s=0，M=1.01，E=2。

十进制的-5.0，写成二进制是`-101.0`，相当于`-1.01×2^2`。那么，s=1，M=1.01，E=2。

IEEE 754规定:
对于32位的浮点数，最高的1位是符号位s，接着的8位是指数E，剩下的23位为有效数字M。



对于64位的浮点数，最高的1位是符号位S，接着的11位是指数E，剩下的52位为有效数字M。



IEEE 754对有效数字M和指数E，还有一些特别规定。
前面说过，`1≤M<2`，也就是说，M可以写成`1.xxxxxx`的形式，其中xxxxxx表示小数部分。

IEEE 754规定，在计算机内部保存M时，默认这个数的第一位总是1，因此可以被舍去，只保存后面的xxxxxx部分。比如保存1.01的时

候，只保存01，等到读取的时候，再把第一位的1加上去。这样做的目的，是节省1位有效数字。以32位浮点数为例，留给M只有23位，

将第一位的1舍去以后，等于可以保存24位有效数字。

至于指数E，情况就比较复杂。

首先，E为一个无符号整数（unsigned int）
这意味着，如果E为8位，它的取值范围为0~255；如果E为11位，它的取值范围为0~2047。但是，我们知道，科学计数法中的E是可以出

现负数的，所以IEEE 754规定，存入内存时E的真实值必须再加上一个中间数，对于8位的E，这个中间数是127；对于11位的E，这个中间

数是1023。比如，2^10的E是10，所以保存成32位浮点数时，必须保存成10+127=137，即10001001。

然后，指数E从内存中取出还可以再分成三种情况：

E不全为0或不全为1

> 这时，浮点数就采用下面的规则表示，即指数E的计算值减去127（或1023），得到真实值，再将有效数字M前加上第一位的1。
>
> 比如：
>
> 0.5（1/2）的二进制形式为0.1，由于规定正数部分必须为1，即将小数点右移1位，则为1.0*2^(-1)，其阶码为-1+127=126，表示为
>
> 01111110，而尾数1.0去掉整数部分为0，补齐0到23位00000000000000000000000，则其二进制表示形式为:

```
0 01111110 00000000000000000000000
```

E全为0

> 这时，浮点数的指数E等于1-127（或者1-1023）即为真实值，
>
> 有效数字M不再加上第一位的1，而是还原为0.xxxxxx的小数。这样做是为了表示±0，以及接近于0的很小的数字。

E全为1

> 这时，如果有效数字M全为0，表示±无穷大（正负取决于符号位s）；

好了，关于浮点数的表示规则，就说到这里。

解释前面的题目：

下面，让我们回到一开始的问题：为什么`0x00000009`还原成浮点数，就成了`0.000000`？

首先，将`0x00000009`拆分，得到第一位符号位s=0，后面8位的指数`E=00000000`，最后23位的有效数字M=000 0000 0000 0000 0000

1001。

```
9 -> 0000 0000 0000 0000 0000 0000 0000 1001
```

由于指数E全为0，所以符合上一节的第二种情况。因此，浮点数V就写成：

`V=(-1)^0 × 0.00000000000000000001001×2^(-126)=1.001×2^(-146)`

显然，V是一个很小的接近于0的正数，所以用十进制小数表示就是0.000000。

再看例题的第二部分。

请问浮点数9.0，如何用二进制表示？还原成十进制又是多少？

首先，浮点数9.0等于二进制的1001.0，即1.001×2^3。

```
9.0 -> 1001.0 ->(-1)^01.0012^3 -> s=0, M=1.001,E=3+127=130
```

那么，第一位的符号位s=0，有效数字M等于001后面再加20个0，凑满23位，指数E等于3+127=130，即10000010。

所以，写成二进制形式，应该是s+E+M，即

```
0 10000010 001 0000 0000 0000 0000 0000
```

这个32位的二进制数，还原成十进制，正是`1091567616`。

本章完。

BIT-2-指针的进阶

本章重点

1. 字符指针
2. 数组指针
3. 指针数组
4. 数组传参和指针传参
5. 函数指针
6. 函数指针数组
7. 指向函数指针数组的指针
8. 回调函数
9. 指针和数组面试题的解析

正文开始©比特就业课

> 指针的主题，我们在初级阶段的《指针》章节已经接触过了，我们知道了指针的概念：
>
> 1. 指针就是个变量，用来存放地址，地址唯一标识一块内存空间。
> 2. 指针的大小是固定的4/8个字节（32位平台/64位平台）。
> 3. 指针是有类型，指针的类型决定了指针的+-整数的步长，指针解引用操作的时候的权限。
> 4. 指针的运算。

这个章节，我们继续探讨指针的高级主题。

1. 字符指针

在指针的类型中我们知道有一种指针类型为字符指针`char*`;

一般使用:

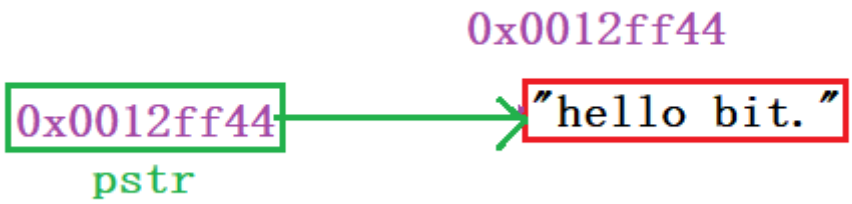
```
int main()
{
    char ch = 'w';
    char *pc = &ch;
    *pc = 'w';
    return 0;
}
```

还有一种使用方式如下:

```
int main()
{
    const char* pstr = "hello bit.";//这里是把一个字符串放到pstr指针变量里了吗?
    printf("%s\n", pstr);
    return 0;
}
```

代码`const char* pstr = "hello bit.";`

特别容易让同学以为是把字符串`hello bit`放到字符指针`pstr`里了，但是/本质是把字符串`hello bit.`首字符的地址放到了pstr中。

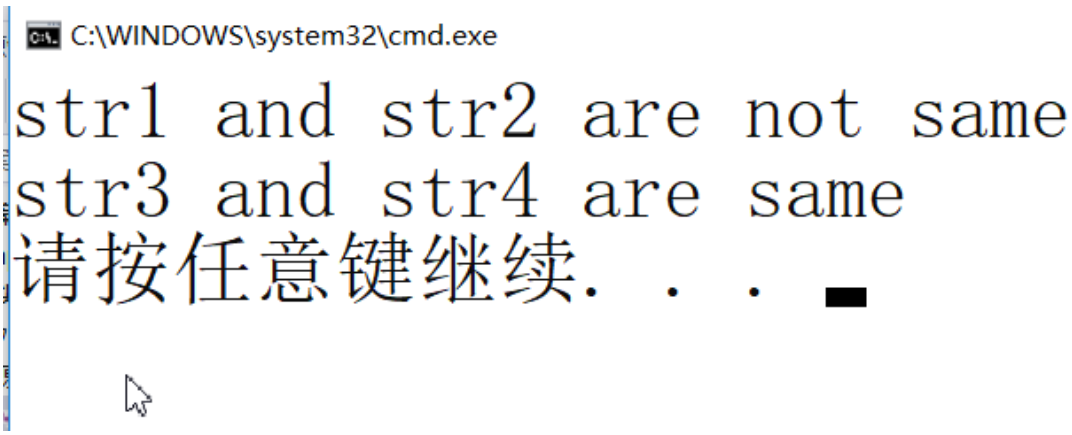


上面代码的意思是把一个常量字符串的首字符`h`的地址存放到指针变量`pstr`中。

那就有可这样的面试题：

```c
#include <stdio.h>

int main()
{
    char str1[] = "hello bit.";
    char str2[] = "hello bit.";
    const char *str3 = "hello bit.";
    const char *str4 = "hello bit.";

    if(str1 ==str2)
        printf("str1 and str2 are same\n");
    else
        printf("str1 and str2 are not same\n");

    if(str3 ==str4)
        printf("str3 and str4 are same\n");
    else
        printf("str3 and str4 are not same\n");

    return 0;
}
```

这里最终输出的是：



这里str3和str4指向的是一个同一个常量字符串。C/C++会把常量字符串存储到单独的一个内存区域，当几个指针。指向同一个字符串的时候，他们实际会指向同一块内存。但是用相同的常量字符串去初始化不同的数组的时候就会开辟出不同的内存块。所以str1和str2不同，str3和str4不同。

2. 指针数组

在《指针》章节我们也学了指针数组，指针数组是一个存放指针的数组。

这里我们再复习一下，下面指针数组是什么意思？

```
int* arr1[10]; //整形指针的数组
char *arr2[4]; //一级字符指针的数组
char **arr3[5];//二级字符指针的数组
```

3. 数组指针

3.1 数组指针的定义

数组指针是指针？还是数组？

答案是：指针。

我们已经熟悉：

整形指针：`int * pint;`能够指向整形数据的指针。

浮点型指针：`float * pf;`能够指向浮点型数据的指针。

那数组指针应该是：能够指向数组的指针。

下面代码哪个是数组指针？

```
int *p1[10];
int (*p2)[10];
//p1, p2分别是什么？
```

解释：

```
int (*p)[10];
//解释：p先和*结合，说明p是一个指针变量，然后指着指向的是一个大小为10个整型的数组。所以p是一个指针，指向一个数组，叫数组指针。

//这里要注意：[]的优先级要高于*号的，所以必须加上（）来保证p先和*结合。
```

3.2 &数组名VS数组名

对于下面的数组：

```
int arr[10];
```

`arr`和`&arr`分别是啥？

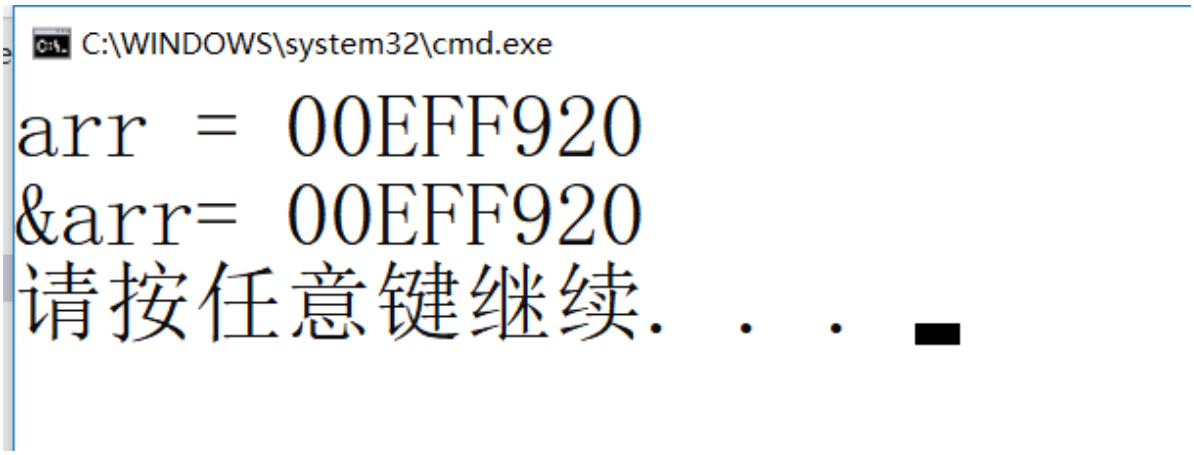
我们知道arr是数组名，数组名表示数组首元素的地址。

那&arr数组名到底是啥？

我们看一段代码：

```c
#include <stdio.h>
int main()
{
    int arr[10] = {0};
    printf("%p\n", arr);
    printf("%p\n", &arr);
    return 0;
}
```

运行结果如下：

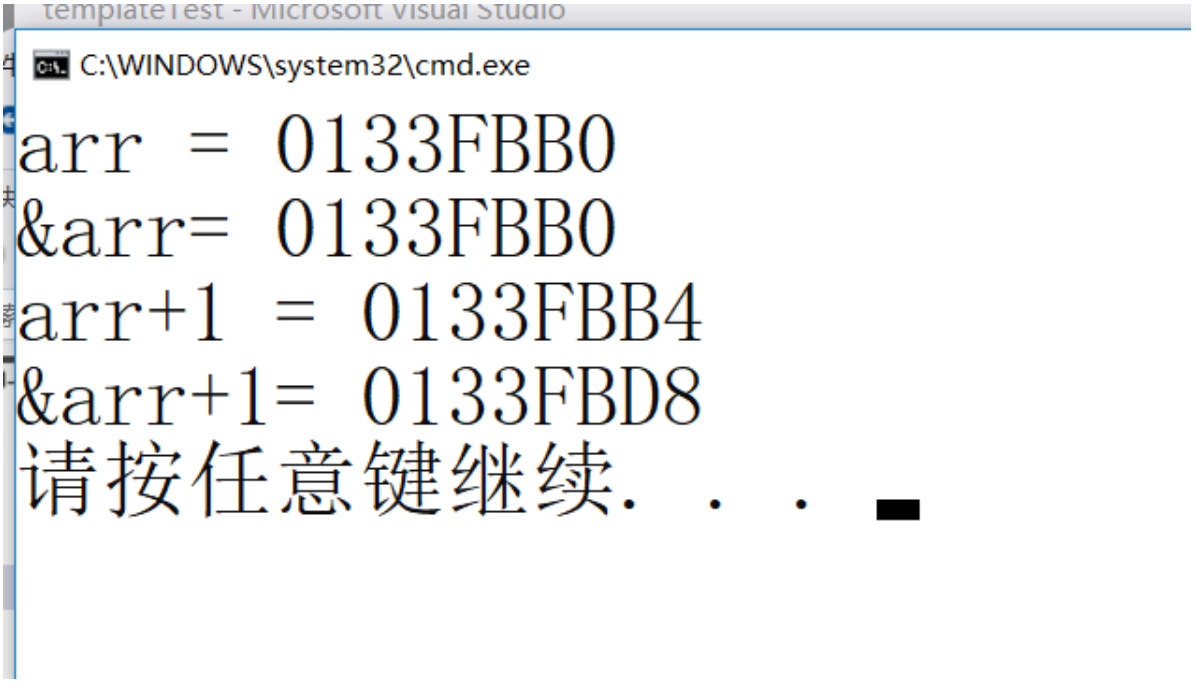


可见数组名和&数组名打印的地址是一样的。

难道两个是一样的吗？

我们再看一段代码：

```c
#include <stdio.h>
int main()
{
    int arr[10] = { 0 };
    printf("arr = %p\n", arr);
    printf("&arr= %p\n", &arr);

    printf("arr+1 = %p\n", arr+1);
    printf("&arr+1= %p\n", &arr+1);
    return 0;
}
```



根据上面的代码我们发现，其实&arr和arr，虽然值是一样的，但是意义应该不一样的。

实际上：`&arr`表示的是**数组的地址**，而不是数组首元素的地址。（细细体会一下）

本例中`&arr`的类型是：`int(*)[10]`，是一种数组指针类型

数组的地址+1，跳过整个数组的大小，所以`&arr+1`相对于`&arr`的差值是40.

3.3 数组指针的使用

那数组指针是怎么使用的呢?

既然数组指针指向的是数组，那数组指针中存放的应该是数组的地址。

看代码：

```
#include <stdio.h>
int main()
{
    int arr[10] = {1,2,3,4,5,6,7,8,9,0};
    int (*p)[10] = &arr;//把数组arr的地址赋值给数组指针变量p
    //但是我们一般很少这样写代码
    return 0;
}
```

一个数组指针的使用：

```
#include <stdio.h>
void print_arr1(int arr[3][5], int row, int col)
{
    int i = 0;
    for(i=0; i<row; i++)
    {
        for(j=0; j<col; j++)
        {
            printf("%d ", arr[i][j]);
        }
```

```
        printf("\n");
    }
}
void print_arr2(int (*arr)[5], int row, int col)
{
    int i = 0;
    for(i=0; i<row; i++)
    {
        for(j=0; j<col; j++)
        {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }
}
int main()
{
    int arr[3][5] = {1,2,3,4,5,6,7,8,9,10};
    print_arr1(arr, 3, 5);
    //数组名arr，表示首元素的地址
    //但是二维数组的首元素是二维数组的第一行
    //所以这里传递的arr，其实相当于第一行的地址，是一维数组的地址
    //可以数组指针来接收
    print_arr2(arr, 3, 5);
    return 0;
}
```

学了指针数组和数组指针我们来一起回顾并看看下面代码的意思：

```
int arr[5];
int *parr1[10];
int (*parr2)[10];
int (*parr3[10])[5];
```

4. 数组参数、指针参数

在写代码的时候难免要把【数组】或者【指针】传给函数，那函数的参数该如何设计呢？

4.1 一维数组传参

```
#include <stdio.h>
void test(int arr[])//ok?
{}
void test(int arr[10])//ok?
{}
void test(int *arr)//ok?
{}
void test2(int *arr[20])//ok?
{}
void test2(int **arr)//ok?
{}
int main()
{
    int arr[10] = {0};
```

```
    int *arr2[20] = {0};
    test(arr);
    test2(arr2);
}
```

4.2 二维数组传参

```
void test(int arr[3][5])//ok?
{}
void test(int arr[][])//ok?
{}
void test(int arr[][5])//ok?
{}
//总结：二维数组传参，函数形参的设计只能省略第一个[]的数字。
//因为对一个二维数组，可以不知道有多少行，但是必须知道一行多少元素。
//这样才方便运算。

void test(int *arr)//ok?
{}
void test(int* arr[5])//ok?
{}
void test(int (*arr)[5])//ok?
{}
void test(int **arr)//ok?
{}
int main()
{
    int arr[3][5] = {0};
    test(arr);
}
```

4.3 一级指针传参

```
#include <stdio.h>
void print(int *p, int sz)
{
    int i = 0;
    for(i=0; i<sz; i++)
    {
        printf("%d\n", *(p+i));
    }
}
int main()
{
    int arr[10] = {1,2,3,4,5,6,7,8,9};
    int *p = arr;
    int sz = sizeof(arr)/sizeof(arr[0]);
    //一级指针p，传给函数
    print(p, sz);
    return 0;
}
```

思考：

> 当一个函数的参数部分为一级指针的时候，函数能接收什么参数？

比如：

```
void test1(int *p)
{}
//test1函数能接收什么参数？
void test2(char* p)
{}
//test2函数能接收什么参数？
```

4.4 二级指针传参

```
#include <stdio.h>
void test(int** ptr)
{
    printf("num = %d\n", **ptr);
}
int main()
{
    int n = 10;
    int*p = &n;
    int **pp = &p;
    test(pp);
    test(&p);
    return 0;
}
```

思考：

> 当函数的参数为二级指针的时候，可以接收什么参数？

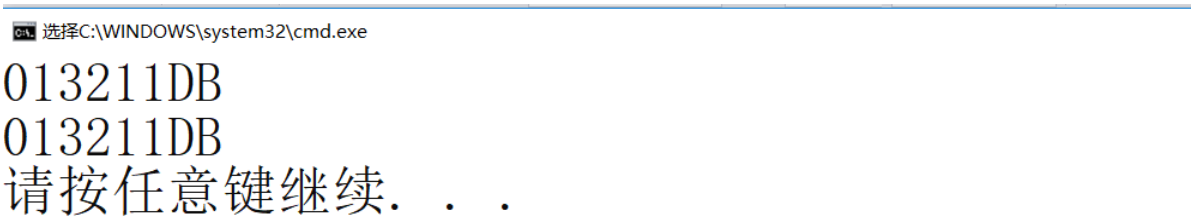
```
void test(char **p)
{

}
int main()
{
    char c = 'b';
    char*pc = &c;
    char**ppc = &pc;
    char* arr[10];
    test(&pc);
    test(ppc);
    test(arr);//Ok?
    return 0;
}
```

5. 函数指针

首先看一段代码：

```c
#include <stdio.h>
void test()
{
    printf("hehe\n");
}
int main()
{
    printf("%p\n", test);
    printf("%p\n", &test);
    return 0;
}
```

输出的结果：



输出的是两个地址，这两个地址是`test`函数的地址。
那我们的函数的地址要想保存起来，怎么保存？
下面我们看代码：

```c
void test()
{
    printf("hehe\n");
}
//下面pfun1和pfun2哪个有能力存放test函数的地址？
void (*pfun1)();
void *pfun2();
```

首先，能给存储地址，就要求pfun1或者pfun2是指针，那哪个是指针？
答案是：

> pfun1可以存放。pfun1先和*结合，说明pfun1是指针，指针指向的是一个函数，指向的函数无参数，返回值类型为void。

阅读两段有趣的代码：

```c
//代码1
(*(void (*)())0)();
//代码2
void (*signal(int , void(*)(int)))(int);
```

`注`:推荐《C陷阱和缺陷》

> 这本书中提及这两个代码。

代码2太复杂，如何简化：

```c
typedef void(*pfun_t)(int);
pfun_t signal(int, pfun_t);
```

6. 函数指针数组

数组是一个存放相同类型数据的存储空间，那我们已经学习了指针数组，
比如：

```
int *arr[10];
//数组的每个元素是int*
```

那要把函数的地址存到一个数组中，那这个数组就叫函数指针数组，那函数指针的数组如何定义呢？

```
int (*parr1[10])();
int *parr2[10]();
int (*)() parr3[10];
```

答案是：parr1
`parr1`先和`[]`结合，说明 parr1是数组，数组的内容是什么呢？
是`int (*)()`类型的函数指针。

函数指针数组的用途：**转移表**

例子：（计算器）

```
#include <stdio.h>
int add(int a, int b)
{
    return a + b;
}
int sub(int a, int b)
{
    return a - b;
}
int mul(int a, int b)
{
    return a*b;
}
int div(int a, int b)
{
    return a / b;
}
int main()
{
    int x, y;
    int input = 1;
    int ret = 0;
    do
    {
        printf( "*************************\n" );
        printf( "  1:add           2:sub  \n" );
        printf( "  3:mul           4:div  \n" );
        printf( "*************************\n" );
        printf( "请选择: " );
        scanf( "%d", &input);
        switch (input)
        {
```

```
        case 1:
              printf( "输入操作数：" );
              scanf( "%d %d", &x, &y);
              ret = add(x, y);
              printf( "ret = %d\n", ret);
              break;
        case 2:
              printf( "输入操作数：" );
              scanf( "%d %d", &x, &y);
              ret = sub(x, y);
              printf( "ret = %d\n", ret);
              break;
        case 3:
              printf( "输入操作数：" );
              scanf( "%d %d", &x, &y);
              ret = mul(x, y);
              printf( "ret = %d\n", ret);
              break;
        case 4:
              printf( "输入操作数：" );
              scanf( "%d %d", &x, &y);
              ret = div(x, y);
              printf( "ret = %d\n", ret);
              break;
        case 0:
                printf("退出程序\n");
                breark;
        default:
              printf( "选择错误\n" );
              break;
        }
    } while (input);

    return 0;
}
```

使用函数指针数组的实现：

```
#include <stdio.h>
int add(int a, int b)
{
           return a + b;
}
int sub(int a, int b)
{
           return a - b;
}
int mul(int a, int b)
{
           return a*b;
}
int div(int a, int b)
{
           return a / b;
}
int main()
{
```

```
    int x, y;
    int input = 1;
    int ret = 0;
    int(*p[5])(int x, int y) = { 0, add, sub, mul, div }; //转移表
    while (input)
    {
        printf( "*************************\n" );
        printf( "  1:add           2:sub  \n" );
        printf( "  3:mul           4:div  \n" );
        printf( "*************************\n" );
        printf( "请选择: " );
        scanf( "%d", &input);
        if ((input <= 4 && input >= 1))
        {
            printf( "输入操作数: " );
            scanf( "%d %d", &x, &y);
            ret = (*p[input])(x, y);
        }
        else
            printf( "输入有误\n" );
        printf( "ret = %d\n", ret);
    }
    return 0;
}
```

7. 指向函数指针数组的指针

指向函数指针数组的指针是一个`指针`
指针指向一个`数组`，数组的元素都是`函数指针`；

如何定义？

```
void test(const char* str)
{
    printf("%s\n", str);
}
int main()
{
    //函数指针pfun
    void (*pfun)(const char*) = test;
    //函数指针的数组pfunArr
    void (*pfunArr[5])(const char* str);
    pfunArr[0] = test;
    //指向函数指针数组pfunArr的指针ppfunArr
    void (*(*ppfunArr)[5])(const char*) = &pfunArr;
    return 0;
}
```

8. 回调函数

> 回调函数就是一个通过函数指针调用的函数。如果你把函数的指针（地址）作为参数传递给另一个函数，当这个指针被用来调用其所指向的函数时，我们就说这是回调函数。回调函数不是由该函数的实现方直接调用，而是在特定的事件或条件发生时由另外的一方调用的，用于对该事件或条件进行响应。

首先演示一下qsort函数的使用：

```
#include <stdio.h>

//qosrt函数的使用者得实现一个比较函数
int int_cmp(const void * p1, const void * p2)
{
      return (*( int *)p1 - *(int *) p2);
}

int main()
{
    int arr[] = { 1, 3, 5, 7, 9, 2, 4, 6, 8, 0 };
    int i = 0;

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

使用回调函数，模拟实现qsort（采用冒泡的方式）。

注意：这里第一次使用`void*`的指针，讲解`void*`的作用。

```
#include <stdio.h>

int int_cmp(const void * p1, const void * p2)
{
      return (*( int *)p1 - *(int *) p2);
}

void _swap(void *p1, void * p2, int size)
{
    int i = 0;
    for (i = 0; i< size; i++)
    {
        char tmp = *((char *)p1 + i);
       *(( char *)p1 + i) = *((char *) p2 + i);
       *(( char *)p2 + i) = tmp;
    }
}

void bubble(void *base, int count , int size, int(*cmp )(void *, void *))
{
    int i = 0;
    int j = 0;
```

```
    for (i = 0; i< count - 1; i++)
    {
       for (j = 0; j<count-i-1; j++)
       {
            if (cmp ((char *) base + j*size , (char *)base + (j + 1)*size) > 0)
           {
               _swap(( char *)base + j*size, (char *)base + (j + 1)*size, size);
           }
       }
    }
}
int main()
{
    int arr[] = { 1, 3, 5, 7, 9, 2, 4, 6, 8, 0 };
    //char *arr[] = {"aaaa","dddd","cccc","bbbb"};
    int i = 0;
    bubble(arr, sizeof(arr) / sizeof(arr[0]), sizeof (int), int_cmp);
    for (i = 0; i< sizeof(arr) / sizeof(arr[0]); i++)
    {
       printf( "%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

9. 指针和数组笔试题解析

```
//一维数组
int a[] = {1,2,3,4};
printf("%d\n",sizeof(a));
printf("%d\n",sizeof(a+0));
printf("%d\n",sizeof(*a));
printf("%d\n",sizeof(a+1));
printf("%d\n",sizeof(a[1]));
printf("%d\n",sizeof(&a));
printf("%d\n",sizeof(*&a));
printf("%d\n",sizeof(&a+1));
printf("%d\n",sizeof(&a[0]));
printf("%d\n",sizeof(&a[0]+1));

//字符数组
char arr[] = {'a','b','c','d','e','f'};
printf("%d\n", sizeof(arr));
printf("%d\n", sizeof(arr+0));
printf("%d\n", sizeof(*arr));
printf("%d\n", sizeof(arr[1]));
printf("%d\n", sizeof(&arr));
printf("%d\n", sizeof(&arr+1));
printf("%d\n", sizeof(&arr[0]+1));

printf("%d\n", strlen(arr));
printf("%d\n", strlen(arr+0));
printf("%d\n", strlen(*arr));
printf("%d\n", strlen(arr[1]));
printf("%d\n", strlen(&arr));
```

```
printf("%d\n", strlen(&arr+1));
printf("%d\n", strlen(&arr[0]+1));

char arr[] = "abcdef";
printf("%d\n", sizeof(arr));
printf("%d\n", sizeof(arr+0));
printf("%d\n", sizeof(*arr));
printf("%d\n", sizeof(arr[1]));
printf("%d\n", sizeof(&arr));
printf("%d\n", sizeof(&arr+1));
printf("%d\n", sizeof(&arr[0]+1));

printf("%d\n", strlen(arr));
printf("%d\n", strlen(arr+0));
printf("%d\n", strlen(*arr));
printf("%d\n", strlen(arr[1]));
printf("%d\n", strlen(&arr));
printf("%d\n", strlen(&arr+1));
printf("%d\n", strlen(&arr[0]+1));

char *p = "abcdef";
printf("%d\n", sizeof(p));
printf("%d\n", sizeof(p+1));
printf("%d\n", sizeof(*p));
printf("%d\n", sizeof(p[0]));
printf("%d\n", sizeof(&p));
printf("%d\n", sizeof(&p+1));
printf("%d\n", sizeof(&p[0]+1));

printf("%d\n", strlen(p));
printf("%d\n", strlen(p+1));
printf("%d\n", strlen(*p));
printf("%d\n", strlen(p[0]));
printf("%d\n", strlen(&p));
printf("%d\n", strlen(&p+1));
printf("%d\n", strlen(&p[0]+1));

//二维数组
int a[3][4] = {0};
printf("%d\n",sizeof(a));
printf("%d\n",sizeof(a[0][0]));
printf("%d\n",sizeof(a[0]));
printf("%d\n",sizeof(a[0]+1));
printf("%d\n",sizeof(*(a[0]+1)));
printf("%d\n",sizeof(a+1));
printf("%d\n",sizeof(*(a+1)));
printf("%d\n",sizeof(&a[0]+1));
printf("%d\n",sizeof(*(&a[0]+1)));
printf("%d\n",sizeof(*a));
printf("%d\n",sizeof(a[3]));
```

总结：

数组名的意义：

> 1. sizeof(数组名)，这里的数组名表示整个数组，计算的是整个数组的大小。
> 2. &数组名，这里的数组名表示整个数组，取出的是整个数组的地址。
> 3. 除此之外所有的数组名都表示首元素的地址。

10. 指针笔试题

> 笔试题1：

```
int main()
{
    int a[5] = { 1, 2, 3, 4, 5 };
    int *ptr = (int *)(&a + 1);
    printf( "%d,%d", *(a + 1), *(ptr - 1));
    return 0;
}
//程序的结果是什么？
```

> 笔试题2

```
//由于还没学习结构体，这里告知结构体的大小是20个字节
struct Test
{
    int Num;
    char *pcName;
    short sDate;
    char cha[2];
    short sBa[4];
}*p;
//假设p 的值为0x100000。 如下表表达式的值分别为多少？
//已知，结构体Test类型的变量大小是20个字节
int main()
{
    printf("%p\n", p + 0x1);
    printf("%p\n", (unsigned long)p + 0x1);
    printf("%p\n", (unsigned int*)p + 0x1);
    return 0;
}
```

> 笔试题3

```
int main()
{
    int a[4] = { 1, 2, 3, 4 };
    int *ptr1 = (int *)(&a + 1);
    int *ptr2 = (int *)((int)a + 1);
    printf( "%x,%x", ptr1[-1], *ptr2);
    return 0;
}
```

> 笔试题4

```
#include <stdio.h>
int main()
{
    int a[3][2] = { (0, 1), (2, 3), (4, 5) };
    int *p;
    p = a[0];
    printf( "%d", p[0]);
    return 0;
}
```

> 笔试题5

```
int main()
{
    int a[5][5];
    int(*p)[4];
    p = a;
    printf( "%p,%d\n", &p[4][2] - &a[4][2], &p[4][2] - &a[4][2]);
    return 0;
}
```

> 笔试题6

```
int main()
{
    int aa[2][5] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int *ptr1 = (int *)(&aa + 1);
    int *ptr2 = (int *)(*(aa + 1));
    printf( "%d,%d", *(ptr1 - 1), *(ptr2 - 1));
    return 0;
}
```

> 笔试题7

```
#include <stdio.h>

int main()
{
    char *a[] = {"work","at","alibaba"};
    char**pa = a;
    pa++;
    printf("%s\n", *pa);
    return 0;
}
```

> 笔试题8

```c
int main()
{
    char *c[] = {"ENTER","NEW","POINT","FIRST"};
    char**cp[] = {c+3,c+2,c+1,c};
    char***cpp = cp;
    printf("%s\n", **++cpp);
    printf("%s\n", *--*++cpp+3);
    printf("%s\n", *cpp[-2]+3);
    printf("%s\n", cpp[-1][-1]+1);
    return 0;
}
```

本章结束

BIT-3-字符函数和字符串函数

本章重点

> 重点介绍处理字符和字符串的库函数的使用和注意事项

- 求字符串长度
  - strlen
- 长度不受限制的字符串函数
  - strcpy
  - strcat
  - strcmp
- 长度受限制的字符串函数介绍
  - strncpy
  - strncat
  - strncmp
- 字符串查找
  - strstr
  - strtok
- 错误信息报告
  - strerror
- 字符操作
- 内存操作函数
  - memcpy
  - memmove
  - memset
  - memcmp

正文开始©比特就业课

0. 前言

> C语言中对字符和字符串的处理很是频繁，但是C语言本身是没有字符串类型的，字符串通常放在`常量字符串`中或者`字符数组`中。
>
> `字符串常量`适用于那些对它不做修改的字符串函数.

1. 函数介绍

1.1 strlen

```
size_t strlen ( const char * str );
```

- 字符串已经`'\0'`作为结束标志，strlen函数返回的是在字符串中`'\0'`前面出现的字符个数（不包含`'\0'`）。

- 参数指向的字符串必须要以`'\0'`结束。
- 注意函数的返回值为size_t，是无符号的（`易错`）
- 学会strlen函数的模拟实现

注:

```
#include <stdio.h>
int main()
{
    const char*str1 = "abcdef";
    const char*str2 = "bbb";
    if(strlen(str2)-strlen(str1)>0)
    {
        printf("str2>str1\n");
    }
    else
    {
        printf("srt1>str2\n");
    }
    return 0;
}
```

1.2 strcpy

```
char* strcpy(char * destination, const char * source );
```

- Copies the C string pointed by source into the array pointed by destination, including the terminating null character (and stopping at that point).
- 源字符串必须以`'\0'`结束。
- 会将源字符串中的`'\0'`拷贝到目标空间。
- 目标空间必须足够大，以确保能存放源字符串。
- 目标空间必须可变。
- 学会模拟实现。

1.3 strcat

```
char * strcat ( char * destination, const char * source );
```

- Appends a copy of the source string to the destination string. The terminating null character in destination is overwritten by the first character of source, and a null-character is included at the end of the new string formed by the concatenation of both in destination.
- 源字符串必须以`'\0'`结束。
- 目标空间必须有足够的大，能容纳下源字符串的内容。
- 目标空间必须可修改。
- 字符串自己给自己追加，如何?

1.4 strcmp

```
int strcmp ( const char * str1, const char * str2 );
```

- This function starts comparing the first character of each string. If they are equal to each other, it continues with the following pairs until the characters differ or until a terminating null-character is reached.
- 标准规定：
  - 第一个字符串大于第二个字符串，则返回大于0的数字
  - 第一个字符串等于第二个字符串，则返回0
  - 第一个字符串小于第二个字符串，则返回小于0的数字
  - `那么如何判断两个字符串？`

1.5 strncpy

```
char * strncpy ( char * destination, const char * source, size_t num );
```

- Copies the first num characters of source to destination. If the end of the source C string (which is signaled by a null-character) is found before num characters have been copied, destination is padded with zeros until a total of num characters have been written to it.
- 拷贝num个字符从源字符串到目标空间。
- 如果源字符串的长度小于num，则拷贝完源字符串之后，在目标的后边追加0，直到num个。

1.6 strncat

```
char * strncat ( char * destination, const char * source, size_t num );
```

- Appends the first num characters of source to destination, plus a terminating null-character.
- If the length of the C string in source is less than num, only the content up to the terminating null-character is copied.

```
/* strncat example */
#include <stdio.h>
#include <string.h>

int main ()
{
  char str1[20];
  char str2[20];
  strcpy (str1,"To be ");
  strcpy (str2,"or not to be");
  strncat (str1, str2, 6);
  puts (str1);
  return 0;
}
```

1.7 strncmp

```
int strncmp ( const char * str1, const char * str2, size_t num );
```

- 比较到出现另个字符不一样或者一个字符串结束或者num个字符全部比较完。

Return Value

Returns an integral value indicating the relationship between the strings:

return value	indicates
<0	the first character that does not match has a lower value in *str1* than in *str2*
0	the contents of both strings are equal
>0	the first character that does not match has a greater value in *str1* than in *str2*

```
/* strncmp example */
#include <stdio.h>
#include <string.h>

int main ()
{
  char str[][5] = { "R2D2" , "C3PO" , "R2A6" };
  int n;
  puts ("Looking for R2 astromech droids...");
  for (n=0 ; n<3 ; n++)
  if (strncmp (str[n],"R2xx",2) == 0)
  {
    printf ("found %s\n",str[n]);
  }
  return 0;
}
```

1.8 strstr

```
char * strstr ( const char *str1, const char * str2);
```

- Returns a pointer to the first occurrence of str2 in str1, or a null pointer if str2 is not part of str1.

```
/* strstr example */
#include <stdio.h>
#include <string.h>

int main ()
{
  char str[] ="This is a simple string";
  char * pch;
  pch = strstr (str,"simple");
  strncpy (pch,"sample",6);
  puts (str);
  return 0;
}
```

1.9 strtok

```
char * strtok ( char * str, const char * sep );
```

- sep参数是个字符串，定义了用作分隔符的字符集合
- 第一个参数指定一个字符串，它包含了0个或者多个由sep字符串中一个或者多个分隔符分割的标记。

- strtok函数找到str中的下一个标记，并将其用`\0`结尾，返回一个指向这个标记的指针。（注：strtok函数会改变被操作的字符串，所以在使用strtok函数切分的字符串一般都是临时拷贝的内容并且可修改。）
- strtok函数的第一个参数不为`NULL`，函数将找到str中第一个标记，strtok函数将保存它在字符串中的位置。
- strtok函数的第一个参数为`NULL`，函数将在同一个字符串中被保存的位置开始，查找下一个标记。
- 如果字符串中不存在更多的标记，则返回`NULL`指针。

```
/* strtok example */
#include <stdio.h>
#include <string.h>

int main ()
{
  char str[] ="- This, a sample string.";
  char * pch;
  printf ("Splitting string \"%s\" into tokens:\n",str);
  pch = strtok (str," ,.-");
  while (pch != NULL)
  {
    printf ("%s\n",pch);
    pch = strtok (NULL, " ,.-");
  }
  return 0;
}
```

```
#include <stdio.h>
int main()
{
   char *p = "zhangpengwei@bitedu.tech";
    const char* sep = ".@";
    char arr[30];
    char *str = NULL;
    strcpy(arr, p);//将数据拷贝一份，处理arr数组的内容
    for(str=strtok(arr, sep); str != NULL; str=strtok(NULL, sep))
    {
        printf("%s\n", str);
    }
}
```

1.10 strerror

```
char * strerror ( int errnum );
```

> 返回错误码，所对应的错误信息。

```
/* strerror example : error list */
#include <stdio.h>
#include <string.h>
#include <errno.h>//必须包含的头文件

int main ()
{
```

```
  FILE * pFile;
  pFile = fopen ("unexist.ent","r");
  if (pFile == NULL)
    printf ("Error opening file unexist.ent: %s\n",strerror(errno));
    //errno: Last error number
  return 0;
}
Edit & Run
```

字符分类函数：

函数	如果他的参数符合下列条件就返回真
iscntrl	任何控制字符
isspace	空白字符：空格' '，换页'\f'，换行'\n'，回车'\r'，制表符'\t'或者垂直制表符'\v'
isdigit	十进制数字 0~9
isxdigit	十六进制数字，包括所有十进制数字，小写字母a~f，大写字母A~F
islower	小写字母a~z
isupper	大写字母A~Z
isalpha	字母a~z或A~Z
isalnum	字母或者数字，a~z,A~Z,0~9
ispunct	标点符号，任何不属于数字或者字母的图形字符（可打印）
isgraph	任何图形字符
isprint	任何可打印字符，包括图形字符和空白字符

字符转换：

```
int tolower ( int c );
int toupper ( int c );
```

```
/* isupper example */
#include <stdio.h>
#include <ctype.h>
int main ()
{
  int i=0;
  char str[]="Test String.\n";
  char c;
  while (str[i])
  {
    c=str[i];
    if (isupper(c))
        c=tolower(c);
    putchar (c);
    i++;
  }
  return 0;
```

```
}
```

1.11 memcpy

```
void * memcpy ( void * destination, const void * source, size_t num );
```

- 函数memcpy从source的位置开始向后复制num个字节的数据到destination的内存位置。
- 这个函数在遇到`'\0'`的时候并不会停下来。
- 如果source和destination有任何的重叠，复制的结果都是未定义的。

```
/* memcpy example */
#include <stdio.h>
#include <string.h>

struct {
  char name[40];
  int age;
} person, person_copy;

int main ()
{
  char myname[] = "Pierre de Fermat";

  /* using memcpy to copy string: */
  memcpy ( person.name, myname, strlen(myname)+1 );
  person.age = 46;

  /* using memcpy to copy structure: */
  memcpy ( &person_copy, &person, sizeof(person) );

  printf ("person_copy: %s, %d \n", person_copy.name, person_copy.age );

  return 0;
}
```

1.12 memmove

```
void * memmove ( void * destination, const void * source, size_t num );
```

- 和memcpy的差别就是memmove函数处理的源内存块和目标内存块是可以重叠的。
- 如果源空间和目标空间出现重叠，就得使用memmove函数处理。

```c
/* memmove example */
#include <stdio.h>
#include <string.h>

int main ()
{
  char str[] = "memmove can be very useful.....";
  memmove (str+20,str+15,11);
  puts (str);
  return 0;
}
```

1.13 memcmp

```c
int memcmp ( const void * ptr1,
             const void * ptr2,
             size_t num );
```

- 比较从ptr1和ptr2指针开始的num个字节
- 返回值如下：

Return Value

Returns an integral value indicating the relationship between the content of the memory blocks:

return value	indicates
<0	the first byte that does not match in both memory blocks has a lower value in *ptr1* than in *ptr2* (if evaluated as *unsigned char* values)
0	the contents of both memory blocks are equal
>0	the first byte that does not match in both memory blocks has a greater value in *ptr1* than in *ptr2* (if evaluated as *unsigned char* values)

```c
/* memcmp example */
#include <stdio.h>
#include <string.h>

int main ()
{
  char buffer1[] = "DWgaOtP12df0";
  char buffer2[] = "DWGAOTP12DF0";

  int n;

  n=memcmp ( buffer1, buffer2, sizeof(buffer1) );

  if (n>0) printf ("'%s' is greater than '%s'.\n",buffer1,buffer2);
  else if (n<0) printf ("'%s' is less than '%s'.\n",buffer1,buffer2);
  else printf ("'%s' is the same as '%s'.\n",buffer1,buffer2);

  return 0;
}
```

2. 库函数的模拟实现

2.1 模拟实现strlen

三种方式：
方式1：

```
//计数器方式
int my_strlen(const char * str)
{
    int count = 0;
    while(*str)
    {
        count++;
        str++;
    }
    return count;
}
```

方式2：

```
//不能创建临时变量计数器
int my_strlen(const char * str)
{
    if(*str == '\0')
        return 0;
    else
        return 1+my_strlen(str+1);
}
```

方式3：

```
//指针-指针的方式
int my_strlen(char *s)
{
        char *p = s;
        while(*p != '\0' )
                p++;
        return p-s;
}
```

2.2 模拟实现strcpy

参考代码：

```
//1.参数顺序
//2.函数的功能，停止条件
//3.assert
//4.const修饰指针
//5.函数返回值
//6.题目出自《高质量C/C++编程》书籍最后的试题部分
char *my_strcpy(char *dest, const char*src)
{
    char *ret = dest;
    assert(dest != NULL);
```

```
    assert(src != NULL);

    while((*dest++ = *src++))
    {
        ;
    }
    return ret;
}
```

2.3 模拟实现strcat

参考代码：

```
char *my_strcat(char *dest, const char*src)
{
    char *ret = dest;
    assert(dest != NULL);
    assert(src != NULL);
    while(*dest)
    {
        dest++;
    }
    while((*dest++ = *src++))
    {
        ;
    }
    return ret;
}
```

2.4 模拟实现strstr

注：让他们下去自己研究一下KMP算法。

```
char *  strstr (const char * str1, const char * str2)
{
        char *cp = (char *) str1;
        char *s1, *s2;

        if ( !*str2 )
            return((char *)str1);

        while (*cp)
        {
                s1 = cp;
                s2 = (char *) str2;

                while ( *s1 && *s2 && !(*s1-*s2) )
                        s1++, s2++;

                if (!*s2)
                        return(cp);

                cp++;
        }

        return(NULL);
```

```
}
```

2.5 模拟实现strcmp

参考代码：

```
int my_strcmp (const char * src, const char * dst)
{
        int ret = 0 ;
        assert(src != NULL);
        assert(dest != NULL);
        while( ! (ret = *(unsigned char *)src - *(unsigned char *)dst) && *dst)
                ++src, ++dst;

        if ( ret < 0 )
                ret = -1 ;
        else if ( ret > 0 )
                ret = 1 ;

        return( ret );
}
```

2.6 模拟实现memcpy

参考代码：

```
void * memcpy ( void * dst, const void * src, size_t count)
{
        void * ret = dst;
        assert(dst);
        assert(src);
        /*
         * copy from lower addresses to higher addresses
         */
        while (count--) {
                *(char *)dst = *(char *)src;
                dst = (char *)dst + 1;
                src = (char *)src + 1;
        }

        return(ret);
}
```

2.7 模拟实现memmove

参考代码：

```
void * memmove ( void * dst, const void * src, size_t count)
{
        void * ret = dst;

        if (dst <= src || (char *)dst >= ((char *)src + count)) {
                /*
                 * Non-Overlapping Buffers
```

```
                 * copy from lower addresses to higher addresses
                 */
                while (count--) {
                        *(char *)dst = *(char *)src;
                        dst = (char *)dst + 1;
                        src = (char *)src + 1;
                }
        }
        else {
                /*
                 * Overlapping Buffers
                 * copy from higher addresses to lower addresses
                 */
                dst = (char *)dst + count - 1;
                src = (char *)src + count - 1;

                while (count--) {
                        *(char *)dst = *(char *)src;
                        dst = (char *)dst - 1;
                        src = (char *)src - 1;
                }
        }

        return(ret);
}
```

本章完

BIT-4-自定义类型：结构体，枚举，联合

本章重点

- 结构体
  - 结构体类型的声明
  - 结构的自引用
  - 结构体变量的定义和初始化
  - 结构体内存对齐
  - 结构体传参
  - 结构体实现位段（位段的填充&可移植性）
- 枚举
  - 枚举类型的定义
  - 枚举的优点
  - 枚举的使用
- 联合
  - 联合类型的定义
  - 联合的特点
  - 联合大小的计算

正文开始©比特就业课

结构体

1 结构体的声明

1.1 结构的基础知识

> 结构是一些值的集合，这些值称为成员变量。结构的每个成员可以是不同类型的变量。

1.2 结构的声明

```
struct tag
{
    member-list;
}variable-list;
```

例如描述一个学生：

```
struct Stu
{
    char name[20];//名字
    int age;//年龄
    char sex[5];//性别
    char id[20];//学号
}; //分号不能丢
```

1.3 特殊的声明

在声明结构的时候，可以不完全的声明。

比如：

```
//匿名结构体类型
struct
{
    int a;
    char b;
    float c;
}x;
struct
{
    int a;
    char b;
    float c;
}a[20], *p;
```

上面的两个结构在声明的时候省略掉了结构体标签（tag）。

那么问题来了？

```
//在上面代码的基础上，下面的代码合法吗？
p = &x;
```

警告：

编译器会把上面的两个声明当成完全不同的两个类型。

所以是非法的。

1.4 结构的自引用

在结构中包含一个类型为该结构本身的成员是否可以呢？

```
//代码1
struct Node
{
    int data;
    struct Node next;
};
//可行否？
如果可以，那sizeof(struct Node)是多少？
```

正确的自引用方式：

```
//代码2
struct Node
{
    int data;
    struct Node* next;
};
```

注意：

```
//代码3
typedef struct
{
    int data;
    Node* next;
}Node;
//这样写代码，可行否？

//解决方案：
typedef struct Node
{
    int data;
    struct Node* next;
}Node;
```

1.5 结构体变量的定义和初始化

有了结构体类型，那如何定义变量，其实很简单。

```
struct Point
{
    int x;
    int y;
}p1;                //声明类型的同时定义变量p1
struct Point p2;    //定义结构体变量p2


//初始化：定义变量的同时赋初值。
struct Point p3 = {x, y};

struct Stu        //类型声明
{
    char name[15];//名字
    int age;      //年龄
};
struct Stu s = {"zhangsan", 20};//初始化

struct Node
{
    int data;
    struct Point p;
    struct Node* next;
}n1 = {10, {4,5}, NULL};            //结构体嵌套初始化

struct Node n2 = {20, {5, 6}, NULL};//结构体嵌套初始化
```

1.6 结构体内存对齐

我们已经掌握了结构体的基本使用了。

现在我们深入讨论一个问题：计算结构体的大小。

这也是一个特别热门的考点：`结构体内存对齐`

```
//练习1
struct S1
```

```
{
    char c1;
    int i;
    char c2;
};
printf("%d\n", sizeof(struct S1));

//练习2
struct S2
{
    char c1;
    char c2;
    int i;
};
printf("%d\n", sizeof(struct S2));

//练习3
struct S3
{
    double d;
    char c;
    int i;
};
printf("%d\n", sizeof(struct S3));

//练习4-结构体嵌套问题
struct S4
{
    char c1;
    struct S3 s3;
    double d;
};
printf("%d\n", sizeof(struct S4));
```

考点
如何计算?
首先得掌握结构体的对齐规则:

> 1. 第一个成员在与结构体变量偏移量为0的地址处。
> 2. 其他成员变量要对齐到某个数字（对齐数）的整数倍的地址处。
>
>   **对齐数** = 编译器默认的一个对齐数 与 该成员大小的**较小值**。
>
>   - `VS中默认的值为8`
>
> 3. 结构体总大小为最大对齐数（每个成员变量都有一个对齐数）的整数倍。
> 4. 如果嵌套了结构体的情况，嵌套的结构体对齐到自己的最大对齐数的整数倍处，结构体的整体大小就是所有最大对齐数（含嵌套结构体的对齐数）的整数倍。

为什么存在内存对齐?

> 大部分的参考资料都是如是说的:

1. **平台原因(移植原因)**:
   不是所有的硬件平台都能访问任意地址上的任意数据的；某些硬件平台只能在某些地址处取某些特定类型的数据，否则抛出硬件异常。

2. **性能原因**：
   数据结构(尤其是栈)应该尽可能地在自然边界上对齐。
   原因在于，为了访问未对齐的内存，处理器需要作两次内存访问；而对齐的内存访问仅需要一次访问。

总体来说：

> 结构体的内存对齐是拿**空间**来换取**时间**的做法。

那在设计结构体的时候，我们既要满足对齐，又要节省空间，如何做到：

> 让占用空间小的成员尽量集中在一起。

```
//例如：
struct S1
{
    char c1;
    int i;
    char c2;
};
struct S2
{
    char c1;
    char c2;
    int i;
};
```

S1和S2类型的成员一模一样，但是S1和S2所占空间的大小有了一些区别。

1.7 修改默认对齐数

之前我们见过了`#pragma`这个预处理指令，这里我们再次使用，可以改变我们的默认对齐数。

```
#include <stdio.h>
#pragma pack(8)//设置默认对齐数为8
struct S1
{
    char c1;
    int i;
    char c2;
};
#pragma pack()//取消设置的默认对齐数，还原为默认

#pragma pack(1)//设置默认对齐数为1
struct S2
{
    char c1;
    int i;
    char c2;
};
#pragma pack()//取消设置的默认对齐数，还原为默认
int main()
{
    //输出的结果是什么？
    printf("%d\n", sizeof(struct S1));
    printf("%d\n", sizeof(struct S2));
```

```
    return 0;
}
```

结论：

> 结构在对齐方式不合适的时候，我么可以自己更改默认对齐数。

百度笔试题：

> 写一个宏，计算结构体中某变量相对于首地址的偏移，并给出说明

考察：`offsetof`宏的实现

注：这里还没学习宏，可以放在宏讲解完后再实现。

1.8 结构体传参

直接上代码：

```
struct S
{
    int data[1000];
    int num;
};

struct S s = {{1,2,3,4}, 1000};
//结构体传参
void print1(struct S s)
{
    printf("%d\n", s.num);
}
//结构体地址传参
void print2(struct S* ps)
{
    printf("%d\n", ps->num);
}

int main()
{
    print1(s);  //传结构体
    print2(&s); //传地址
    return 0;
}
```

上面的`print1`和`print2`函数哪个好些?

答案是：首选print2函数。
原因：

> 函数传参的时候，参数是需要压栈，会有时间和空间上的系统开销。
>
> 如果传递一个结构体对象的时候，结构体过大，参数压栈的的系统开销比较大，所以会导致性能的下降。

结论：
结构体传参的时候，要传结构体的地址。

2. 位段

结构体讲完就得讲讲结构体实现`位段`的能力。

2.1 什么是位段

位段的声明和结构是类似的，有两个不同：

> 1.位段的成员必须是`int、unsigned int 或signed int`。
>
> 2.位段的成员名后边有一个冒号和一个数字。

比如：

```
struct A
{
    int _a:2;
    int _b:5;
    int _c:10;
    int _d:30;
};
```

A就是一个位段类型。

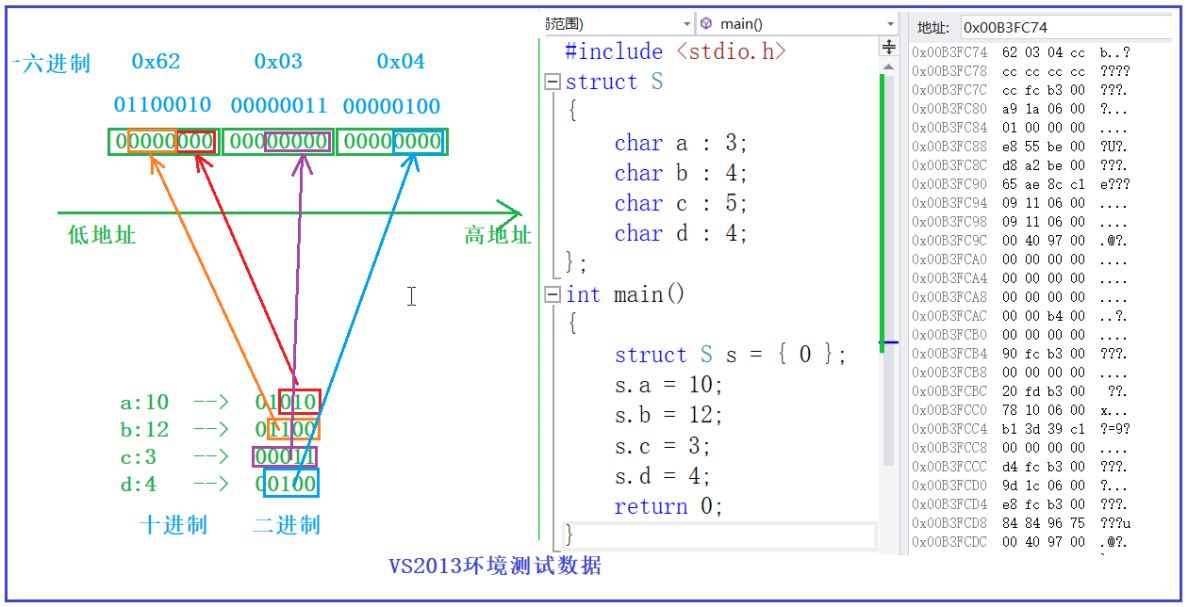
那位段A的大小是多少？

```
printf("%d\n", sizeof(struct A));
```

2.2 位段的内存分配

1. 位段的成员可以是`int` `unsigned int` `signed int`或者是`char`（属于整形家族）类型
2. 位段的空间上是按照需要以4个字节（`int`）或者1个字节（`char`）的方式来开辟的。
3. 位段涉及很多不确定因素，位段是不跨平台的，注重可移植的程序应该避免使用位段。

```
//一个例子
struct S
{
    char a:3;
    char b:4;
    char c:5;
    char d:4;
};
struct S s = {0};
s.a = 10;
s.b = 12;
s.c = 3;
s.d = 4;

//空间是如何开辟的？
```



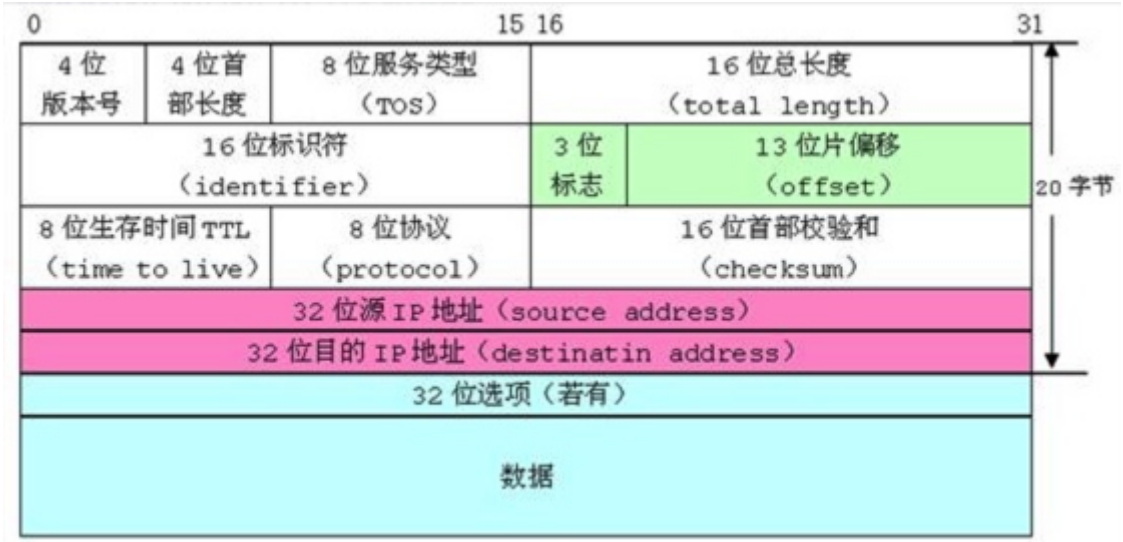
2.3 位段的跨平台问题

1. int 位段被当成有符号数还是无符号数是不确定的。
2. 位段中最大位的数目不能确定。（16位机器最大16，32位机器最大32，写成27，在16位机器会出问题。
3. 位段中的成员在内存中从左向右分配，还是从右向左分配标准尚未定义。
4. 当一个结构包含两个位段，第二个位段成员比较大，无法容纳于第一个位段剩余的位时，是舍弃剩余的位还是利用，这是不确定的。

总结:

> 跟结构相比，位段可以达到同样的效果，但是可以很好的节省空间，但是有跨平台的问题存在。

2.4 位段的应用



3. 枚举

枚举顾名思义就是一一列举。

把可能的取值一一列举。

比如我们现实生活中：

> 一周的星期一到星期日是有限的7天，可以一一列举。

> 性别有：男、女、保密，也可以一一列举。
>
> 月份有12个月，也可以一一列举

这里就可以使用枚举了。

3.1 枚举类型的定义

```
enum Day//星期
{
    Mon,
    Tues,
    Wed,
    Thur,
    Fri,
    Sat,
    Sun
};
enum Sex//性别
{
    MALE,
    FEMALE,
    SECRET
};
enum Color//颜色
{
    RED,
    GREEN,
    BLUE
};
```

以上定义的`enum Day`，`enum Sex`，`enum Color`都是枚举类型。
{}中的内容是枚举类型的可能取值，也叫`枚举常量`。

这些可能取值都是有值的，默认从0开始，一次递增1，当然在定义的时候也可以赋初值。
例如：

```
enum Color//颜色
{
    RED=1,
    GREEN=2,
    BLUE=4
};
```

3.2 枚举的优点

> 为什么使用枚举?

我们可以使用`#define`定义常量，为什么非要使用枚举?
枚举的优点：

1. 增加代码的可读性和可维护性
2. 和#define定义的标识符比较枚举有类型检查，更加严谨。
3. 防止了命名污染（封装）
4. 便于调试
5. 使用方便，一次可以定义多个常量

3.3 枚举的使用

```
enum Color//颜色
{
    RED=1,
    GREEN=2,
    BLUE=4
};

enum Color clr = GREEN;//只能拿枚举常量给枚举变量赋值，才不会出现类型的差异。
clr = 5;               //ok??
```

4. 联合（共用体）

4.1 联合类型的定义

联合也是一种特殊的自定义类型
这种类型定义的变量也包含一系列的成员，特征是这些成员公用同一块空间（所以联合也叫共用体）。
比如：

```
//联合类型的声明
union Un
{
    char c;
    int i;
};

//联合变量的定义
union Un un;
//计算连个变量的大小
printf("%d\n", sizeof(un));
```

4.2 联合的特点

> 联合的成员是共用同一块内存空间的，这样一个联合变量的大小，至少是最大成员的大小（因为联合至少得有能力保存最大的那个成员）。

```
union Un
{
    int i;
    char c;
};
union Un un;

// 下面输出的结果是一样的吗？
printf("%d\n", &(un.i));
printf("%d\n", &(un.c));

//下面输出的结果是什么？
un.i = 0x11223344;
un.c = 0x55;
printf("%x\n", un.i);
```

面试题：

> 判断当前计算机的大小端存储

4.3 联合大小的计算

- 联合的大小至少是最大成员的大小。
- 当最大成员大小不是最大对齐数的整数倍的时候，就要对齐到最大对齐数的整数倍。

比如：

```
union Un1
{
    char c[5];
    int i;
};
union Un2
{
    short c[7];
    int i;
};
//下面输出的结果是什么？
printf("%d\n", sizeof(union Un1));
printf("%d\n", sizeof(union Un2));
```

---

5. 练习

通讯录程序

本章完

BIT-5-动态内存管理

本章重点

- 为什么存在动态内存分配
- 动态内存函数的介绍
  - malloc
  - free
  - calloc
  - realloc
- 常见的动态内存错误
- 几个经典的笔试题
- 柔性数组

正文开始©比特就业课

1. 为什么存在动态内存分配

我们已经掌握的内存开辟方式有：

```
int val = 20;//在栈空间上开辟四个字节
char arr[10] = {0};//在栈空间上开辟10个字节的连续空间
```

但是上述的开辟空间的方式有两个特点：

1. 空间开辟大小是固定的。
2. 数组在申明的时候，必须指定数组的长度，它所需要的内存在编译时分配。

但是对于空间的需求，不仅仅是上述的情况。有时候我们需要的空间大小在程序运行的时候才能知道，那数组的编译时开辟空间的方式就不能满足了。
这时候就只能试试动态存开辟了。

2. 动态内存函数的介绍

2.1 malloc和free

C语言提供了一个动态内存开辟的函数：

```
void* malloc (size_t size);
```

这个函数向内存申请一块**连续可用**的空间，并返回指向这块空间的指针。

- 如果开辟成功，则返回一个指向开辟好空间的指针。
- 如果开辟失败，则返回一个NULL指针，因此malloc的返回值一定要做检查。
- 返回值的类型是`void*`，所以malloc函数并不知道开辟空间的类型，具体在使用的时候使用者自己来决定。
- 如果参数`size`为0，malloc的行为是标准是未定义的，取决于编译器。

C语言提供了另外一个函数free，专门是用来做动态内存的释放和回收的，函数原型如下：

```
void free (void* ptr);
```

free函数用来释放动态开辟的内存。

- 如果参数`ptr`指向的空间不是动态开辟的，那free函数的行为是未定义的。
- 如果参数`ptr`是NULL指针，则函数什么事都不做。

malloc和free都声明在`stdlib.h`头文件中。
举个例子：

```
#include <stdio.h>

int main()
{
    //代码1
    int num = 0;
    scanf("%d", &num);
    int arr[num] = {0};
    //代码2
    int* ptr = NULL;
    ptr = (int*)malloc(num*sizeof(int));
    if(NULL != ptr)//判断ptr指针是否为空
    {
        int i = 0;
        for(i=0; i<num; i++)
        {
            *(ptr+i) = 0;
        }
    }
    free(ptr);//释放ptr所指向的动态内存
    ptr = NULL;//是否有必要？
    return 0;
}
```

2.2 calloc

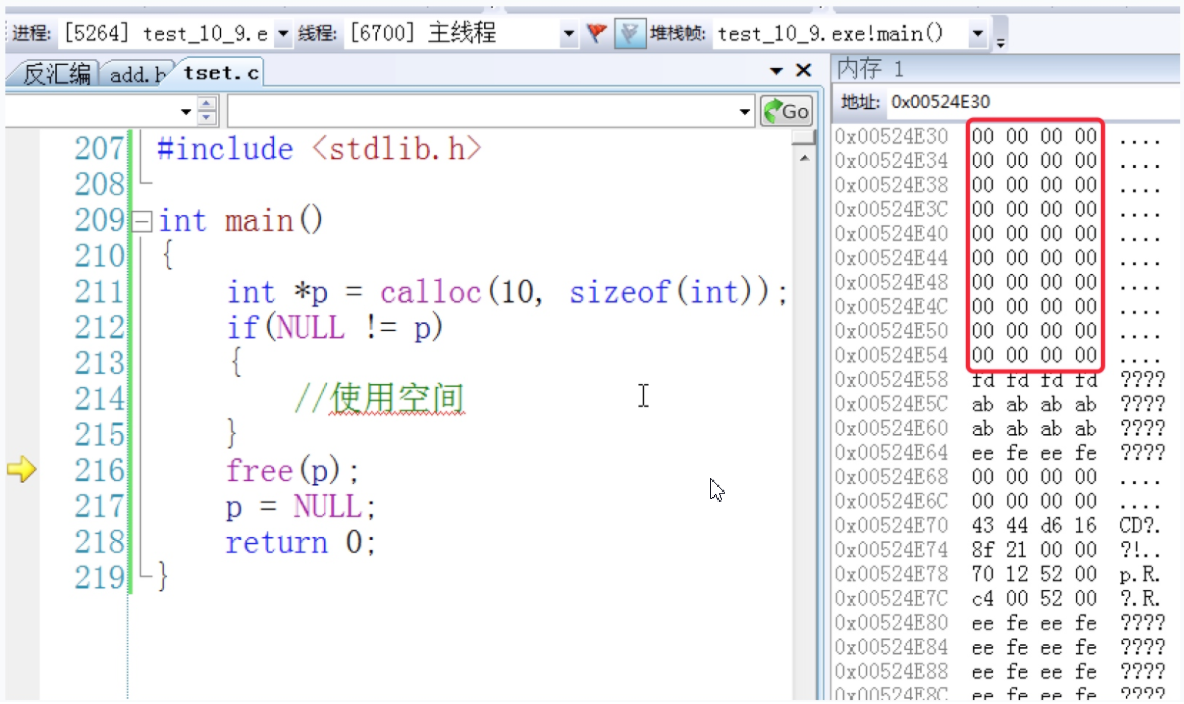
C语言还提供了一个函数叫`calloc`，`calloc`函数也用来动态内存分配。原型如下：

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

- 函数的功能是为`num`个大小为`size`的元素开辟一块空间，并且把空间的每个字节初始化为0。
- 与函数`malloc`的区别只在于`calloc`会在返回地址之前把申请的空间的每个字节初始化为全0。
  举个例子：

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

int main()
{
    int *p = (int*)calloc(10, sizeof(int));
    if(NULL != p)
    {
        //使用空间
```

```
    }
    free(p);
    p = NULL;
    return 0;
}
```



所以如何我们对申请的内存空间的内容要求初始化，那么可以很方便的使用calloc函数来完成任务。

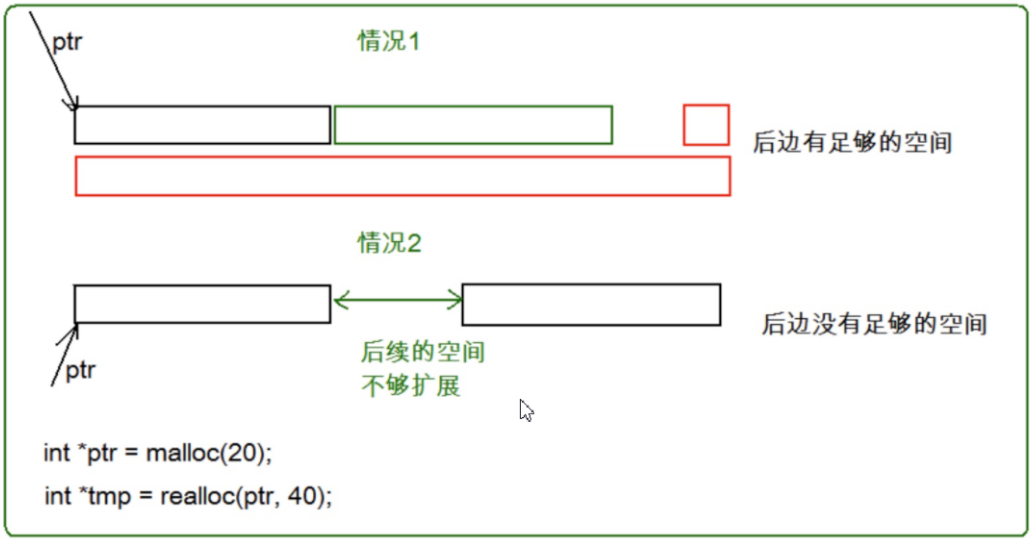
2.3 realloc

- realloc函数的出现让动态内存管理更加灵活。
- 有时会我们发现过去申请的空间太小了，有时候我们又会觉得申请的空间过大了，那为了合理的时候内存，我们一定会对内存的大小做灵活的调整。那 `realloc` 函数就可以做到对动态开辟内存大小的调整。
  函数原型如下：

```
void* realloc (void* ptr, size_t size);
```

- `ptr` 是要调整的内存地址
- `size` 调整之后新大小
- 返回值为调整之后的内存起始位置。
- 这个函数调整原内存空间大小的基础上，还会将原来内存中的数据移动到 `新` 的空间。
- realloc在调整内存空间的是存在两种情况：
  - 情况1：原有空间之后有足够大的空间

- 情况2：原有空间之后没有足够大的空间



情况1
当是情况1 的时候，要扩展内存就直接原有内存之后直接追加空间，原来空间的数据不发生变化。
情况2
当是情况2 的时候，原有空间之后没有足够多的空间时，扩展的方法是：在堆空间上另找一个合适大小的连续空间来使用。这样函数返回的是一个新的内存地址。
由于上述的两种情况，realloc函数的使用就要注意一些。
举个例子：

```
#include <stdio.h>

int main()
{
    int *ptr = (int*)malloc(100);
    if(ptr != NULL)
    {
         //业务处理
    }
    else
    {
         exit(EXIT_FAILURE);
    }
    //扩展容量
    //代码1
    ptr = (int*)realloc(ptr, 1000);//这样可以吗？(如果申请失败会如何？)

    //代码2
    int*p = NULL;
    p = realloc(ptr, 1000);
    if(p != NULL)
    {
        ptr = p;
    }
    //业务处理
    free(ptr);
    return 0;
}
```

3. 常见的动态内存错误

3.1 对NULL指针的解引用操作

```
void test()
{
    int *p = (int *)malloc(INT_MAX/4);
    *p = 20;//如果p的值是NULL，就会有问题
    free(p);
}
```

3.2 对动态开辟空间的越界访问

```
void test()
{
    int i = 0;
    int *p = (int *)malloc(10*sizeof(int));
    if(NULL == p)
    {
        exit(EXIT_FAILURE);
    }
    for(i=0; i<=10; i++)
    {
        *(p+i) = i;//当i是10的时候越界访问
    }
    free(p);
}
```

3.3 对非动态开辟内存使用free释放

```
void test()
{
    int a = 10;
    int *p = &a;
    free(p);//ok?
}
```

3.4 使用free释放一块动态开辟内存的一部分

```
void test()
{
    int *p = (int *)malloc(100);
    p++;
    free(p);//p不再指向动态内存的起始位置
}
```

3.5 对同一块动态内存多次释放

```c
void test()
{
    int *p = (int *)malloc(100);
    free(p);
    free(p);//重复释放
}
```

3.6 动态开辟内存忘记释放（内存泄漏）

```c
void test()
{
    int *p = (int *)malloc(100);
    if(NULL != p)
    {
        *p = 20;
    }
}

int main()
{
    test();
    while(1);
}
```

忘记释放不再使用的动态开辟的空间会造成内存泄漏。
切记：
`动态开辟的空间一定要释放，并且正确释放`。

4. 几个经典的笔试题

4.1 题目1：

```c
void GetMemory(char *p)
{
    p = (char *)malloc(100);
}
void Test(void)
{
    char *str = NULL;
    GetMemory(str);
    strcpy(str, "hello world");
    printf(str);
}
```

请问运行Test 函数会有什么样的结果?

4.2 题目2：

```
char *GetMemory(void)
{
    char p[] = "hello world";
    return p;
}
void Test(void)
{
    char *str = NULL;
    str = GetMemory();
    printf(str);
}
```

请问运行Test 函数会有什么样的结果?

4.3 题目3:

```
void GetMemory(char **p, int num)
{
    *p = (char *)malloc(num);
}
void Test(void)
{
    char *str = NULL;
    GetMemory(&str, 100);
    strcpy(str, "hello");
    printf(str);
}
```

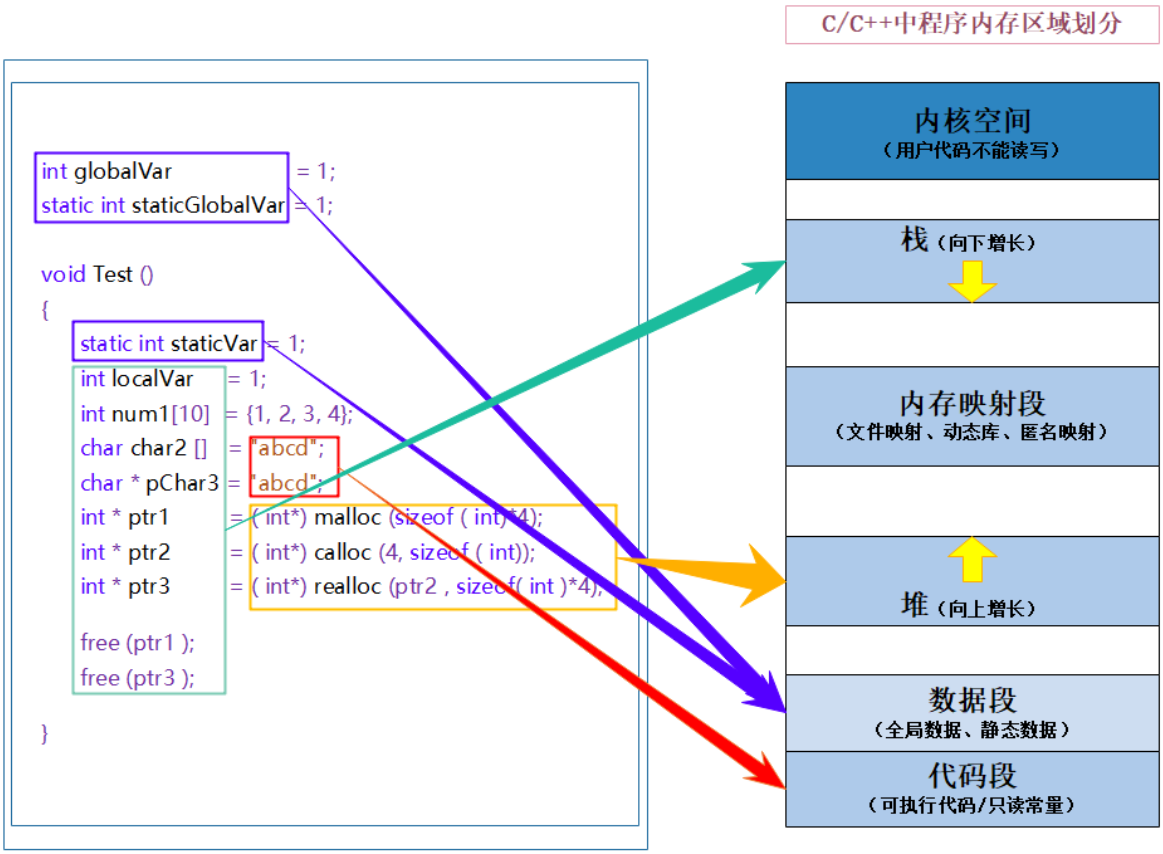
请问运行Test 函数会有什么样的结果?

4.4 题目4:

```
void Test(void)
{
    char *str = (char *) malloc(100);
    strcpy(str, "hello");
    free(str);
    if(str != NULL)
    {
        strcpy(str, "world");
        printf(str);
    }
}
```

请问运行Test 函数会有什么样的结果?

5. C/C++程序的内存开辟



C/C++程序内存分配的几个区域：

> 1. 栈区（stack）：在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。 栈区主要存放运行函数而分配的局部变量、函数参数、返回数据、返回地址等。
> 2. 堆区（heap）：一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS回收 。分配方式类似于链表。
> 3. 数据段（静态区）（static）存放全局变量、静态数据。程序结束后由系统释放。
> 4. 代码段：存放函数体（类成员函数和全局函数）的二进制代码。

有了这幅图，我们就可以更好的理解在《C语言初识》中讲的static关键字修饰局部变量的例子了。

> 实际上普通的局部变量是在**栈区**分配空间的，栈区的特点是在上面创建的变量出了作用域就销毁。
>
> 但是被static修饰的变量存放在**数据段（静态区）**，数据段的特点是在上面创建的变量，直到程序结束才销毁
>
> 所以生命周期变长。

6. 柔性数组

> 也许你从来没有听说过**柔性数组（flexible array）**这个概念，但是它确实是存在的。
> C99 中，结构中的最后一个元素允许是未知大小的数组，这就叫做『柔性数组』成员。

例如：

```
typedef struct st_type
{
    int i;
    int a[0];//柔性数组成员
}type_a;
```

有些编译器会报错无法编译可以改成：

```
typedef struct st_type
{
    int i;
    int a[];//柔性数组成员
}type_a;
```

6.1 柔性数组的特点：

- 结构中的柔性数组成员前面必须至少一个其他成员。
- sizeof 返回的这种结构大小不包括柔性数组的内存。
- 包含柔性数组成员的结构用malloc ()函数进行内存的动态分配，并且分配的内存应该大于结构的大小，以适应柔性数组的预期大小。

例如：

```
//code1
typedef struct st_type
{
    int i;
    int a[0];//柔性数组成员
}type_a;
printf("%d\n", sizeof(type_a));//输出的是4
```

6.2 柔性数组的使用

```
//代码1
int i = 0;
type_a *p = (type_a*)malloc(sizeof(type_a)+100*sizeof(int));
//业务处理
p->i = 100;
for(i=0; i<100; i++)
{
    p->a[i] = i;
}
free(p);
```

这样柔性数组成员a，相当于获得了100个整型元素的连续空间。

6.3 柔性数组的优势

上述的`type_a`结构也可以设计为：

```
//代码2
typedef struct st_type
{
    int i;
```

```
    int *p_a;
}type_a;
type_a *p = (type_a *)malloc(sizeof(type_a));
p->i = 100;
p->p_a = (int *)malloc(p->i*sizeof(int));

//业务处理
for(i=0; i<100; i++)
{
    p->p_a[i] = i;
}

//释放空间
free(p->p_a);
p->p_a = NULL;
free(p);
p = NULL;
```

上述 代码1 和 代码2 可以完成同样的功能，但是 方法1 的实现有两个好处：

第一个好处是：方便内存释放

> 如果我们的代码是在一个给别人用的函数中，你在里面做了二次内存分配，并把整个结构体返回给用户。用户调用free可以释放结构体，但是用户并不知道这个结构体内的成员也需要free，所以你不能指望用户来发现这个事。所以，如果我们把结构体的内存以及其成员要的内存一次性分配好了，并返回给用户一个结构体指针，用户做一次free就可以把所有的内存也给释放掉。

第二个好处是：这样有利于访问速度.

> 连续的内存有益于提高访问速度，也有益于减少内存碎片。（其实，我个人觉得也没多高了，反正你跑不了要用做偏移量的加法来寻址）

扩展阅读：

C语言结构体里的数组和指针

本章完

BIT-6-C语言文件操作

本章重点

1. 为什么使用文件
2. 什么是文件
3. 文件的打开和关闭
4. 文件的顺序读写
5. 文件的随机读写
6. 文本文件和二进制文件
7. 文件读取结束的判定
8. 文件缓冲区

---

1. 为什么使用文件

我们前面学习结构体时，写了通讯录的程序，当通讯录运行起来的时候，可以给通讯录中增加、删除数据，此时数据是存放在内存中，当程序退出的时候，通讯录中的数据自然就不存在了，等下次运行通讯录程序的时候，数据又得重新录入，如果使用这样的通讯录就很难受。

我们在想既然是通讯录就应该把信息记录下来，只有我们自己选择删除数据的时候，数据才不复存在。这就涉及到了数据持久化的问题，我们一般数据持久化的方法有，把数据存放在磁盘文件、存放到数据库等方式。

使用文件我们可以将数据直接存放在电脑的硬盘上，做到了数据的持久化。

2. 什么是文件

磁盘上的文件是文件。

但是在程序设计中，我们一般谈的文件有两种：程序文件、数据文件（从文件功能的角度来分类的）。

2.1 程序文件

> 包括源程序文件（后缀为.c）,目标文件（windows环境后缀为.obj）,可执行程序（windows环境后缀为.exe）。

2.2 数据文件

> 文件的内容不一定是程序，而是程序运行时读写的数据，比如程序运行需要从中读取数据的文件，或者输出内容的文件。

本章讨论的是数据文件。

在以前各章所处理数据的输入输出都是以终端为对象的，即从终端的键盘输入数据，运行结果显示到显示器上。

其实有时候我们会把信息输出到磁盘上，当需要的时候再从磁盘上把数据读取到内存中使用，这里处理的就是磁盘上文件。

2.3 文件名

一个文件要有一个唯一的文件标识，以便用户识别和引用。

文件名包含3部分：文件路径+文件名主干+文件后缀

例如：`c:\code\test.txt`

为了方便起见，文件标识常被称为**文件名。**

3. 文件的打开和关闭

3.1 文件指针

缓冲文件系统中，关键的概念是“**文件类型指针**”，简称“**文件指针**”。

每个被使用的文件都在内存中开辟了一个相应的文件信息区，用来存放文件的相关信息（如文件的名字，文件状态及文件当前的位置等）。这些信息是保存在一个结构体变量中的。该结构体类型是有系统声明的，取名**FILE**.

例如，VS2013编译环境提供的`stdio.h`头文件中有以下的文件类型申明：

```
struct _iobuf {
        char *_ptr;
        int   _cnt;
        char *_base;
        int   _flag;
        int   _file;
        int   _charbuf;
        int   _bufsiz;
        char *_tmpfname;
        };
typedef struct _iobuf FILE;
```

不同的C编译器的FILE类型包含的内容不完全相同，但是大同小异。

每当打开一个文件的时候，系统会根据文件的情况自动创建一个FILE结构的变量，并填充其中的信息，使用者不必关心细节。

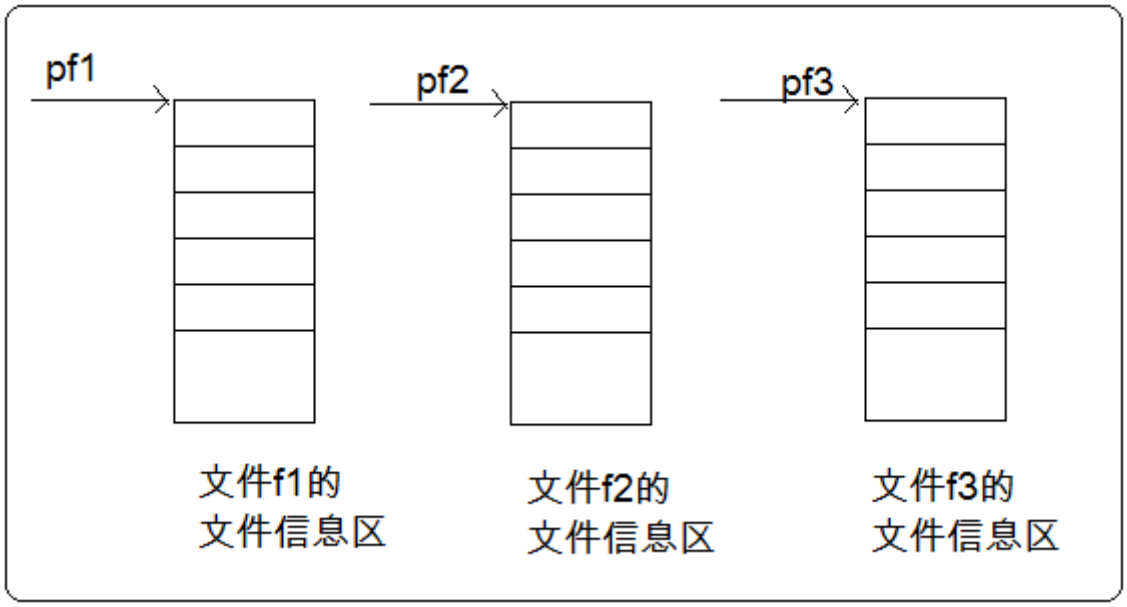
一般都是通过一个FILE*的指针来维护这个FILE结构的变量，这样使用起来更加方便。*

下面我们可以创建一个FILE*的指针变量:

```
FILE* pf;//文件指针变量
```

定义pf是一个指向FILE类型数据的指针变量。可以使pf指向某个文件的文件信息区（是一个结构体变量）。通过该文件信息区中的信息就能够访问该文件。也就是说，**通过文件指针变量能够找到与它关联的文件。**

比如：



3.2 文件的打开和关闭

文件在读写之前应该先**打开文件**，在使用结束之后应该**关闭文件**。

在编写程序的时候，在打开文件的同时，都会返回一个FILE*的指针变量指向该文件，也相当于建立了指针和文件的关系。

ANSIC 规定使用fopen函数来打开文件，fclose来关闭文件。

```
//打开文件
FILE * fopen ( const char * filename, const char * mode );
//关闭文件
int fclose ( FILE * stream );
```

打开方式如下：

文件使用方式	含义	如果指定文件不存在
“r”（只读）	为了输入数据，打开一个已经存在的文本文件	出错
“w”（只写）	为了输出数据，打开一个文本文件	建立一个新的文件
“a”（追加）	向文本文件尾添加数据	建立一个新的文件
“rb”（只读）	为了输入数据，打开一个二进制文件	出错
“wb”（只写）	为了输出数据，打开一个二进制文件	建立一个新的文件
“ab”（追加）	向一个二进制文件尾添加数据	出错
“r+”（读写）	为了读和写，打开一个文本文件	出错
“w+”（读写）	为了读和写，建议一个新的文件	建立一个新的文件
“a+”（读写）	打开一个文件，在文件尾进行读写	建立一个新的文件
“rb+”（读写）	为了读和写打开一个二进制文件	出错
“wb+”（读写）	为了读和写，新建一个新的二进制文件	建立一个新的文件
“ab+”（读写）	打开一个二进制文件，在文件尾进行读和写	建立一个新的文件

实例代码：

```c
/* fopen fclose example */
#include <stdio.h>
int main ()
{
  FILE * pFile;
  //打开文件
  pFile = fopen ("myfile.txt","w");
  //文件操作
  if (pFile!=NULL)
  {
    fputs ("fopen example",pFile);
    //关闭文件
    fclose (pFile);
  }
  return 0;
}
```

4. 文件的顺序读写

功能	函数名	适用于
字符输入函数	fgetc	所有输入流
字符输出函数	fputc	所有输出流
文本行输入函数	fgets	所有输入流
文本行输出函数	fputs	所有输出流
格式化输入函数	fscanf	所有输入流
格式化输出函数	fprintf	所有输出流
二进制输入	fread	文件
二进制输出	fwrite	文件

4.1 对比一组函数：

> scanf/fscanf/sscanf
>
> printf/fprintf/sprintf

这里演示讲解这句函数的使用和对比

5. 文件的随机读写

5.1 fseek

> 根据文件指针的位置和偏移量来定位文件指针。

```
int fseek ( FILE * stream, long int offset, int origin );
```

例子：

```
/* fseek example */
#include <stdio.h>

int main ()
{
  FILE * pFile;
  pFile = fopen ( "example.txt" , "wb" );
  fputs ( "This is an apple." , pFile );
  fseek ( pFile , 9 , SEEK_SET );
  fputs ( " sam" , pFile );
  fclose ( pFile );
  return 0;
}
```

5.2 ftell

> 返回文件指针相对于起始位置的偏移量

```
long int ftell ( FILE * stream );
```

例子:

```
/* ftell example : getting size of a file */
#include <stdio.h>

int main ()
{
  FILE * pFile;
  long size;

  pFile = fopen ("myfile.txt","rb");
  if (pFile==NULL) perror ("Error opening file");
  else
  {
    fseek (pFile, 0, SEEK_END);   // non-portable
    size=ftell (pFile);
    fclose (pFile);
    printf ("Size of myfile.txt: %ld bytes.\n",size);
  }
  return 0;
}
```

5.3 rewind

> 让文件指针的位置回到文件的起始位置

```
void rewind ( FILE * stream );
```

例子:

```
/* rewind example */
#include <stdio.h>

int main ()
{
  int n;
  FILE * pFile;
  char buffer [27];

  pFile = fopen ("myfile.txt","w+");
  for ( n='A' ; n<='Z' ; n++)
    fputc ( n, pFile);
  rewind (pFile);
  fread (buffer,1,26,pFile);
  fclose (pFile);
  buffer[26]='\0';
  puts (buffer);
```

```
    return 0;
}
```

6. 文本文件和二进制文件

根据数据的组织形式，数据文件被称为**文本文件**或者**二进制文件**。

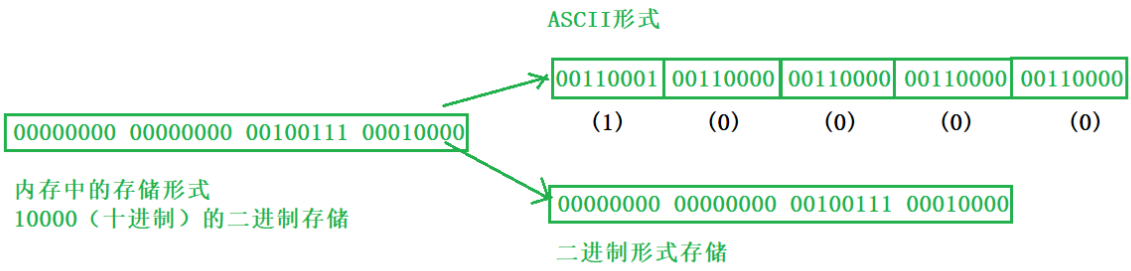
数据在内存中以二进制的形式存储，如果不加转换的输出到外存，就是**二进制文件**。

如果要求在外存上以ASCII码的形式存储，则需要在存储前转换。以ASCII字符的形式存储的文件就是**文本文件**。

一个数据在内存中是怎么存储的呢?

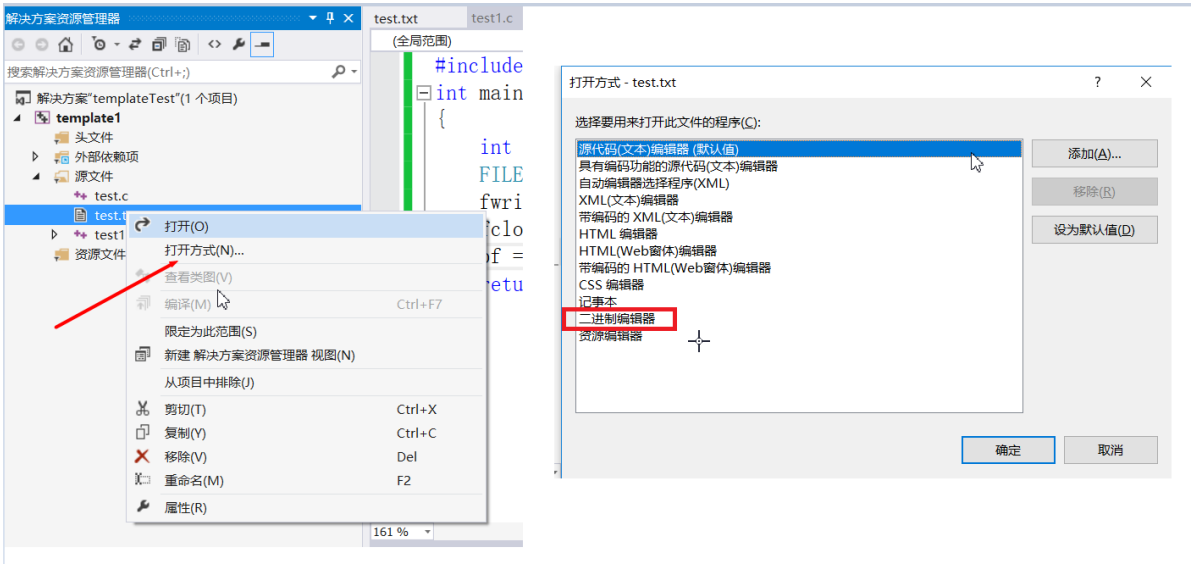
字符一律以ASCII形式存储，数值型数据既可以用ASCII形式存储，也可以使用二进制形式存储。

如有整数10000，如果以ASCII码的形式输出到磁盘，则磁盘中占用5个字节（每个字符一个字节），而二进制形式输出，则在磁盘上只占4个字节（VS2013测试）。



测试代码：

```
#include <stdio.h>
int main()
{
    int a = 10000;
    FILE* pf = fopen("test.txt", "wb");
    fwrite(&a, 4, 1, pf);//二进制的形式写到文件中
    fclose(pf);
    pf = NULL;
    return 0;
}
```



7. 文件读取结束的判定

7.1 被错误使用的`feof`

牢记：在文件读取过程中，不能用feof函数的返回值直接用来判断文件的是否结束。

而是**应用于当文件读取结束的时候，判断是读取失败结束，还是遇到文件尾结束。**

1. 文本文件读取是否结束，判断返回值是否为`EOF`（`fgetc`），或者`NULL`（`fgets`）

   例如：

   - `fgetc`判断是否为`EOF`.
   - `fgets`判断返回值是否为`NULL`.
2. 二进制文件的读取结束判断，判断返回值是否小于实际要读的个数。

   例如：

   - fread判断返回值是否小于实际要读的个数。

正确的使用：

文本文件的例子：

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

int main(void)
{
    int c; // 注意：int，非char，要求处理EOF
    FILE* fp = fopen("test.txt", "r");
    if(!fp) {
        perror("File opening failed");
        return EXIT_FAILURE;
    }
    //fgetc 当读取失败的时候或者遇到文件结束的时候，都会返回EOF
    while ((c = fgetc(fp)) != EOF) // 标准C I/O读取文件循环
    {
       putchar(c);
    }
```

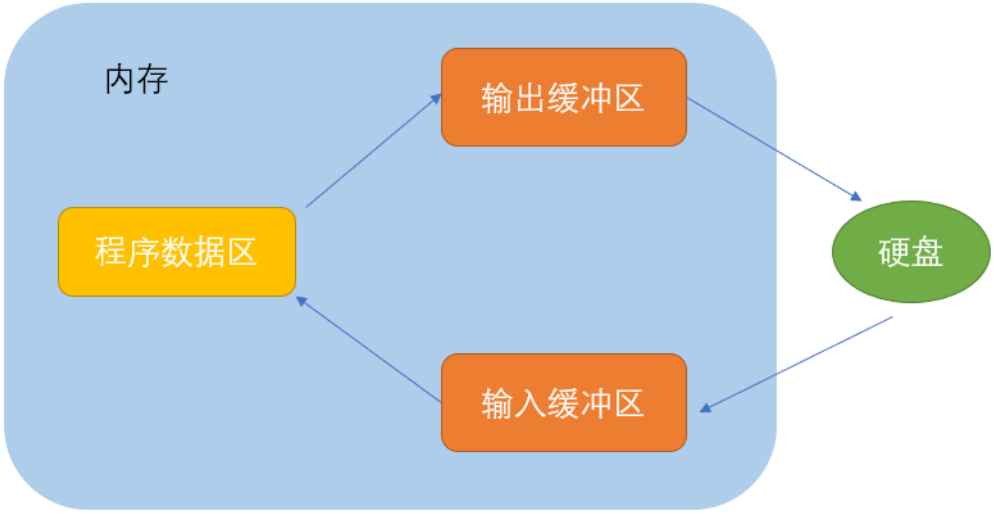
```
    //判断是什么原因结束的
    if (ferror(fp))
        puts("I/O error when reading");
    else if (feof(fp))
        puts("End of file reached successfully");

    fclose(fp);
}
```

二进制文件的例子：

```
#include <stdio.h>

enum { SIZE = 5 };
int main(void)
{
    double a[SIZE] = {1.,2.,3.,4.,5.};
    FILE *fp = fopen("test.bin", "wb"); // 必须用二进制模式
    fwrite(a, sizeof *a, SIZE, fp); // 写 double 的数组
    fclose(fp);

    double b[SIZE];
    fp = fopen("test.bin","rb");
    size_t ret_code = fread(b, sizeof *b, SIZE, fp); // 读 double 的数组
    if(ret_code == SIZE) {
        puts("Array read successfully, contents: ");
        for(int n = 0; n < SIZE; ++n) printf("%f ", b[n]);
        putchar('\n');
    } else { // error handling
       if (feof(fp))
          printf("Error reading test.bin: unexpected end of file\n");
       else if (ferror(fp)) {
           perror("Error reading test.bin");
       }
    }

    fclose(fp);
}
```

8. 文件缓冲区

ANSIC 标准采用**“缓冲文件系统”**处理的数据文件的，所谓缓冲文件系统是指系统自动地在内存中为程序中每一个正在使用的文件开辟一块**“文件缓冲区”**。从内存向磁盘输出数据会先送到内存中的缓冲区，装满缓冲区后才一起送到磁盘上。如果从磁盘向计算机读入数据，则从磁盘文件中读取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（程序变量等）。缓冲区的大小根据C编译系统决定的。




```c
#include <stdio.h>
#include <windows.h>
//VS2013 WIN10环境测试
int main()
{
    FILE*pf = fopen("test.txt", "w");
    fputs("abcdef", pf);//先将代码放在输出缓冲区
    printf("睡眠10秒-已经写数据了，打开test.txt文件，发现文件没有内容\n");
    Sleep(10000);
    printf("刷新缓冲区\n");
    fflush(pf);//刷新缓冲区时，才将输出缓冲区的数据写到文件（磁盘）
    //注：fflush 在高版本的VS上不能使用了
    printf("再睡眠10秒-此时，再次打开test.txt文件，文件有内容了\n");
    Sleep(10000);
    fclose(pf);
    //注：fclose在关闭文件的时候，也会刷新缓冲区
    pf = NULL;

    return 0;
}
```

这里可以得出一个**结论**:

因为有缓冲区的存在，C语言在操作文件的时候，需要做刷新缓冲区或者在文件操作结束的时候关闭文件。

如果不做，可能导致读写文件的问题。

本章完


比特就业课

BIT-7-程序环境和预处理

本章重点：

- 程序的翻译环境
- 程序的执行环境
- 详解：C语言程序的编译+链接
- 预定义符号介绍
- 预处理指令`#define`
- 宏和函数的对比
- 预处理操作符#和##的介绍
- 命令定义
- 预处理指令`#include`
- 预处理指令`#undef`
- 条件编译

---

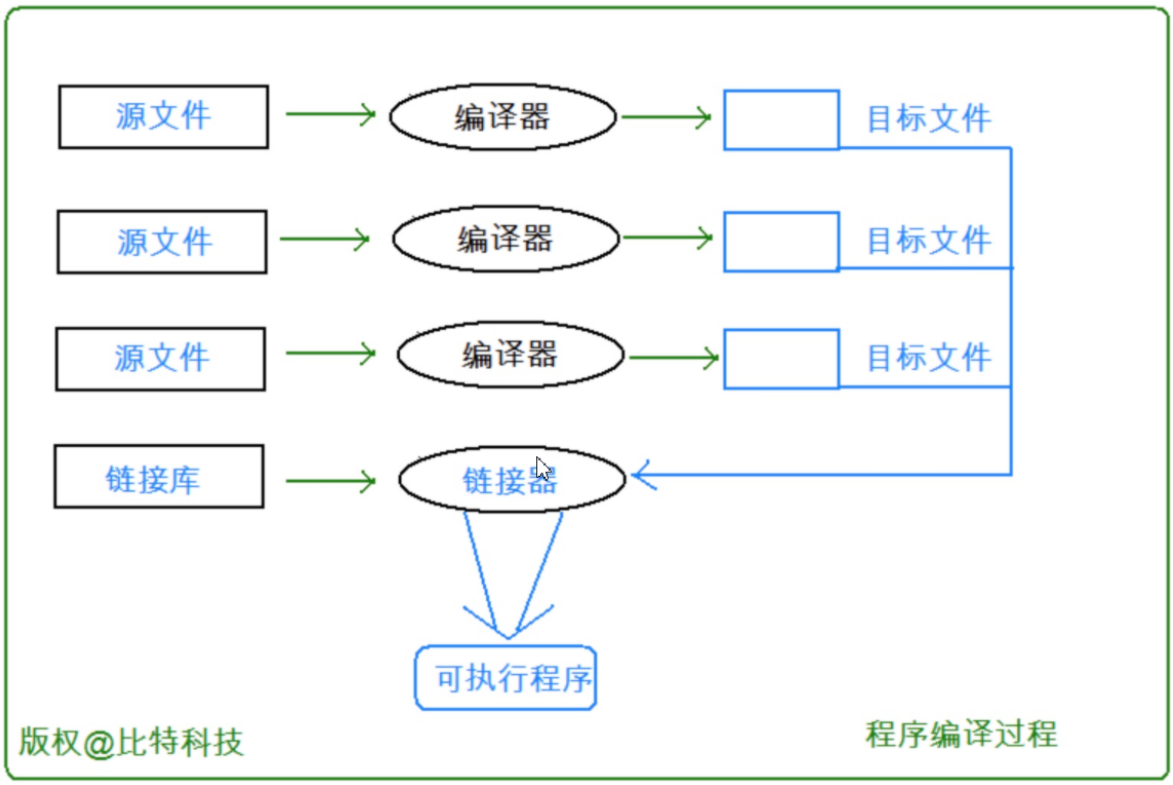
1. 程序的翻译环境和执行环境

在ANSI C的任何一种实现中，存在两个不同的环境。

> 第1种是翻译环境，在这个环境中源代码被转换为可执行的机器指令。
> 第2种是执行环境，它用于实际执行代码。

2. 详解编译+链接

2.1 翻译环境



- 组成一个程序的每个源文件通过编译过程分别转换成目标代码（object code）。
- 每个目标文件由链接器（linker）捆绑在一起，形成一个单一而完整的可执行程序。
- 链接器同时也会引入标准C函数库中任何被该程序所用到的函数，而且它可以搜索程序员个人的程序库，将其需要的函数也链接到程序中。

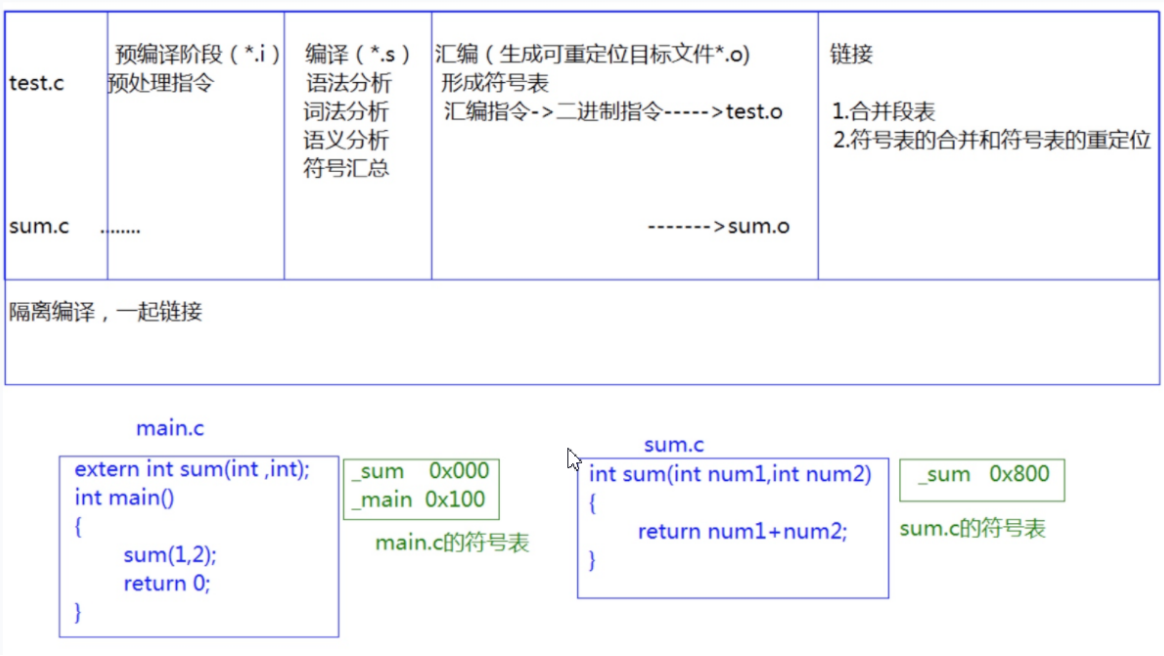
2.2 编译本身也分为几个阶段：

看代码：
sum.c

```
int g_val = 2016;
void print(const char *str)
{
    printf("%s\n", str);
}
```

test.c

```
#include <stdio.h>
int main()
{
    extern void print(char *str);
    extern int g_val;
    printf("%d\n", g_val);
    print("hello bit.\n");
    return 0;
}
```



如何查看编译期间的每一步发生了什么呢?

test.c

```c
#include <stdio.h>
int main()
{
    int i = 0;
    for(i=0; i<10; i++)
    {
        printf("%d ", i);
    }
    return 0;
}
```

1. 预处理 选项`gcc -E test.c -o test.i`
   预处理完成之后就停下来，预处理之后产生的结果都放在test.i文件中。
2. 编译 选项 `gcc -S test.c`
   编译完成之后就停下来，结果保存在test.s中。
3. 汇编 `gcc -c test.c`
   汇编完成之后就停下来，结果保存在test.o中。

> VIM学习资料
>
> 简明VIM练级攻略:
>
> https://coolshell.cn/articles/5426.html
>
> 给程序员的VIM速查卡
>
> https://coolshell.cn/articles/5479.html

2.3 运行环境

程序执行的过程:

1. 程序必须载入内存中。在有操作系统的环境中：一般这个由操作系统完成。在独立的环境中，程序的载入必须由手工安排，也可能是通过可执行代码置入只读内存来完成。
2. 程序的执行便开始。接着便调用main函数。

3. 开始执行程序代码。这个时候程序将使用一个运行时堆栈（stack），存储函数的局部变量和返回地址。程序同时也可以使用静态（static）内存，存储于静态内存中的变量在程序的整个执行过程一直保留他们的值。
4. 终止程序。正常终止main函数；也有可能是意外终止。

注：
介绍一本书《程序员的自我修养》

3. 预处理详解

3.1 预定义符号

```
__FILE__       //进行编译的源文件
__LINE__      //文件当前的行号
__DATE__     //文件被编译的日期
__TIME__     //文件被编译的时间
__STDC__     //如果编译器遵循ANSI C，其值为1，否则未定义
```

这些预定义符号都是语言内置的。
举个栗子：

```
printf("file:%s line:%d\n", __FILE__, __LINE__);
```

3.2 #define

3.2.1 #define 定义标识符

```
语法：
    #define name  stuff
```

举个栗子：

```
#define MAX 1000
#define reg register           //为 register这个关键字，创建一个简短的名字
#define do_forever for(;;)      //用更形象的符号来替换一种实现
#define CASE break;case         //在写case语句的时候自动把 break写上。
// 如果定义的 stuff过长，可以分成几行写，除了最后一行外，每行的后面都加一个反斜杠(续行符)。
#define DEBUG_PRINT printf("file:%s\tline:%d\t \
                          date:%s\ttime:%s\n" ,\
                          __FILE__,__LINE__ ,       \
                          __DATE__,__TIME__ )
```

提问：

> 在define定义标识符的时候，要不要在最后加上;？

比如：

```
#define MAX 1000;
#define MAX 1000
```

建议不要加上`;`,这样容易导致问题。
比如下面的场景：

```
if(condition)
    max = MAX;
else
    max = 0;
```

这里会出现语法错误。

3.2.2 #define 定义宏

> #define 机制包括了一个规定，允许把参数替换到文本中，这种实现通常称为宏（macro）或定义宏（define macro）。

下面是宏的申明方式：

`#define name( parament-list ) stuff`
其中的`parament-list`是一个由逗号隔开的符号表，它们可能出现在stuff中。

注意：
参数列表的左括号必须与name紧邻。
如果两者之间有任何空白存在，参数列表就会被解释为stuff的一部分。

如：

```
#define SQUARE( x )  x * x
```

这个宏接收一个参数`x`.
如果在上述声明之后，你把

```
SQUARE( 5 );
```

置于程序中，预处理器就会用下面这个表达式替换上面的表达式：

```
5 * 5
```

警告：
这个宏存在一个问题：
观察下面的代码段：

```
int a = 5;
printf("%d\n" ,SQUARE( a + 1) );
```

乍一看，你可能觉得这段代码将打印36这个值。
事实上，它将打印11.
为什么?

> 替换文本时，参数x被替换成a + 1,所以这条语句实际上变成了：
> printf ("%d\n",a + 1 * a + 1);

这样就比较清晰了，由替换产生的表达式并没有按照预想的次序进行求值。

在宏定义上加上两个括号，这个问题便轻松的解决了：

```
#define SQUARE(x)  (x) * (x)
```

这样预处理之后就产生了预期的效果：

```
printf ("%d\n",(a + 1) * (a + 1) );
```

这里还有一个宏定义：

```
#define DOUBLE(x) (x) + (x)
```

定义中我们使用了括号，想避免之前的问题，但是这个宏可能会出现新的错误。

```
int a = 5;
printf("%d\n" ,10 * DOUBLE(a));
```

这将打印什么值呢?

warning:
看上去，好像打印100，但事实上打印的是55.
我们发现替换之后：

```
printf ("%d\n",10 * (5) + (5));
```

乘法运算先于宏定义的加法，所以出现了

`55`.

这个问题，的解决办法是在宏定义表达式两边加上一对括号就可以了。

```
#define DOUBLE( x)   ( ( x ) + ( x ) )
```

提示:

> 所以用于对数值表达式进行求值的宏定义都应该用这种方式加上括号，避免在使用宏时由于参数中的操作符或邻近操作符之间不可预料的相互作用。

3.2.3 #define 替换规则

在程序中扩展#define定义符号和宏时，需要涉及几个步骤。

1. 在调用宏时，首先对参数进行检查，看看是否包含任何由#define定义的符号。如果是，它们首先被替换。
2. 替换文本随后被插入到程序中原来文本的位置。对于宏，参数名被他们的值替换。
3. 最后，再次对结果文件进行扫描，看看它是否包含任何由#define定义的符号。如果是，就重复上述处理过程。

注意:

1. 宏参数和#define 定义中可以出现其他#define定义的变量。但是对于宏，不能出现递归。
2. 当预处理器搜索#define定义的符号的时候，字符串常量的内容并不被搜索。

3.2.4 #和##

> 如何把参数插入到字符串中?

首先我们看看这样的代码:

```
char* p = "hello ""bit\n";
printf("hello"," bit\n");
printf("%s", p);
```

这里输出的是不是

`hello bit`?

答案是确定的:是。

我们发现字符串是有自动连接的特点的。

1. 那我们是不是可以写这样的代码?:

```
#define PRINT(FORMAT, VALUE)\
    printf("the value is "FORMAT"\n", VALUE);
...
PRINT("%d", 10);
```

这里只有当字符串作为宏参数的时候才可以把字符串放在字符串中。

1. 另外一个技巧是:
   使用`#`,**把一个宏参数变成对应的字符串。**
   比如:

```
int i = 10;
#define PRINT(FORMAT, VALUE)\
    printf("the value of " #VALUE "is "FORMAT "\n", VALUE);
...
PRINT("%d", i+3);//产生了什么效果?
```

代码中的`#VALUE`会预处理器处理为:

`"VALUE"`.
最终的输出的结果应该是:

```
the value of i+3 is 13
```

的作用

> ##可以把位于它两边的符号合成一个符号。
> 它允许宏定义从分离的文本片段创建标识符。

```
#define ADD_TO_SUM(num, value) \
    sum##num += value;
...
ADD_TO_SUM(5, 10);//作用是:给sum5增加10.
```

注:
这样的连接必须产生一个合法的标识符。否则其结果就是未定义的。

3.2.5 带副作用的宏参数

当宏参数在宏的定义中出现超过一次的时候，如果参数带有副作用，那么你在使用这个宏的时候就可能出现危险，导致不可预测的后果。副作用就是表达式求值的时候出现的永久性效果。
例如：

```
x+1;//不带副作用
x++;//带有副作用
```

MAX宏可以证明具有副作用的参数所引起的问题。

```
#define MAX(a, b)  ( (a) > (b) ? (a) : (b) )
...
x = 5;
y = 8;
z = MAX(x++, y++);
printf("x=%d y=%d z=%d\n", x, y, z);//输出的结果是什么？
```

这里我们得知道预处理器处理之后的结果是什么：

```
z = ( (x++) > (y++) ? (x++) : (y++));
```

所以输出的结果是：

```
x=6 y=10 z=9
```

3.2.6 宏和函数对比

宏通常被应用于执行简单的运算。比如在两个数中找出较大的一个。

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

那为什么不用函数来完成这个任务？
原因有二：

1. 用于调用函数和从函数返回的代码可能比实际执行这个小型计算工作所需要的时间更多。**所以宏比函数在程序的规模和速度方面更胜一筹。**
2. 更为重要的是函数的参数必须声明为特定的类型。所以函数只能在类型合适的表达式上使用。反之这个宏怎可以适用于整形、长整型、浮点型等可以用于>来比较的类型。**宏是类型无关的。**

当然和宏相比函数也有劣势的地方：

1. 每次使用宏的时候，一份宏定义的代码将插入到程序中。除非宏比较短，否则可能大幅度增加程序的长度。
2. 宏是没法调试的。
3. 宏由于类型无关，也就不够严谨。
4. 宏可能会带来运算符优先级的问题，导致程容易出现错。

宏有时候可以做函数做不到的事情。比如：宏的参数可以出现**类型**，但是函数做不到。

```
#define MALLOC(num, type)\
    (type *)malloc(num * sizeof(type))
...
//使用
MALLOC(10, int);//类型作为参数

//预处理器替换之后：
(int *)malloc(10 * sizeof(int));
```

宏和函数的一个对比

属性	#define定义宏	函数
代码长度	每次使用时，宏代码都会被插入到程序中。除了非常小的宏之外，程序的长度会大幅度增长	函数代码只出现于一个地方；每次使用这个函数时，都调用那个地方的同一份代码
执行速度	更快	存在函数的调用和返回的额外开销，所以相对慢一些
操作符优先级	宏参数的求值是在所有周围表达式的上下文环境里，除非加上括号，否则邻近操作符的优先级可能会产生不可预料的后果，所以建议宏在书写的时候多些括号。	函数参数只在函数调用的时候求值一次，它的结果值传递给函数。表达式的求值结果更容易预测。
带有副作用的参数	参数可能被替换到宏体中的多个位置，所以带有副作用的参数求值可能会产生不可预料的结果。	函数参数只在传参的时候求值一次，结果更容易控制。
参数类型	宏的参数与类型无关，只要对参数的操作是合法的，它就可以使用于任何参数类型。	函数的参数是与类型有关的，如果参数的类型不同，就需要不同的函数，即使他们执行的任务是不同的。
调试	宏是不方便调试的	函数是可以逐语句调试的
递归	宏是不能递归的	函数是可以递归的

命名约定

一般来讲函数的宏的使用语法很相似。所以语言本身没法帮我们区分二者。
那我们平时的一个习惯是：

> 把宏名全部大写
> 函数名不要全部大写

3.3 #undef

这条指令用于移除一个宏定义。

```
#undef NAME
//如果现存的一个名字需要被重新定义，那么它的旧名字首先要被移除。
```

3.4 命令行定义

许多C 的编译器提供了一种能力，允许在命令行中定义符号。用于启动编译过程。
例如：当我们根据同一个源文件要编译出不同的一个程序的不同版本的时候，这个特性有点用处。（假定某个程序中声明了一个某个长度的数组，如果机器内存有限，我们需要一个很小的数组，但是另外一个机器内存大写，我们需要一个数组能够大写。）

```
#include <stdio.h>
int main()
{
    int array [ARRAY_SIZE];
    int i = 0;
    for(i = 0; i< ARRAY_SIZE; i ++)
    {
        array[i] = i;
    }
    for(i = 0; i< ARRAY_SIZE; i ++)
    {
        printf("%d " ,array[i]);
    }
    printf("\n" );
    return 0;
}
```

编译指令：

```
gcc -D ARRAY_SIZE=10 programe.c
```

3.5 条件编译

在编译一个程序的时候我们如果要将一条语句（一组语句）编译或者放弃是很方便的。因为我们有条件编译指令。

比如说：

> 调试性的代码，删除可惜，保留又碍事，所以我们可以选择性的编译。

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

```
int main()
{
    int i = 0;
    int arr[10] = {0};
    for(i=0; i<10; i++)
    {
        arr[i] = i;
        #ifdef __DEBUG__
        printf("%d\n", arr[i]);//为了观察数组是否赋值成功。
        #endif //__DEBUG__
    }
    return 0;
}
```

常见的条件编译指令：

```
1.
#if  常量表达式
    //...
#endif
//常量表达式由预处理器求值。
如：
#define __DEBUG__ 1
#if __DEBUG__
    //..
#endif

2.多个分支的条件编译
#if 常量表达式
    //...
#elif 常量表达式
    //...
#else
    //...
#endif

3.判断是否被定义
#if defined(symbol)
#ifdef symbol

#if !defined(symbol)
#ifndef symbol

4.嵌套指令
#if defined(OS_UNIX)
    #ifdef OPTION1
        unix_version_option1();
    #endif
    #ifdef OPTION2
        unix_version_option2();
    #endif
#elif defined(OS_MSDOS)
    #ifdef OPTION2
        msdos_version_option2();
    #endif
#endif
```

3.6 文件包含

我们已经知道，`#include` 指令可以使另外一个文件被编译。就像它实际出现于`#include`指令的地方一样。

这种替换的方式很简单：
预处理器先删除这条指令，并用包含文件的内容替换。
这样一个源文件被包含10次，那就实际被编译10次。

3.6.1 头文件被包含的方式：

- 本地文件包含

```
#include "filename"
```

查找策略：先在源文件所在目录下查找，如果该头文件未找到，编译器就像查找库函数头文件一样在标准位置查找头文件。
如果找不到就提示编译错误。

Linux环境的标准头文件的路径：

```
/usr/include
```

VS环境的标准头文件的路径：

```
C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\include
```

注意按照自己的安装路径去找。

- 库文件包含

```
#include <filename.h>
```

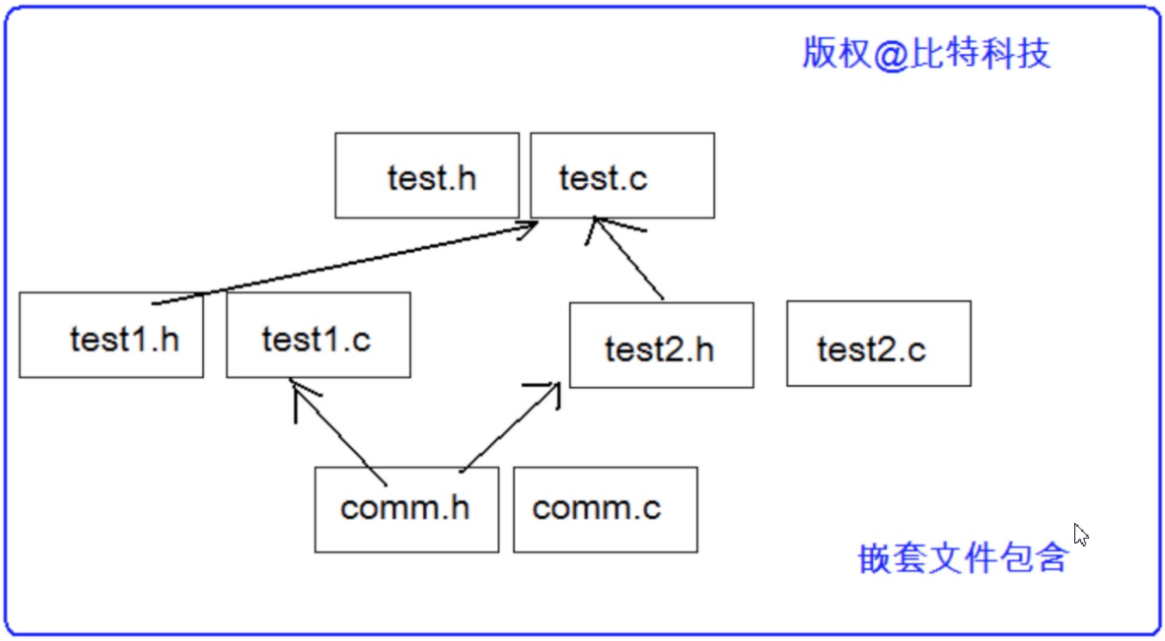
查找头文件直接去标准路径下去查找，如果找不到就提示编译错误。

这样是不是可以说，对于库文件也可以使用“”的形式包含？
答案是肯定的，**可以。**

但是这样做查找的效率就低些，当然这样也不容易区分是库文件还是本地文件了。

3.6.2 嵌套文件包含

如果出现这样的场景：



comm.h和comm.c是公共模块。
test1.h和test1.c使用了公共模块。
test2.h和test2.c使用了公共模块。
test.h和test.c使用了test1模块和test2模块。
这样最终程序中就会出现两份comm.h的内容。这样就造成了文件内容的重复。

如何解决这个问题?
答案：条件编译。

每个头文件的开头写：

```
#ifndef __TEST_H__
#define __TEST_H__
//头文件的内容
#endif   //__TEST_H__
```

或者：

```
#pragma once
```

就可以避免头文件的重复引入。

注:
推荐《高质量C/C++编程指南》中附录的考试试卷（很重要）。
笔试题：

```
1. 头文件中的 ifndef/define/endif是干什么用的?
2. #include <filename.h> 和 #include "filename.h"有什么区别?
```

4. 其他预处理指令

```
#error
#pragma
#line
...
不做介绍，自己去了解。

#pragma pack()在结构体部分介绍。
```

参考《C语言深度解剖》学习

本章完

鹏哥的代码在码云，此链接是鹏哥的码云地址：https://gitee.com/bitpg/bit-c-code