



目录



- 变量基本类型
 - 变量内存大小
 - 无符号类型
 - 变量定义
- 

变量基本类型

- ◆ 有哪些基本变量？

char, short, int, long, float, double

变量内存大小

- 请说明变量占用内存大小(32位系统)
`char, short, int, long, float, double`

`char(1), short(2), int(4), long(4), float(4), double(8)`

数据模型相关

无符号类型

- 请说明，基本变量类型，哪些是可以加unsigned的，哪些不能？

可以加: char, short, int, long

不可以: float, double

变量定义

- 请说明，下面定义的变量是什么类型？

```
int *p1;  
const int *p2;  
int const *p3;  
int *const p4 = 1个地址;  
const int *const p5 = 1个地址;  
int p6[2];  
int *p7, p8;
```

基本数据类型的字节长度---数据模型

数据模型： LP32 ILP32 LP64 LLP64 ILP64

- 32位环境涉及"ILP32"数据模型，是因为C数据类型为32位的int、long、指针。
- 64位环境使用不同的数据模型，此时的long和指针已为64位，故称作"LP64"数据模型。
- 现今所有64位的类Unix平台均使用LP64数据模型，而64位Windows使用LLP64数据模型，除了指针是64位，其他基本类型都没有变。

参考：<https://www.cnblogs.com/lsgxeva/p/7614856.html>

TYPE	LP32	ILP32	LP64	ILP64	LLP64
CHAR	8	8	8	8	8
SHORT	16	16	16	16	16
INT	16	32	32	64	32
LONG	32	32	64	64	32
LONG LONG	64	64	64	64	64
POINTER	32	32	64	64	64

```
#编译选项，切勿随意修改
CFLAGS += -mabi=lp64 -O2 -g -fno-common -freg-struct-return -mbig-endian -march=cortex-a57 -fno-strict-aliasing \
          -fomit-frame-pointer -Werror -Wshadow -Wformat=2 -Wtrigraphs \
          -fno-short-wchar -D_LP64_ \
          -fno-toplevel-reorder -frecord-gcc-switches -Wa,-EB -fno-strict-aliasing
```

32位和64位下基本数据类型的字节长度（典型配置）

数据类型	Windows		Linux	
	32位	64位	32位	64位
char	1	1	1	1
short int	2	2	2	2
int	4	4	4	4
long	4	4	4	8
long long	8	8	8	8
float	4	4	4	4
double	8	8	8	8
long double	8	8	12	16
char*指针类型	4	8	4	8
size_t	4	8	4	8
bool	1	1	1	1

- 左表统计的是各个基本数据数据类型在Window/Linux 32/64位系统下的字节数。
- 对于编程的时候需要注意标红色的部分。

目录

- 取词
- 类型转换(32位和64位的区别)

C语言进阶--表达式



0:10 / 7:46

Note 720P 1.5x



取词

下面哪些编译会出错？

1. int n = 0;
2. int m1 = 3;
3. int m2 = +3;
4. int m3 = -3;
5. int m4 = ++3;
6. int m5 = --3;
7. int m6 = ++n;
8. int m7 = --n;
9. int m8 = +-3;
10. int m9 = -+-3;
11. int m10 = ---n;
12. int m11 = -(-(--n));

下面代码能够编译通过么？

```
int m = 2;  
int n = 4;  
int k = m+++n;
```

输出多少？

C:\Windows\sy
m=3, n=4, k=6

原理：编译器总是尽量多的去匹配操作符！

```
int m = 5;  
int n = 3;  
int k = m+++++-+--n;  
printf("m=%d, n=%d, k=%d", m, n, k);
```



C:\Windows\sy
m=6, n=2, k=7

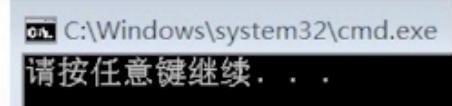
类型自动转换 (1)

下面函数的运行结果是什么？

```
void main()
{
    int i = -1;

    for(; i<sizeof(int); i++)
    {
        printf("%d\n", i);
    }
}
```

运行结果



C:\Windows\system32\cmd.exe
请按任意键继续. . .

结论：

1. `sizeof()`返回`unsigned int`
2. 有符号数与无符号数进行运算，有符号数先自动转换成无符号数。

类型自动转换 (2)

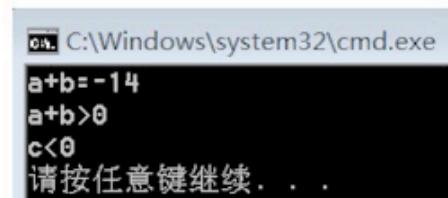
```
void main(void)
{
    unsigned int a=6;
    int b=-20;
    int c=a+b;

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

    if ((a+b) > 0)
        printf("a+b>0\n");
    else
        printf("a+b<=0\n");

    if (c > 0)
        printf("c>0\n");
    else
        printf("c<0\n");
}
```

输出结果是 ?



```
C:\Windows\system32\cmd.exe
a+b=-14
a+b>0
c<0
请按任意键继续. . .
```

类型自动转换 (3)

原则:

1. 参与运算的类型不同，先转换成相同类型再运算。
2. 数据类型向数据长度增长的方向转换，`char->short->int->unsigned int ->long`
3. 所有float都会先转成double进行运算，哪怕只有一个float
4. 赋值运算时，赋值号右边的类型向左边的类型转换。
5. 浮点数和整形数，整形数向浮点数转换。
6. 在表达式中，如果char 和 short 类型的值进行运算，无论char和short有无符号，结果都会自动转换成 int。
7. 如果char或short与int类型进行计算，结果和int类型相同。即：如果int是有符号，结果就是带符号的，如果int是无符号的，结果就是无符号的。

数据类型排序：

long double, double, float, unsigned long long, long long, unsigned long,
long, unsigned int , int, **unsigned short, short, unsigned char, char**

注意：同一类型，有符号数和无符号数运算，有符号数向无符号数转变。

C语言进阶—宏



常用功能 (1) : 常量替换

代码案例:

```
#define MAX_STUDENTS_ONE_CLASS 100  
#define MAX_CLASS 10  
#define MAX_STUDENTS (MAX_STUDENTS_ONE_CLASS*MAX_CLASS)
```

目录

- 常用功能
- 风险说明
- 宏的优点和不足

常用功能 (3) : 函数宏

代码案例:

```
#define MAX(x, y) (((x) > (y))?(x):(y))

#define MIN(x, y) (((x) < (y))?(x):(y))
```

常用功能 (4) : 防止头文件重复包含

代码案例:

```
#ifndef TEST_H
#define TEST_H

#include "xxx.h"

#endif __cplusplus
extern "C" {
#endif

//头文件内容

#ifndef __cplusplus
}
#endif
#endif
```



常用功能 (5) : 内定调试宏

功能描述:

LINE : 当前代码行数
FILE : 当前源文件名
DATE : 当前日期
TIME : 当前时间

代码案例:

```
#define WARING(str) printf("%s, line=%d, file=%s, data=%s,  
time=%s" , str, __LINE__, __FILE__, __DATE__, __TIME__)  
  
int main()  
{  
    WARING("test");  
    return 0;  
}
```

z00114095@CTUY1Z001140951 /var
\$ a.exe
test, line=28, file=main.c, data=Feb 18 2014, time=11:53:37

风险说明（1）：运算优先级引发的问题

代码案例：

```
#define ceil_div(x, y) ((x + y - 1) / y)  
a = ceil_div( b & c, sizeof(int) );
```

问题描述：

```
a = ( b & c + sizeof(int) - 1) / sizeof(int);  
// 由于+/-的优先级高于&的优先级，那么上面式子等同于  
a = ( b & (c + sizeof(int) - 1)) / sizeof(int);
```

解决方案：

```
#define ceil_div((x), (y)) (((x) + (y) - 1) / (y))
```

风险说明（2）：多次运算引发的问题

代码案例：

```
#define min(X,Y) ((X) > (Y) ? (Y) : (X))  
c = min(a,foo(b));
```

问题描述： foo运行了2次，与期待结果不同。

解决方案： 尽量不使用宏函数，而使用普通函数进行处理。除非有其他原因，比如性能问题。

风险说明 (4) : 空格

代码案例:

```
#define func (x) (x+1)  
func(1);
```

问题描述: func后面有空格, func被替换为(x) (x+1),最后展开式:
(x) (x+1)(1);

宏的优点和不足

不足:

1. 宏在符号表中不存在，所以宏函数无法打热补丁，在vs或者单板调试过程中，无法引用
2. 程序的代码空间增大
3. 宏函数单步调试时无法进入
4. 宏常量没有数据类型，编译器不能在编译时进行深入类型检查

优点:

1. 宏函数效率较高
2. 数组大小，只能在编译阶段确定，只能用宏
3. 其他前面讲的功能

C语言进阶一枚举



目录

- 枚举类型的大小
- 枚举的值
- 建议

枚举类型的大小 (1) : 案例说明

```
typedef enum
{
    E1 = 0x1,
    E2,
}ENUM_1;

typedef enum
{
    E3 = 0x123456789,
    E4,
}ENUM_2;

int main(void)
{
    printf("sizeof(ENUM_1) = %d \r\n",
           sizeof(ENUM_1));
    printf("sizeof(ENUM_2) = %d \r\n",
           sizeof(ENUM_2));
    return 0;
}
```

VS 2005 云桌面2007 SP1

warning C4341: “E3”：有符号的值超出枚举常量的范围
warning C4309: “初始化”：截断常量值

```
C:\Windows\system32\cmd.exe
sizeof(ENUM_1) = 4
sizeof(ENUM_2) = 4
请按任意键继续. . .
```

Gcc 4.5.2 云桌面2007 SP1

```
z00114095@CTUY1Z001140951 /var
$ a.exe
sizeof(ENUM_1) = 4
sizeof(ENUM_2) = 8
```

枚举类型的大小 (2) : 注意事项

结论: 枚举的大小和编译器, 编译选项有关系, 具体大小无法统一确定。

建议:

1. 不要使用枚举的大小(与编译器有关)。
2. 枚举的值不要超过32位。

特别注意:

```
typedef enum
{
    E1 = 0x1,
    E2,
}ENUM_1;
```

```
typedef struct
{
    U32 m;
    ENUM_1 testEnum;
    U32 n;
}TEST_STRU;
```

结构体4字节对齐

某些编译器, 会按照枚举类型最大值定义类型大小, 大小可能为1

枚举的值（1）：案例说明

```
typedef enum
{
    E1,
    E2
}ENUM_1;

typedef enum
{
    E3 = 0x4,
    E4
}ENUM_2;

typedef enum
{
    E5 = 0x4,
    E6 = 0x7,
    E7
}ENUM_3;
```

```
typedef enum
{
    E8 = 0x4,
    E9 = 0x2,
    E10,
    E11
}ENUM_4;

typedef enum
{
    E12 = 0x4,
    E13,
    E14 = 0x5,
    E15 = 0x5,
}ENUM_5;

typedef enum
{
    E16 = 0x123456789,
    E17
}ENUM_6;
```

E1~E13结果是

```
$ ./a.exe
E1=0
E2=1
E3=4
E4=5
E5=4
E6=7
E7=8
E8=4
E9=2
E10=3
E11=4
E12=4
E13=5
E14=5
E15=5
```



枚举的值（2）：规则说明

基本规则：

1. 枚举第一个值如果没有定义，那么从0开始。
2. 枚举的值如果指定，那么就是指定值
3. 枚举的值如果没有指定，那么就是其上一个值+1。
4. 同一个枚举下面的值可以相同的。

注意事项：

考虑到不同编译器对枚举大小实现不同，建议枚举值不要超过32位。

使用建议

1. 尽量不要指定枚举值的大小。
2. 在结构体中，使用枚举字段需**谨慎**。
3. 枚举间尽量**不要强制转换**(无法检查)。

C语言进阶—结构体



目录

- 结构体初始化
- 结构体对齐
- 结构体大小计算

结构体初始化 (1)

结构体初始化的方式

```
typedef struct
{
    int a;
    int b;
    int c;
}TEST_STRU;
```

如何初始化?

```
TEST_STRU a={1, 2, 3};  
TEST_STRU a1={1, 2}; ?
```

结构体初始化 (2)

数组、嵌套结构体初始化

```
typedef struct
{
    int a;
    char b[10];
}ARR_STRU;
```

```
ARR_STRU arr1={10, "test"};
ARR_STRU arr2={10, {"t",'e','s','t'}};
```

```
typedef struct
{
    ARR_STRU a;
    int num;
    ARR_STRU buff[10];
}ARR_LIST;
```

```
ARR_LIST list={
    {0}, 10,
    {{1, "a"}, {2, "b"}}
};
```

结构体对齐

- 什么是结构体对齐
- 为什么要结构体对齐
- 结构体大小计算

结构体对齐：什么是结构体对齐

- **字节对齐：**现代计算机中内存空间都是按照byte划分的，从理论上讲似乎对任何类型的变量的访问可以从任何地址开始，但实际情况是在访问特定类型变量的时候经常在特定的内存地址访问，这就需要各种类型数据按照一定的规则在空间上排列，而不是顺序的一个接一个的排放，这就是对齐。
- **四字节对齐：**4字节类型数据希望从能被4整除的地址开始取操作数。

结构体对齐：为什么要结构体对齐（1）

为什么要四字节对齐？



- 运行速度更快？
 - 指针从地址读取数据可能出错？
 - 结构体直接赋值可能出错？
- 还有其他原因么？为什么？

结构体对齐：为什么要结构体对齐（2）

运行速度更快

```
#pragma pack(1)
typedef struct
{
    char a;
    int m;
}TEST_STRU_1;
```

```
#pragma pack(1)
typedef struct
{
    char a;
    char reserve[3];
    int m;
}TEST_STRU_4;
```

```
#pragma pack(4)
typedef struct
{
    char a;
    int m;
}TEST_STRU_4_2;
```

```
TEST_STRU_x test_x[1000];
for (int k = 0; k < 100000; k++)
{
    for (int m = 0; m < 1000; m++)
    {
        test_x[m].m = 0xcd;
    }
}
```

CPU:Intel 2.93GHz 4核 32位 (云) 运行结果 (时间tick)



```
C:\Windows\system32\cmd.exe
struct_1: size=5 type, run time=328
struct_4: size=8 type, run time=297
struct_42: size=8 type, run time=281
请按任意键继续. . .
```

结构体对齐：为什么要结构体对齐（3）

指针从基地址读取数据可能出错

```
typedef struct
{
    U8 a;
    U8 b;
    U8 c;
}TEST;

TEST = test{1,2,3};
U16 usShort;
U32 ullnt;

usShort = *(U16 *)&test.a;
usShort = *(U16 *)&test.b;
ullnt = *(U32 *)&test.c;
```

对于ARM体系结构来说，一般整型类型数据的对齐规则

类型	大小(位)	自然对齐(字节)
char	8	1 (字节对齐)
short	16	2 (半字对齐)
int	32	4 (字对齐)
long	32	4 (字对齐)

当CPU指令访问的内存地址违反对齐规则时：

1. 可能触发data abort exception，导致单板重启。
2. 读写数据出现偏差，结果与期待不同。

注意：是否出错和具体CPU和编译器强相关。

结构体对齐：为什么要结构体对齐（4）

结构体直接赋值时可能出错

```
typedef struct
{
    U8 a;
    U8 b;
    U32 c;
}TEST_STRU;

void TestStruCopy()
{
    TEST_STRU test = {1, 2, 3};

    char *p = (char *)malloc(100);
    if (p == NULL)
        return;
    p++;
    *(TEST_STRU *)p = test;
}
```

结论：

对于部分CPU，结构体赋值语句对应的汇编指令是块拷贝指令（LDM和STM），而块拷贝指令要求基址必须是4字节对齐，否则就会导致数据异常复位或者，或读取数据与期望不一致的情况。

注意：是否出错和具体CPU和编译器强相关。

结构体对齐：为什么要结构体对齐（5）

解决办法

- 自然对齐
 - ✓ 调整成员声明的次序,按照8、4、2、1字节的顺序排列
 - ✓ 不够对齐字段的用保留字段填充
- 用预编译选项对齐，`#pragma pack(n)`
- 不对齐结构不用非同类型指针访问，不进行类型转换。
- 尽量不对结构体直接赋值，使用`memcpy`拷贝整个结构体。

结构体大小计算 (1)

下面结构体大小是多少呢? (4字节对齐)

1) 6
typedef struct
{
 U8 a;
 U16 b;
 U16 c;
}TEST_1;



2) 8
typedef struct
{
 U32 a;
 U16 b;
}TEST_2;

4) 0 or 1
typedef struct
{
 ?;
}TEST_4;

基本概念:

1. 数据类型自身的对齐值: char自身对齐值为1, short为2, float为4, int通常为4等。
2. 结构体或者类的自身对齐值: 其成员中自身对齐值最大的那个值。
3. 指定对齐值: #pragma pack (value)时指定的对齐value。
4. 数据成员、结构体有效对齐值: 自身对齐值和指定对齐值中小的那个值。

对齐原则:

1. 每个成员分别按“有效对齐值”进行对齐, 起始地址%有效对齐值=0。
2. 结构体的默认对齐方式是它最长的成员的对齐方式, “起始地址%最大成员有效对齐值” = 0。
3. 结构体对齐后的长度必须是成员中有效对齐值的整数倍。

结构体大小计算 (2)

结构体字段为0字节

```
typedef struct
{
    int a;
    int b[0];
}TEST_1;
```

```
typedef struct
{
    int a;
    int b[];
}TEST_2;
```

```
void test()
{
    printf("test_1=%d\r\n", sizeof(TEST_1));
    printf("test_2=%d\r\n", sizeof(TEST_2));
}
```

?

test_1=4
test_2=4



C语言进阶—联合体

目录

- 基本用法
- sizeof大小

基本用法

```
typedef union
{
    unsigned int a;
    unsigned int b;
}TEST_U1;

typedef union
{
    unsigned int a;
    unsigned char b;
    unsigned short c;
}TEST_U2;
```

```
typedef struct
{
    unsigned char a;
    unsigned char b;
    unsigned char c;
    unsigned char d;
}TEST_S3;

typedef union
{
    unsigned int a;
    TEST_S3 b;
}TEST_U3;
```

// 小端模式

```
TEST_U1 u1;
TEST_U2 u2;
TEST_U3 u3;
```

```
u1.a = 0x12345678;
printf("u1:a=%x,b=%x\r\n", u1.a, u1.b);
```

```
u2.a = 0x12345678;
printf("u2:b=%x,c=%x\r\n", u2.b , u2.c);
```

```
u3.a = 0x12345678;
printf("u3:ba=%x, bd=%x\r\n", u3.b.a, u3.b.d)
```

```
u1 : a=12345678, b=12345678
u2 : b=78, c=5678
u3 : ba=78, bd=12
```

请按任意键继续. . .

基本用法

```
typedef union
{
    unsigned int a;
    unsigned int b;
}TEST_U1;

typedef union
{
    unsigned int a;
    unsigned char b;
    unsigned short c;
}TEST_U2;
```

```
typedef struct
{
    unsigned char a;
    unsigned char b;
    unsigned char c;
    unsigned char d;
}TEST_S3;

typedef union
{
    unsigned int a;
    TEST_S3 b;
}TEST_U3;
```

// 小端模式

TEST_U1 u1;
TEST_U2 u2;
TEST_U3 u3;

```
u1.a = 0x12345  
printf("u1:a=%x
```

u2.a = 0x12345678;

```
printf("u2:b=%x,c=%x\r\n", u2.b , u2.c);
```

u3.a = 0x12345678;

```
printf("u3:ba=%x,bd=%x\r\n", u3.b.a, u3.b.d)
```

u1:a=12345678,b=12345678
u2:b=78,c=5678
u3:ba=78, bd=12
请按任意键继续. . .

基本用法

```
typedef union
{
    unsigned int a;
    unsigned int b;
}TEST_U1;
```

```
typedef union
{
    unsigned int a;
    unsigned char b;
    unsigned short c;
}TEST_U2;
```

```
typedef struct
{
    unsigned char a;
    unsigned char b;
    unsigned char c;
    unsigned char d;
}TEST_S3;
```

```
typedef union
{
    unsigned int a;
    TEST_S3 b;
}TEST_U3;
```

// 小端模式

```
TEST_U1 u1;
TEST_U2 u2;
TEST_U3 u3;
```

```
u1.a = 0x12345678
printf("u1:a=%0X",
```

```
u1:a=12345678
u2:b=78,c=5678
u3:ba=78,bd=12
请按任意键继续. . .
```

TEST_U3			
a	0x78	0x56	0x34
b	0x78	0x56	0x34
a b c			d
低地址			高地址

```
u2.a = 0x12345678;
printf("u2:b=%0X,c=%0X\r\n", u2.b , u2.c);
```

```
u3.a = 0x12345678;
printf("u3:ba=%0X,bd=%0X\r\n", u3.b.a, u3.b.d);
```

sizeof大小

```
typedef union
{
    unsigned int a;
    unsigned int b;
}TEST_1;
```

```
typedef union
{
    unsigned int a;
    unsigned char b;
    unsigned short c;
}TEST_2;
```

```
typedef struct
{
    unsigned char a;
    unsigned char b;
}TEST_3;
```

```
typedef union
{
    unsigned int a;
    TEST_3 b;
}TEST_4;
```

```
TEST_1 u1;
TEST_2 u2;
TEST_3 u3;
TEST_4 u4;
```

```
printf("u1=%x\r\n", sizeof(u1));
printf("u2=%x\r\n", sizeof(u2));
printf("u3=%x\r\n", sizeof(u3));
printf("u4=%x\r\n", sizeof(u4));
```

```
u1=4  
u2=4  
u3=2  
u4=4
```

请按任意键继续. . .

C语言进阶—函数



目录

- 函数定义声明
- 函数入参
- inline函数
- static函数



0:13 / 5:38



Note

720P

1.5x



函数定义声明

1. 请申明一个函数，函数返回int？

```
int Test1();
```

2. 请申明一个函数，函数2个入参，一个是整形，一个是char类型的数组？

```
void Test2(int num, char array[]);
```

3. 请定义一个函数指针，这个指针指向上面的第一个函数。

```
typedef int (*PFUN)();  
PFUN pf = Test1;
```



0:54 / 5:38



Note



720P



1.5x



函数入参

请说明下面代码的输出时什么？

```
void Test(char a, char *b, char c[], char d[5])
{
    printf( "a=%d;" ,sizeof(a));
    printf( "b=%d;" ,sizeof(b));
    printf( "c=%d;" ,sizeof(c));
    printf( "d=%d;" ,sizeof(d));
}
```

输出： a=1;b=4;c=4;d=4;

inline 函数

1. 如何写inline函数？

```
inline int Test1();
```

2. inline函数的目的是什么？

- a) 没有调用开销，效率高；
- b) 是一个真正的函数，编译器会检查参数类型，消除宏函数的隐患；
- c) 内联函数太复杂or调用点太多，展开后导致的代码膨胀带来的恶化可能大于效率提升带来的益处。

3. 函数加上inline头，就一定会被inline么？

- a) inline对编译器只是建议，编译器可以选择忽略这个建议；
- b) 在调用内联函数时，要保证内联函数的定义让编译器“看”到，也就是说内联函数的定义要在头文件中，这与通常的函数定义不一样。

extern 函数

1.如何写extern函数?

```
extern int Test1();
```

2.extern函数的目的是什么?

- a)extern可置于变量或者函数前，以表示变量或者函数的定义在别的文件中，提示编译器遇到此变量和函数时在其他模块中寻找其定义；
- b)取代include “*.h”来声明函数；
- c)extern “C”
C++在编译时为解决函数多态问题，会将函数名和参数联合起来生成一个中间的函数名，而C语言则不会，因此会造成链接时找不到对应函数的情况。

static 函数

1. 如何写static函数?

```
static int Test1();
```

2. static函数的目的是什么?

- a) 限定作用域，只能被本文件中的其他函数调用，
而不能被同一程序其它文件中的函数调用。

C语言进阶—内存分布



0:04 / 5:21

Note

720P

1.5x

目录

- 案例说明
- 内存分布概述

案例说明 (1)

```
char* GetString_10
{
    char dataList[] = {'a','b','c','\0'};
    return dataList;
}

char* GetString_20
{
    char dataList[] = "abc";
    return dataList;
}

char* GetString_30
{
    char *dataList = "abc";
    return dataList;
}

char* GetString_40
{
    static char dataList []= "abc";
    return dataList;
}
```



```
void Test()
{
    char *dataList;
    dataList = GetString_10();
    printf("string_1:%s \r\n",dataList);

    dataList = GetString_20();
    printf("string_2:%s \r\n",dataList);

    dataList = GetString_30();
    printf("string_3:%s \r\n",dataList);

    dataList = GetString_40();
    printf("string_4:%s \r\n",dataList);
}
```

```
C:\Windows\system32\cmd.exe
string_1:?
string_2:?
string_3:abc
string_4:abc
请按任意键继续. . .
```

案例说明 (2)

```
int g_globleVariable;  
  
void TestVariableInit_1()  
{  
    printf("g_globleVariable=%x \r\n", g_globleVariable);  
}  
  
void TestVariableInit_2()  
{  
    int tempVariable;  
    printf("tempVariable=%x\r\n", tempVariable);  
}  
  
void TestVariableInit_3()  
{  
    int *mallocVar = (int*)malloc(sizeof(int));  
    if (mallocVar != NULL)  
    {  
        printf(" mallocVariable=%x \r\n ", * mallocVar);  
    }  
}
```

```
void OutputVariableDefaultInitValue()  
{  
    TestVariableInit_1();  
    TestVariableInit_2();  
    TestVariableInit_3();  
}
```

Release

```
C:\Windows\system32\cmd.exe  
g_globleVariable=0  
tempVariable=73ef32e4  
mallocVariable=554a30  
请按任意键继续... .
```

Debug

```
C:\Windows\system32\cmd.exe  
g_globleVariable=0  
程序崩溃了  
mallocVariable=cccccccc  
请按任意键继续... .
```

案例说明 (3)

```
1 // Copyright (c) Huawei Technologies Co., Ltd. 2012-2018. All rights reserved.
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6
7 int a = 0x0506; // 全局初始化区 -- 数据段
8 char *p0 = "this is test string";
9 char *p1; // 全局未初始化区 -- BSS段
10
11 int main()
12 {
13     int b; // 局部变量 - 栈
14     char s[] = "abc"; // 栈 (数组)
15     char *p2; // 栈
16     char *p3 = "1234"; // 1234在常量区, p3在栈上。
17     static int c = 0x0203; // 全局 (静态) 初始化区
18     static char *str1 = "abcd";
19
20     p2 = (char *)malloc(10); // 分配得来得10字节的区域就在堆区。
21     strcpy(p2, "5678");
22
23     printf("addr = %p\n", p2);
24
25     free(p2);
26
27     return 0;
28 }
```

Contents of section .data:
601038 00000000 00000000 00000000 00000000
601048 06050000 00000000 c4064000 00000000@....
601058 e8064000 00000000 03020000 ...@.....

Contents of section .rodata:
4006c0 01000200 74686973 20697320 74657374this is test
4006d0 20737472 696e6700 31323334 00616464 string.1234.add
4006e0 72203d20 25700a00 61626364 00 r = %p..abcd.

案例说明 (3)

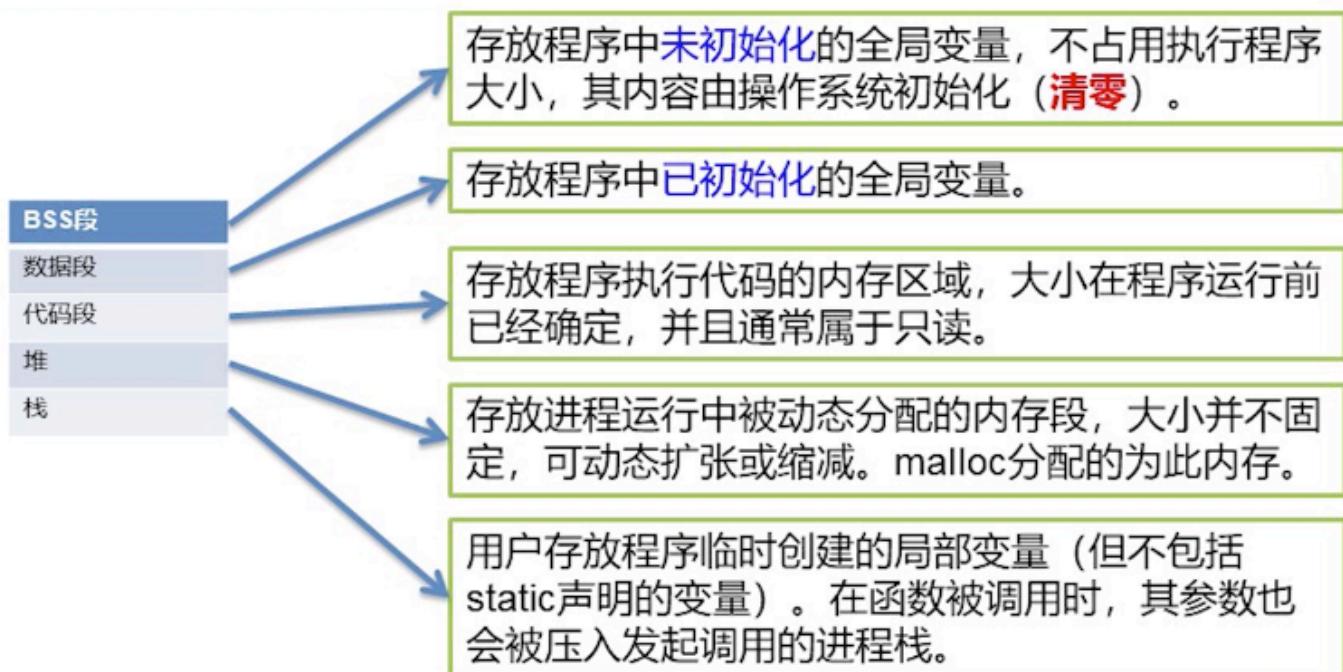
```
1 // Copyright (c) Huawei Technologies Co., Ltd. 2012-2018. All rights reserved.
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6
7 int a = 0x0506; // 全局初始化区 - 数据段
8 char *p0 = "this is test string";
9 char *p1; // 全局未初始化区 --BSS段
10
11 int main()
12 {
13     int b; // 局部变量 - 栈
14     char s[] = "abc"; // 栈 (数组)
15     char *p2; // 栈
16     char *p3 = "1234"; // 1234在常量区, p3在栈上。
17     static int c = 0x0203; // 全局 (静态) 初始化区
18     static char *str1 = "abcd";
19
20     p2 = (char *)malloc(10); // 分配得来得10字节的区域就在堆区。
21     strcpy(p2, "5678");
22
23     printf("addr = %p\n", p2);
24
25     free(p2);
26
27     return 0;
28 }
```

Contents of section .data:
601038 00000000 00000000 00000000 00000000
601048 06050000 00000000 C4064000 00000000@...
601058 e8064000 00000000 03020000 ...@.....

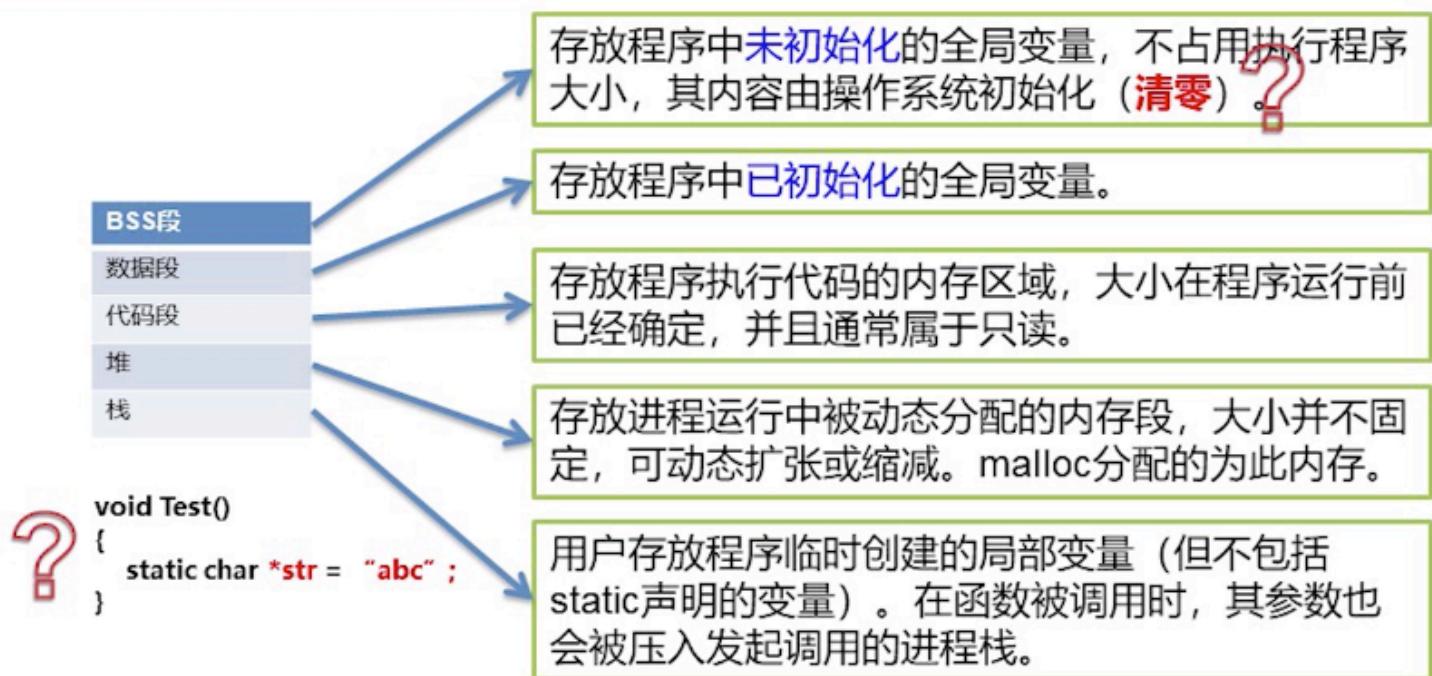
Contents of section .rodata:
4006c0 01000200 74686973 20697320 74657374this is test
4006d0 20737472 696e6700 31323334 00616464 string.1234.add
4006e0 72203d20 25700a00 61626364 00 r = %p..abcd.

C4064000, 转换为大端, 即004006C4, 即this is test string的起始地址。
结论: 指针p0保存在data段, 内容在rodata段。

内存分布概述



内存分布概述



C语言进阶—数组和指针



目录

- 指针和数组基本形态
- 数组和指针不可互换场景
- 数组和指针可互换场景
- 多维数组和指针
- 数组和指针初始化

指针和数组基本形态（1）

指针用法

```
char ch = 'c';
① char *pch = &ch;
char *pStr = "abc" ;

int m;
② int *pm = &m;

struct Test stTest;
③ struct Test *pst = &stTest;

extern void TestFunc(int m);
④ void (*pf)(int) = TestFunc;
```

数组用法

```
char m[10];
char m[] = "123";
char m[] = {'1','2'};

int m[3][2];
int *k[10];
```

指针和数组基本形态（2）

数组访问

```
int Test1()
{
    char test[] = {'1','2','3', '4', '5'};
    int m = test[1];

    return m;
}
```

获取test[1]的步骤：

1. test表示数组起始地址，先获得其地址
2. test地址加上偏移，获取具体值

指针和数组基本形态（3）

指针访问

```
int Test2()
{
    char *test = "12345";
    int m = test[3];

    return m;
}
```

获取test[3]的步骤：

1. 先获取test变量地址。
2. 获取test地址的数据，即“12345”地址。
3. 上面的“12345”首地址+偏移获取具体值。

数组和指针不可互换场景： sizeof

```
char arr[] = "123456";  
int m = sizeof(arr);
```



```
char *pstr = "123456";  
int m = sizeof(pstr);
```

数组和指针可互换场景(1): 函数参数

```
void TestArray_1(char test[10])
{
    int m = sizeof(test);
}

void TestArray_2(char test[])
{
    int m = sizeof(test);
}
```



```
void TestPointer(char *test)
{
    int m = sizeof(test);
}
```

数组和指针可互换场景(2): 表达式

```
char arr[] = "123456";  
char ch = arr[2];
```

VS

```
char *pstr = "123456";  
char ch2= *(pstr+2);
```

注意: 表达式中, arr[i]总是被编译器翻译成*(arr + i)

多维数组和指针（1）：步长

什么是步长：数组或者指针地址，偏移一步的内存大小。

char a[10]; // 假设a起始地址: 0x100

(a+1) = ?

0x101

int b[5]; // 假设b起始地址: 0x100

(b+1) = ?

0x104

int c[4][2]; // 假设c起始地址: 0x100

(c+1) = ?

0x108

(*c + 1) + 1 = ?

0x10C

多维数组和指针（2）：定义

请按照功能定义字数组：

1. 定义一个二维数组，用来存放二维地图，其中X轴最大10，Y轴最大20。数据为int型。
2. 定义一个三维数组，用来存放三维地图，其中X轴最大10，Y轴最大20，Z轴最大30。数据为int型。
3. 定义一个数组，数组有10个，每个数组中，存放一个字符串指针。
4. 申明一个函数，函数返回一个二维数组地图，其中最右边空间为10，数据为int型。



参考答案：

- | | |
|---------------------------|-----------------------|
| 1) int map_1[20][10]; | 3) char *str_1[10]; |
| 2) int map_2[30][20][10]; | 4) int (*Func())[10]; |

多维数组和指针（3）：函数形参

一维数组

```
void Test_1(char p[10]);  
void Test_2(char p[]);  
void Test_3(char *p);
```

二维数组

```
void Test_1(char p[10][2]);
```

如何翻译？

注意：形参数组，均会被编译成指针。

```
void Test_2(char p[][2]);  
void Test_3(char (*p)[2]);
```

数组和指针初始化（1）：一维

下面为全局变量

```
int test_1[10];
int *test_2;

int test_3[10] = {1,2,3};
int test_4[] = {1,2,3};
char test_5[] = "123";

int m = 0;
int *test_6=&m;
```



test_1[5] = ?

test_2[5] = ?

test_3[5] = ?

test_4[2] = ?

test_5[5] = ?

sizeof(test_5)=?

0

未知

0

3

未知

4 (包含'\0')