@[toc]

第11次直播 指针数组与malloc

易错三大例子

例子一:

```c
#include <stdio.h>

int main()
{
    int a[3] = { 2,7,8 };
    int* p;
    int j;
    p = a;//让指针变量p，指向数组的开头
    j = *p++;//j=*p;p++,任何时候都是把后加加去掉，第二步另外一个运算符看优先
    printf("a[0]=%d, j=%d, *p=%d\n", a[0], j, *p);
    return 0;
}
```


上述正确的输出为： 2 2 7

例子二:

```c
#include <stdio.h>

int main()
{
    int a[3] = { 2,7,8 };
    int* p;
    int j;
    p = a;//让指针变量p，指向数组的开头
    j = (*p)++;//j=*p;(*p)++,任何时候都是把后加加去掉，第二步另外一个运算符
    printf("a[0]=%d, j=%d, *p=%d\n", a[0], j, *p);
    return 0;
}
```


上述正确的输出为： 3 2 3

例子三:

```c
int main()
{
    int a[3] = { 2,7,8 };
    int* p;
    int j;
    p = a;//让指针变量p，指向数组的开头
    j = *p++;//j=*p;(*p)++,任何时候都是把后加加去掉，第二步另外一个运算符看
    printf("a[0]=%d,j=%d,*p=%d\n", a[0], j, *p);//2 2 7
    j = p[0]++;//j=p[0];p[0]++;
    printf("a[0]=%d,j=%d,*p=%d\n", a[0], j, *p);
    return 0;
}
```


上述正确的输出为：2 7 8

指针与一维数组

例子一：

```c
#include <stdio.h>

void change(char *d)
{
    *d = 'H';
}
int main()
{
    char c[10] = "hello";
    change(c);
    puts(c);
    return 0;
}
```


输出结果为： Hello

- 数组名作为实参传递给子函数时，是弱化为指针的

例子二：

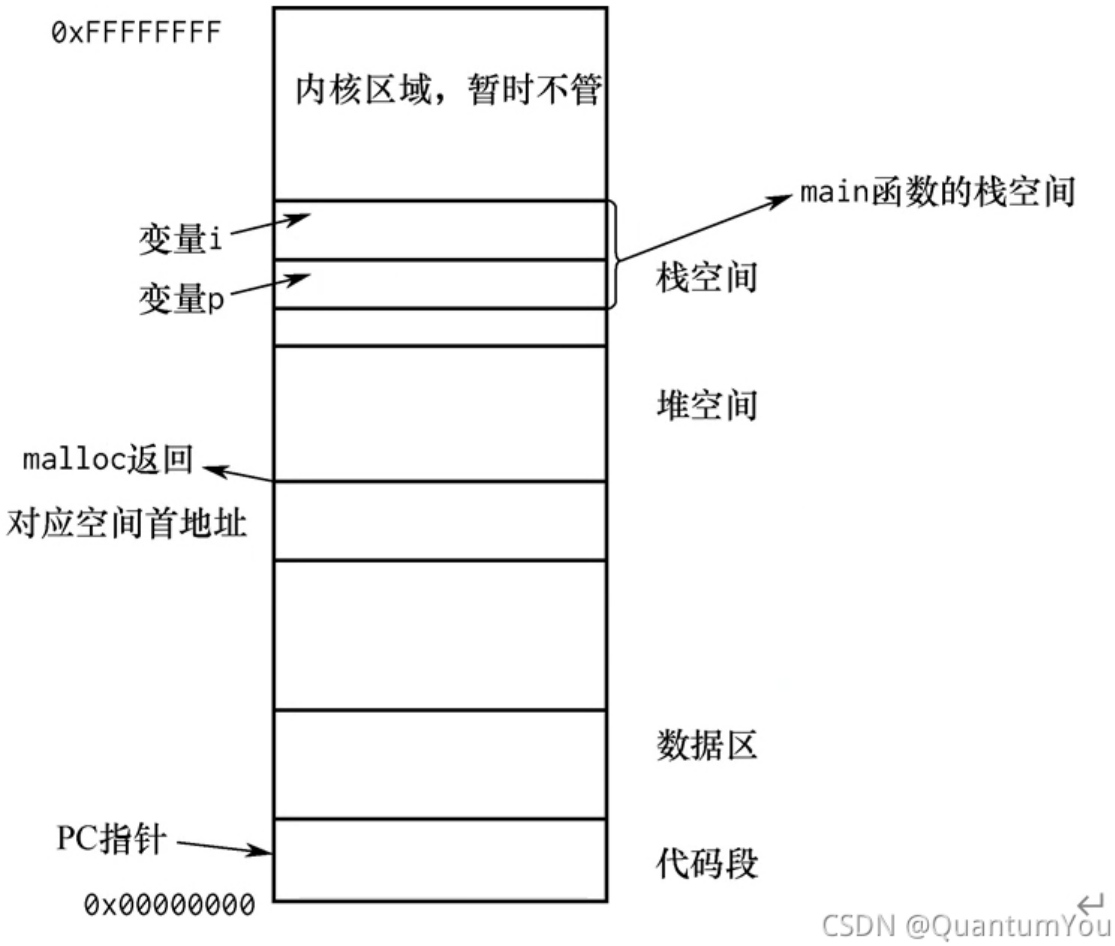
```c
void change(char *d)
{
    *d = 'H';
    d[1] = 'E';
    *(d + 2) = 'L';
}
int main()
{
    char c[10] = "hello";
    change(c);
    puts(c);
```


关问题

指针与动态内存申请

- 很多读者在学习C语言的数组后都会觉得数组长度固定很不方便，其实C语言的数组长度固定是因为其定义的**整型、浮点型、字符型变量**、数组变量都在栈空间中，而栈空间的大小在编译时是确定的。如果使用的空间大小不确定，那么就要使用**堆空间**。
- 既然都是**内存空间**，为什么还要分栈空间和堆空间呢？栈是计算机系统提供的数据结构，计算机会在底层对栈提供支持：分配专门的寄存器存放栈的地址，压栈操作、出栈操作都有专门的旨令执行，这就决定了栈的效率比较高；堆则是CC艹函数库提供的数据结构，它的机制很复杂，例如为了分配一块内存，库函数会按照一定的算法（具体的算法请参考关于数据结构、操作系统的书籍）在堆内存中搜索可用的足够大小的空间，如果没有足够大小的空间（可能由于内存碎片太多），那么就有可能调用系统功能去增加程序数据段的内存空间，这样就有机会分到足够大小的内存，然后返回。显然，堆的效率要比栈低得多栈空间由系统自动管理，而堆空间的申请和释放需要自行管理，所以在具体例子中需要通过`free`函数释放堆空间。`free`函数的头文件及格式为

进程地址空间如下



- 首先进行解析 `malloc` 在 `# include< stdlib. h>` 头文件中， 函数的定义为`void* malloc(size_ t size)` `void *` 表示定义的为无类型指针，因为是无类型所以才使用强制类型转换，在前面加上 `(char *)`，`malloc` 申请空间的单位是字节

易错点： 指针在释放的时候发生偏移

- 释放空间，p的值必须和最初 `malloc` 返回的值一致，如果发生偏移则会有下下图的报错

```c
int main()
{
    int i;//申请多大的空间
    scanf("%d", &i);
    char* p;
    p = (char*)malloc(i);//malloc申请空间的单位是字节
    strcpy(p, "malloc success");
    p++;
    puts(p);
    free(p);//释放空间

    return 0;
}
```

报错如下：

`free(p) ; p=NULL ;` 释放操作

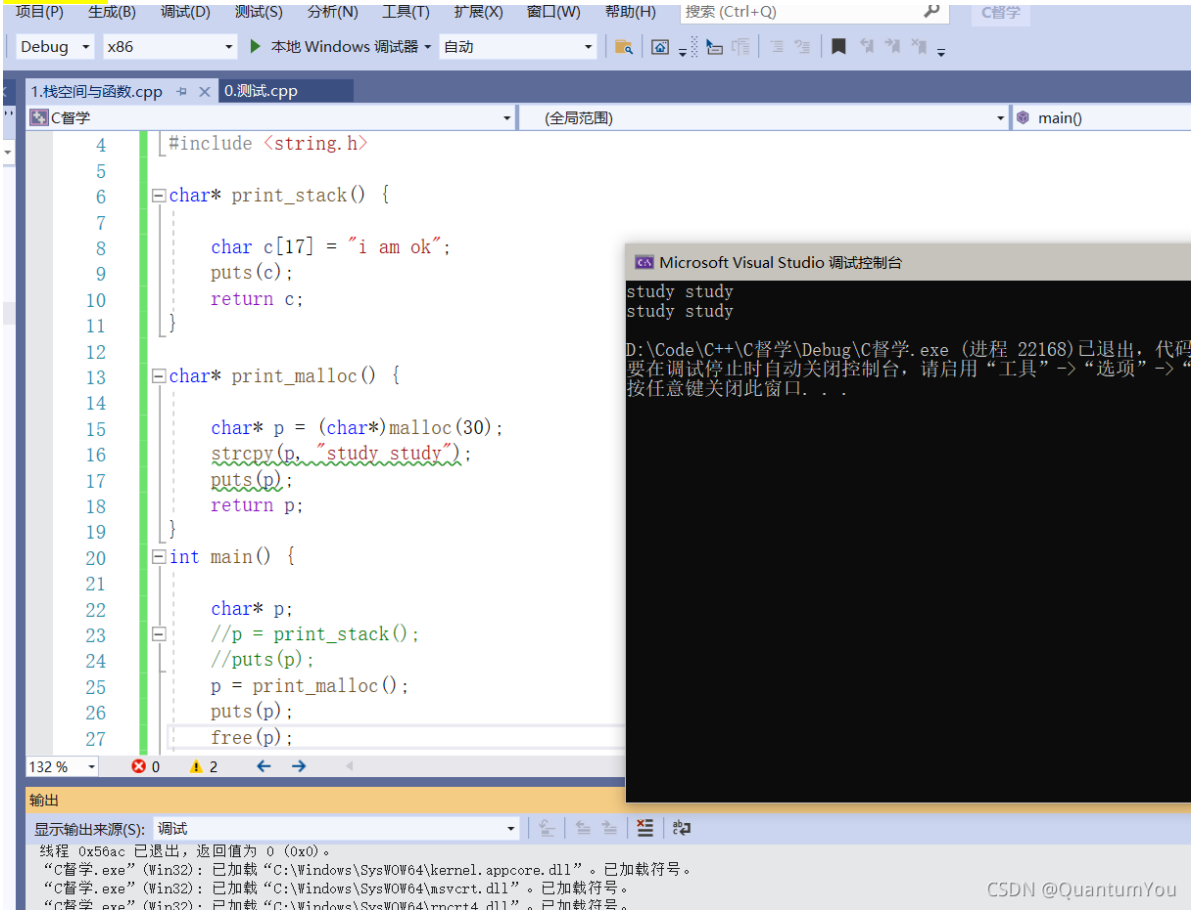
栈空间与堆空间的差异

问题引入：

```c
#include  <stdio.h>

char* print_stack() {
    char c[17] = "i am ok";
    puts(c);
    return c;
}

int main() {

    char* p;
    p = print_stack();
    puts(p);
    return 0;
}
```

- 原因在在于函数是栈空间，栈空间会随着函数结束而释放。
- 不能使指针变量指向栈空间，进行操作，因为栈空间会在之后进行释放

解决方法



- 堆空间不会随子函数的结束而释放，必须自己free

第12次直播 函数与指针

字符指针与字符数组的初始化（了解）

- 字符指针可以初始化赋值一个字符串，字符数组初始化也可以赋值一个字符串。`char *p="hello"`和`char c[10]="hello"`有什么区别呢?

易错分析一：

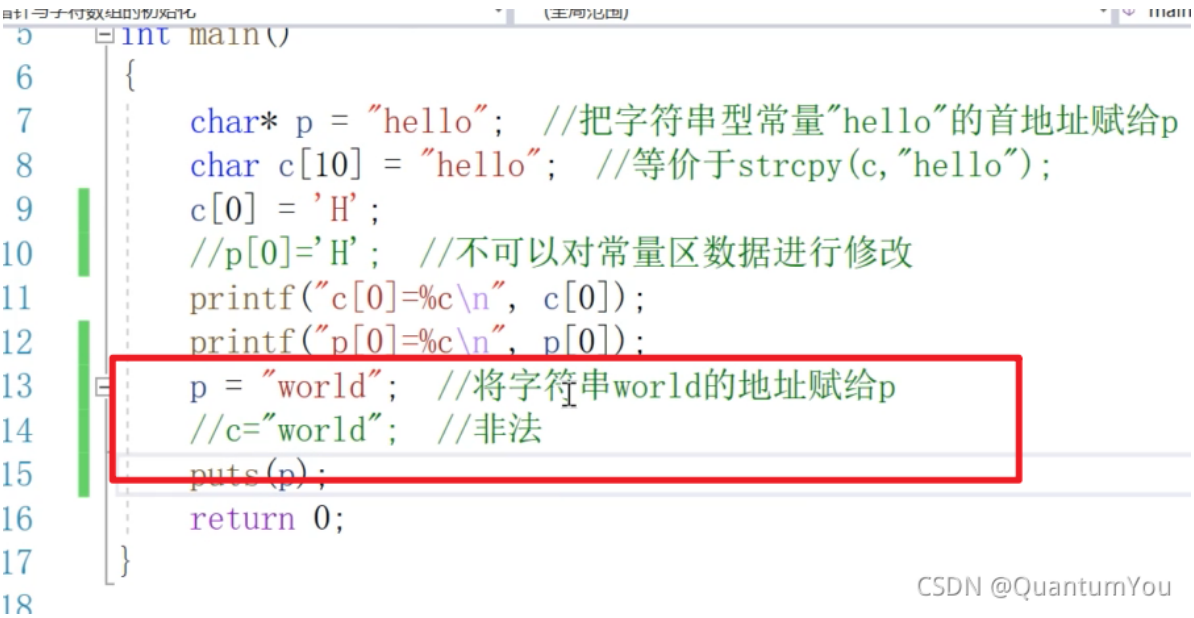
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    char *p="hello";   //把字符串型常量"hello"的首地址赋给 p
    char c[10]="hello";   //等价于 strcpy(c,"hello");
    c[0]='H';
    printf("c[0]=%c\n",c[0]);
    printf("p[0]=%c\n",p[0]);
    //p[0]='H';   //不可以对常量区数据进行修改
    p="world";   //将字符串 world 的地址赋给 p
    //c="world";   //非法
    system("pause");
    return 0;
}
```


错误分析：

- 对`p[0]`进行修改，会报如下错误，然而`c[0]`对数组进行修改可以，因为`char c [10] = "hello"`实际等价于`strcpy(c,"hello");`操作的是堆区（可读可写），`p[0]`实际操作的是字符串常量区（数据区），该区域只读不能写

原因在于：不能对常量区数据进行修改

易错分析二：

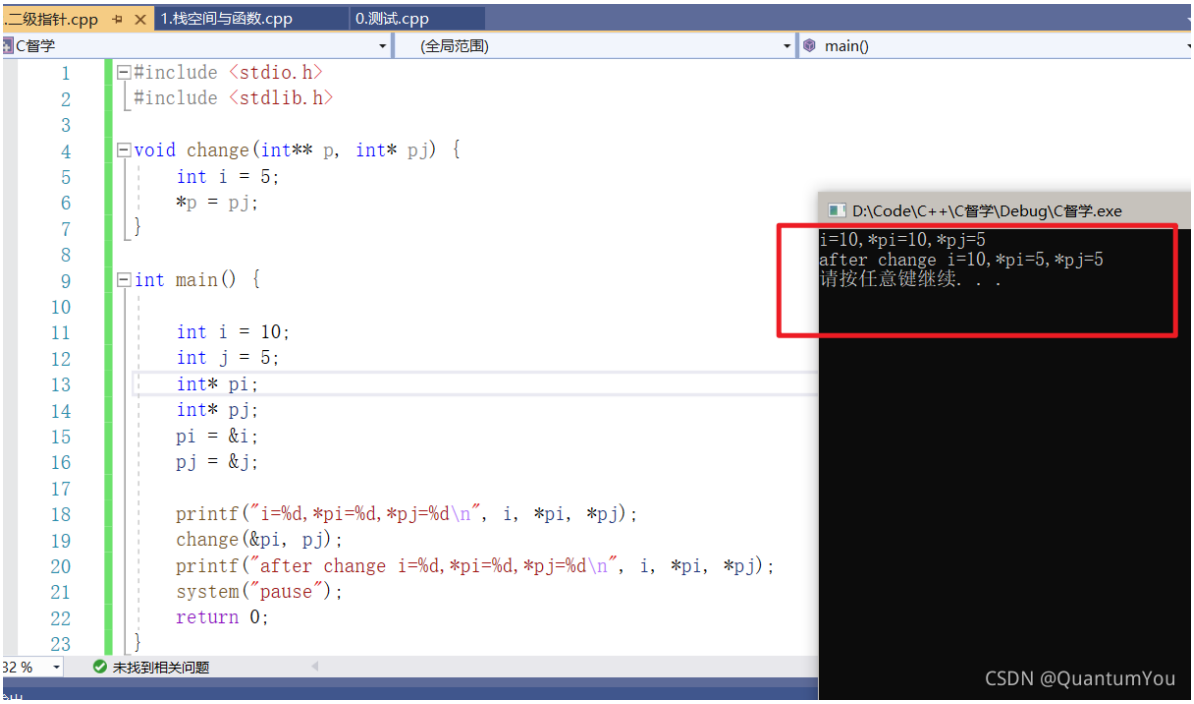


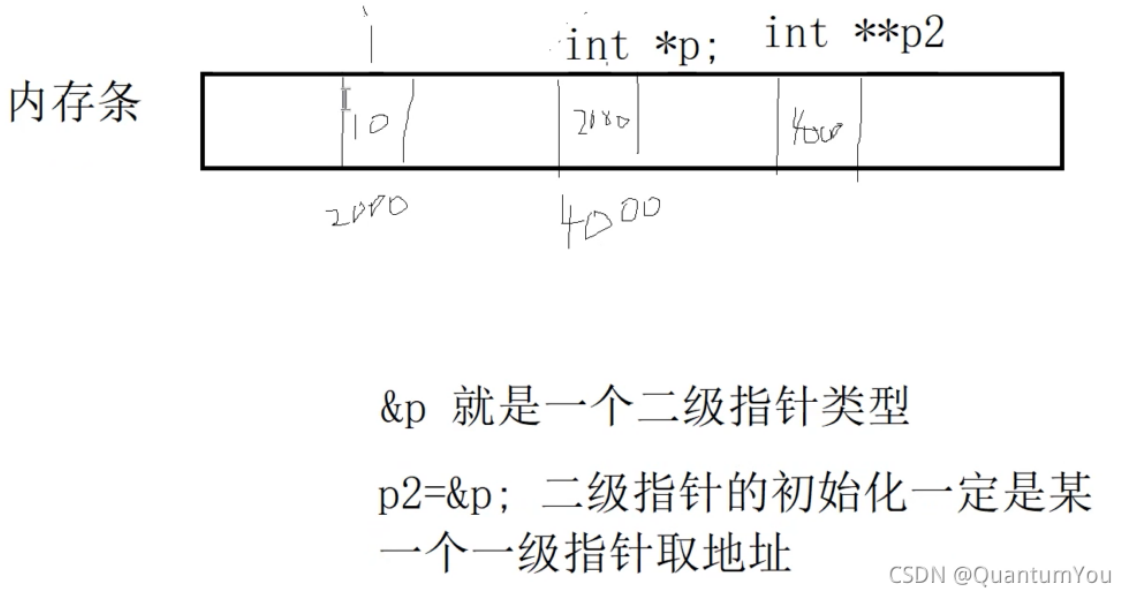
第12次直播 二级指针 结构体

- 二级指针只服务于一级指针的传递与偏移
- 要想在子函数中改变一个变量的值，必须把该**变量的地址**传进去
- 要想在子函数中改变一个指针变量的值，必须把该**指针变量的地址**传进去

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

void change(int** p, int* pj) {
    int i = 5;
    *p = pj;
}

int main() {

```

```
    int i = 10;
    int j = 5;
    int* pi;
    int* pj;
    pi = &i;
    pj = &j;

    printf("i=%d,*pi=%d,*pj=%d\n", i, *pi, *pj);
    change(&pi, pj);
    printf("after change i=%d,*pi=%d,*pj=%d\n", i, *pi, *pj);
    system("pause");
    return 0;
}
```



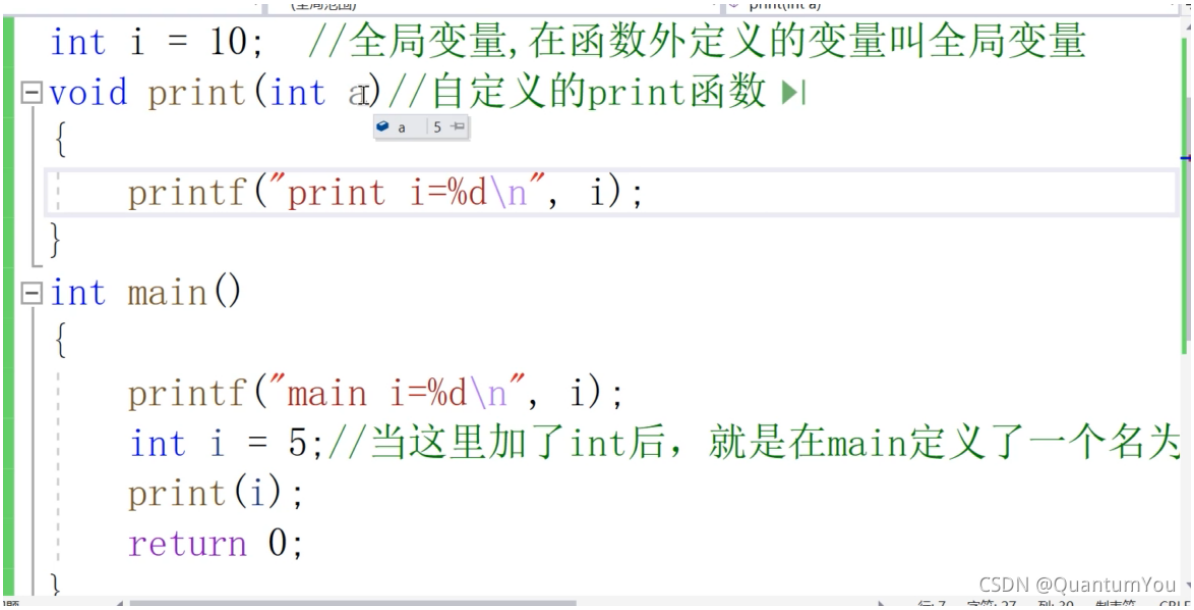


微软的C动态库 后缀为 `dll`

> 注意区分函数的声明与函数的定义：

链接错误 `main.obj`

关于全局变量与局部变量



输出结果为： 10 5

```c
#include  <stdio.h>
// 定义全局变量
int i = 10;
void print(int a) {
    printf("print i =%d\n", i);
}

int main() {
    printf("main i =%d\n", i);
    i = 5;    //局部变量
    print(i);
    return 0;
}
```

输出结果为 10 10

- 这是因为全局变量存储于数据段中（全局变量定义存储于数据段）

结构体

- 结构体所占的空间，并不是简单的里面所包含的数据类型容量简单相加，因为存在对齐策略，会比预期要大

结构体指针（重要）

易错分析一：

```
4   //结构体指针
5   struct student {
6       int num;
7       char name[20];
8       char sex;
9   };
10
11  int main()
12  {
13      struct student s = { 1001,"wangle",'M' };
14      struct student* p;
15      p = &s;
16      printf("%d %s %c\n", *p.num, *p.name, *p.sex);
17      return 0;
```

CSDN @QuantumYou

- 正确写法为： `(*p).num ,(*p).name,(*p).sex`
- `.` 的优先级比 `*` 高
- 优化写法： `p->num  p->name p->sex`

关于结构体的偏移：

```
//结构体初始化
struct student sarr[3] = { 1001,"lilei",'M',1005,"zhangsan",'M
int num;
p = sarr;
printf("----------------------------\n");
num = p->num++;//num=p->num;p->num++
printf("num=%d,p->num=%d\n", num, p->num);//1001,1002
num = p++->num;//num=p->num;p++;
printf("num=%d,p->num=%d\n", num, p->num);//1002,1005
```