

寄语

C语言编程指南是华为公司研发和专家多年实践
经验的总结，希望本指南能帮助到你！

总体介绍

简介

课前准备

- 1、具备基本的C语言编码能力
- 2、完成《华为C语言编程指南》全部内容的学习

学习目标

- 1、掌握C语言编码风格的要求
- 2、掌握C语言使用的基本要求
- 3、掌握常见编码漏洞的产生原理和基本防范方法

主要 章节内容

本课程基于华为长期的编码实践和总结，依托《华为C语言编程指南》，并借鉴业界相关规范标准，从中提取出了关键点内容来进行介绍。相关内容分别为：

第一章 引言（明确可读、可维护、安全等特征的重要性）

第二章 代码风格（命名 注释 格式）

第三章 编程实践（C语言基础和难点，函数设计，库函数使用等）

课程会从原理、业界案例、实际问题展示、典型编码错误及预防这几个方面进行阐述。

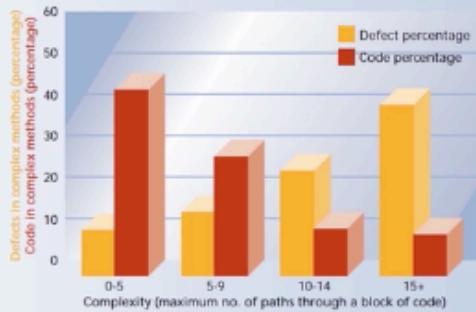
企业编程指南 (C)



◆ 可读，可维护

- 代码阅读的次数远远多于编写的次数。
-- 《代码大全》
- 代码的写法应当使别人理解他所需的时间最小化。
-- 《编写可读代码的艺术》
- 代码应更多关注于向人们解释我们想让计算机做什么，而不仅仅是告诉计算机怎么做。
-- Donald Knuth
"Literate Programming"

Figure Complex code contains more defects.
Understanding complexity metrics can help you
spend review time wisely.



上图说明了使用圈复杂度度量来分析了一个数据库系统。很容易看出，几乎一半的代码存在于具有相当简单的方法中，而大部分缺陷位于少量的复杂代码中。

Mark Schroeder, "A Practical Guide to Object-Oriented Metrics", IT Pro, Nov/Dec 1999

Keep it simple.

◆ 安全

■ 2020年CWE Top 25

In the 2020 CWE Top 25, including the overall score of each.

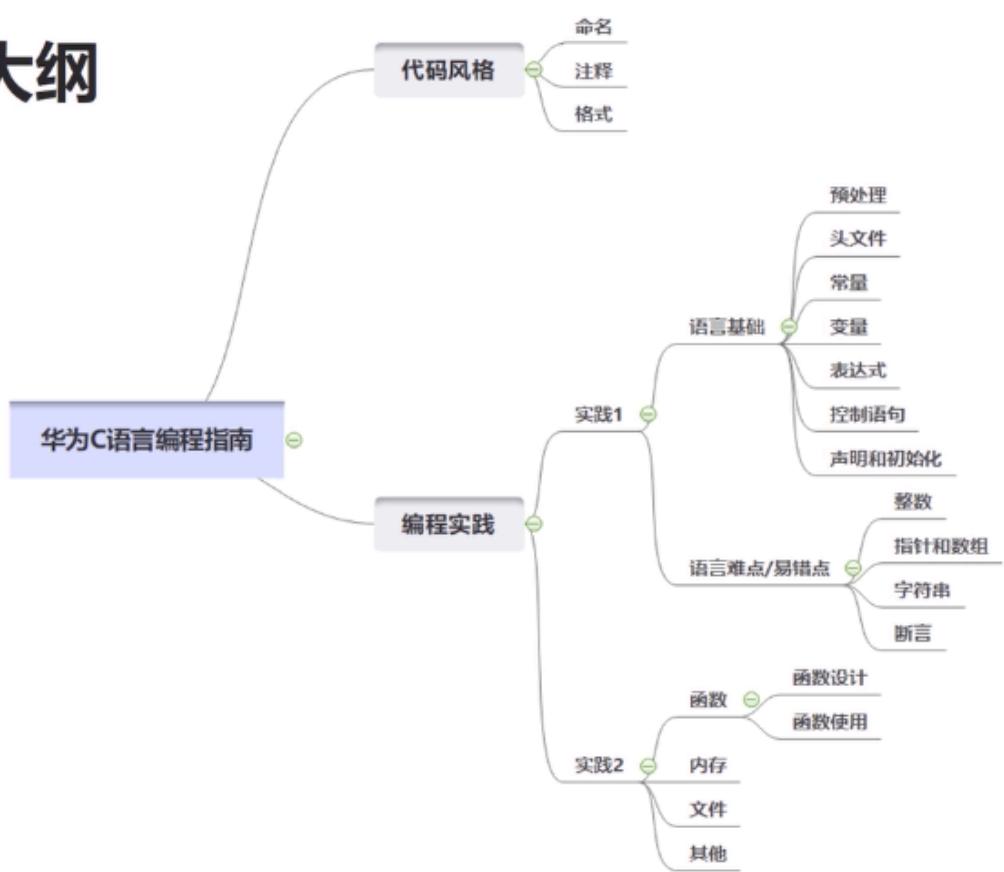
Rank	ID	Name	Score
[1]	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	46.82
[2]	CWE-787	Out-of-bounds Write	46.17
[3]	CWE-20	Improper Input Validation	33.47
[4]	CWE-125	Out-of-bounds Read	26.50
[5]	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	23.73
[6]	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	20.69
[7]	CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	19.16
[8]	CWE-416	Use After Free	18.87
[9]	CWE-352	Cross-Site Request Forgery (CSRF)	17.29
[10]	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	16.44
[11]	CWE-190	Integer Overflow or Wraparound	15.81
[12]	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	13.67
[13]	CWE-476	NULL Pointer Dereference	8.35
[14]	CWE-287	Improper Authentication	8.17
[15]	CWE-434	Unrestricted Upload of File with Dangerous Type	7.38
[16]	CWE-732	Incorrect Permission Assignment for Critical Resource	6.95
[17]	CWE-94	Improper Control of Generation of Code ('Code Injection')	6.53
[18]	CWE-522	Insufficiently Protected Credentials	5.49
[19]	CWE-611	Improper Restriction of XML External Entity Reference	5.33
[20]	CWE-798	Use of Hard-coded Credentials	5.19
[21]	CWE-502	Deserialization of Untrusted Data	4.93
[22]	CWE-269	Improper Privilege Management	4.87
[23]	CWE-400	Uncontrolled Resource Consumption	4.14
[24]	CWE-306	Missing Authentication for Critical Function	3.85
[25]	CWE-862	Missing Authorization	3.77

数据来源：https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html

2020年CWE/SANS Top25最危险编码错误，涉及C语言有14项，占比56%

引言

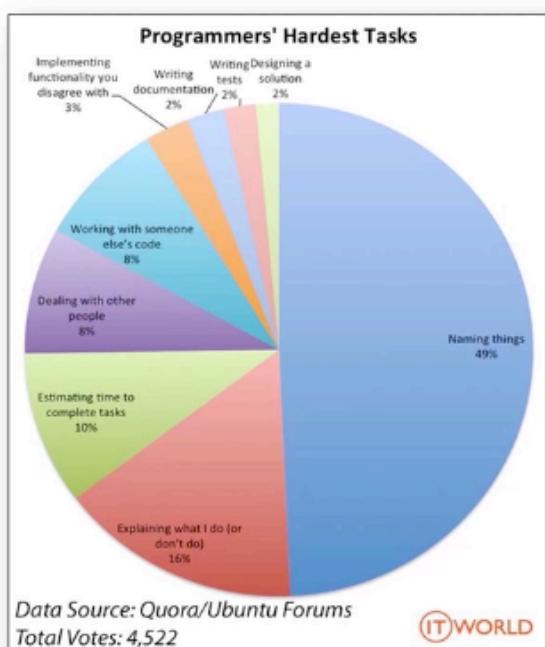
指南大纲



代码风格



◆ P.01 标识符命名应符合阅读习惯



程序员头疼的事情：命名（49%）

文件的命名
模块的命名
函数的命名
变量的命名
宏的命名
类型的命名
枚举、常量的命名

命令困难时，请考虑：
-设计是否合理?
-职责是否单一?
-是否可拆分?
-是否多余?
-英文不行?

图片来源：

<https://www.computerworld.com/article/2833265/don-t-go-into-programming-if-you-don-t-have-a-good-thesaurus.html>

◆ G.NAM.01 使用统一的命名风格

■ “驼峰” 风格

- 大驼峰 (UpperCamelCase)
- 小驼峰 (lowerCamelCase)

■ “内核” 风格

- `snake_case`, 又称蛇形风格。

■ “匈牙利” 风格

- `uiUpperCamelCase`

// 符合: 函数大驼峰, 参数小驼峰
`int SoftCommand(const char *softCmd, size_t cmdLen);`

`enum MyColor {` // 符合: 枚举类型, 大驼峰
 `BLACK,` // 符合: 枚举值, 全大写
 `WHITE`
}; `g_bgColor = WHITE;` // 符合: 全局变量, 带g_前缀的小驼峰

// 符合: 常量定义, 用全大写下划线分割
`const int NAME_MAX_LEN = 100;`

代码风格

◆ 按需注释，全面且不冗余

不要少：

换位思考：阅读者此时需要哪些信息？（可读、可维）

阅读者无从知晓此处
0x40000000的含义

```
86: S32 g_as1DupEqDefaultCoef[DUP_PH_EQU_COEF_NUM] = { 0, 0, 0, 0x40000000, 0, 0 };
```

不要多：

冗余注释会弱化有效注释，让所有注释缺乏有效维护

```
292: ULONG ulRxSolicit; /* 接收到的Solicit */
293: ULONG ulRxAdvertise; /* 接收到的Advertise */
294: ULONG ulRxRequest; /* 接收到的Request */
295: ULONG ulRxConfirm; /* 接收到的Confirm */
296: ULONG ulRxRenew; /* 接收到的Renew */
297: ULONG ulRxRebind; /* 接收到的Rebind */
298: ULONG ulRxDecline; /* 接收到的Decline */
299: ULONG ulRxRelease; /* 接收到的Release */
300: ULONG ulRxReply; /* 接收到的Reply */
301: ULONG ulRxReconfigure; /* 接收到的Reconfigure */
```

为了注释而注释，无意义

代码风格

小测验

1. 下列代码中的符号命名有哪些问题？

```
#define MAX 100
static int g_abcUserCnt;

int save_abc_user_cnt(int abcUserCnt, bool flag)
{
    if (flag && abcUserCnt > MAX) {
        return -1;
    }
    g_abcUserCnt = abcUserCnt;
    return 0;
}
```

MAX 命名不够精确

函数名命名风格与其他不一致（非驼峰）

参数 flag 过于简短，含义不明确

代码风格

◆ 函数头注释 不要“复制、粘贴”

```
55: /*****
56: 函数名 : DHCPV6_L3RelayInit
57: 功能描述: DHCPv6 L3 Relay初始化
58: 输入参数: VOID
59: 输出参数: 无
60: 返回值: 无
61: 调用函数:
62: 被调函数:
63:
64: 修改历史 :
65: 1.日期 : 2010年12月7日
66: 修改内容 : 新生成函数
67:
```



固定格式函数头的害处：

- 常空有格式无内容
- 信息冗余
- 忘记修改，注释与代码冲突
- 弱化有效注释，缺乏维护，最终不可信

“复制粘贴”是程序员的天敌！

代码风格

小测验

1. 某公司对代码要求注释率不低于50%。是否合理？为什么？

不合理。

好的代码应该是自注释的，只在难以理解的地方辅以注释说明，提高可读性。

给注释率设置一个下限，很容易导致冗余注释的产生，
增加了维护成本的同时，也渐渐降低了代码可读性。

◆ 格式



◆ 换行

选择合适的断行点换行

从可读性角度出发：

- 选择附近连接符优先级较低的
- 不破坏局部整改性
- 不要拿行宽上限作为断行依据

```
dbBandNcoValue = (double)adcBandNcoInfo[devIndex].Freq - calcSetValue * sampleFreq /  
    pow(2.0, adcNecoBitWidth);
```



```
dbBandNcoValue = (double)adcBandNcoInfo[devIndex].Freq -  
    calcSetValue * sampleFreq / pow(2.0, adcNecoBitWidth);
```



新行要么按层次缩进，要么同类对齐

有时适当对齐，可读性会更好

```
static OmciMibTableSetHookTable tableSetTbl[] = {  
    {ME_EXTENDED_VLAN_TAG, true, OMCI_MibSetTxVlanTblAtt},  
    {ME_IP_STATIC_ROUTES, false, OMCI_MibSetIpStaticRouteTblAtt},  
    {ME_HB_MULTICAST_VLANMODE, false, OMCI_MibUpdateMeVlanTciTable},  
    {ME_MULTI_SUB_CONFIG, true, OMCI_MibSetMcSvcPackTblAtt},  
    {ME_HB_PROPRIETARY_CAP_DECLARE, false, OMCI_SetTestAbilityMibEntry}  
};
```



代码风格

◆ 写风格统一的代码

相比个人习惯，所有人共享同一种风格带来的好处，远远超出为统一而付出的代价。

在编程指南的指导下，审慎地编排代码以使代码尽可能清晰、风格统一。

小到个人项目，大到公司项目，要保持统一的代码风格！

编程实践

实践.1

- 01 预处理
- 02 头文件
- 03 数据类型
- 04 常量
- 05 变量
- 06 表达式
- 07 控制语句
- 08 声明和初始化
- 09 整数
- 10 指针和数组
- 11 字符串
- 12 断言

预处理等八类实践

◆ 常见不好的宏封装

```
// 不符合: 宏参数缺少括号  
#define MAX(a, b) a > b ? a : b
```

G.PRE.02 定义宏时, 要使用完备的括号

```
// 不符合: 多条语句未用 do-while(0) 包装  
#define AAA(a) \  
    Foo(a); \  
    Bar(b);
```

G.PRE.03 包含多条语句的函数式宏的实现语句必须放在 do-while(0) 中

```
// 不符合: 宏依赖函数中的局部变量  
#define AAA_MAX_CNT param->maxCnt
```

G.PRE.10 宏定义不应依赖宏外部的局部变量名

```
// 不符合: 宏结果有多余分号  
#define AAA(a) Foo(a);
```

G.PRE.09 宏定义不以分号结尾

```
// 不符合: 内部使用了return语句  
#define CHECK_RET(ret) \  
    if (ret != OK) { \  
        DEBUG_PRINT("ret = %d", ret); \  
        return ret; \  
    }
```

G.PRE.05 函数式宏定义中慎用return、goto、continue、break 等改变程序流程的语句

预处理等八类实践

◆ G.INC.01在头文件中声明需要对外公开的接口

头文件在编译时被文本替换展开。

从而实现代码复用，使用同一套接口（声明），遵从相同的“约定”。

头文件是用于**被别人包含的**。

头文件的依赖关系，**体现了架构设计**。

错误观点：

- .c是函数的堆叠，其他所有写到.h中
- .h是为了解决“使用前未声明”问题的
- 源文件过大，拆分一个“私有头文件”出来

预处理等八类实践

◆ G.INC.05 头文件应当自包含

简单的说，自包含就是任意一个头文件均可独立编译。

```
// a.h  
#define MY_PRINTF printf
```



```
// b.h  
#include <stdio.h>  
#define MY_PRINTF printf
```



预处理等八类实践

◆ G.INC.05 头文件应当自包含

只要包含万能头文件，
再不用担心找不到声明了！

开发一时爽，维护泪两行！

```
19  +#include "src/common/include/vos_std.h"
20  +#include "src/common/algorithm/xpon_ordertbl.h"
21  +#include "src/common/algorithm/xpon_ordertbl.inc"
22  +#include "src/common/include/xpon_pub.h"
23  +#include "src/common/stub/pile_type.h"
24  +#include "src/common/include/xpon_err.h"
25  +#include "src/common/include/xpon_pub.h"
26  +#include "src/common/pub/omshl_xpon.h"
27  +#include "/xpon_type.lib.h"
28  +#include "src/common/omshl/xpon_omshl.h"
29  +#include "src/common/pub/xpon_db.h"
30  +#include "src/common/include/err_xpon_plt.h"
31  +#include "src/common/stub/pile_type.h"
32  +#include "src/common/include/xpon_pub.h"
33  +#include "src/common/pub/omshl_xpon.h"
34  +#include "src/common/omshl/xpon_omshl.h"
35  +#include "src/common/pub/xpon_db.h"
36  +#include "src/common/algorithm/xpon_ordertbl_lock.h"
37  +#include "src/include/xpon_api.h"
38  +#include "src/include/xpon_mscoapi.h"
39  +#include "src/include/inner/xpon_omshl_api.h"
40  +#include "src/common/include/xpon_id.h"
41  +#include "src/xponcfg/kernel/common/xpon_sym.inc"
42  +#include "src/include/xpon_api.h"
43  +#include "include/dbccb_api.h"
```

预处理等八类实践

小测验

1. 宏定义是放在 .h 中还是 .c 中？

只在当前 .c 中使用的，内部的，不对外的宏，放在 .c 中
如果宏需要对外开放，供别人使用，则放在 .h 中

预处理等八类实践

◆ G.VAR.01 禁止读取未经初始化的变量

■ 场景举例：

- 函数传参未初始化
- 申请内存的变量未初始化
- 内存拷贝引用的长度值未初始化
- 解引用的指针未初始化
-



■ 危害很大：

- 缓冲区溢出
- 访问非法内存
- 拒绝服务
-

预处理等八类实践

◆ G.VAR.04 慎用全局变量



- ✓ 尽量少用全局变量
- ✓ 尽量不跨文件
- ✓ 尽量封装

全局变量应视为内部实现方法之一，
不应用作接口。应使用函数做接口。

示例：右边是同一接口的3个演进版本
如果直接用全局变量作接口，将带来很多麻烦。

```
int GetSchoolPeopleCnt()
{
    return g_curSchoolPeopleCnt;
}
```



```
int GetSchoolPeopleCnt()
{
    return g_studentCnt + g_teacherCnt;
}
```



```
int GetSchoolPeopleCnt()
{
    int teacherCnt = GetTeacherCnt();
    int studentCnt = GetStudentCnt();
    return teacherCnt + studentCnt;
}
```



预处理等八类实践

◆G.VAR.08 资源不再使用时应予以关闭或释放

这里的资源包括：计算机内存、文件描述符、socket描述符。

开发者在创建或分配资源后，如果该资源不再被使用，应将其正确的关闭或释放。

要注意所有可能的异常路径，避免遗漏。

【反例】

```
#define BLOCK_SIZE_MAX 256  
  
char *GetBlock(int fd)  
{  
    ...  
    char *buf = (char *)malloc(BLOCK_SIZE_MAX);  
    if (buf == NULL) {  
        return NULL;  
    }  
  
    if (read(fd, buf, BLOCK_SIZE_MAX) != BLOCK_SIZE_MAX) {  
        return NULL; ①  
    }  
    return buf;  
}
```



错误：在异常路径中返回前未释放buf指向的内存资源，存在内存泄漏

【正例】

```
#define BLOCK_SIZE_MAX 256  
  
char *GetBlock(int fd)  
{  
    ...  
    char *buf = (char *)malloc(BLOCK_SIZE_MAX);  
    if (buf == NULL) {  
        return NULL;  
    }  
  
    if (read(fd, buf, BLOCK_SIZE_MAX) != BLOCK_SIZE_MAX) {  
        free(buf);  
        buf = NULL;  
    }  
    return buf;  
}
```



3.2

整数



▶ 0:00 / 8:32

Note 720P 1.5x 🔍 ↻

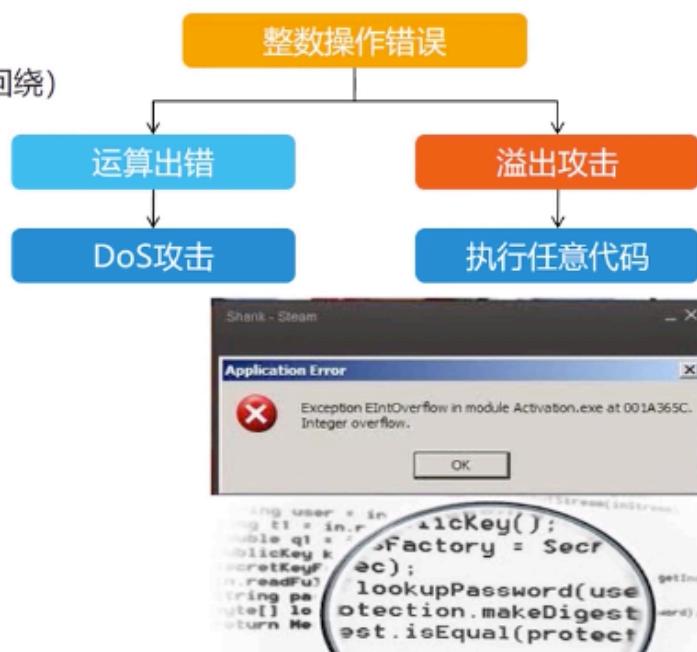
◆ 常见的整数操作问题

■ 整数操作不当的几种场景

- 有（无）符号整数运算操作出现溢出（回绕）
- 整型转换时出现截断错误
- 有符号整数使用位操作符运算

■ 整数操作不当带来的风险

- 内存分配出错
 - 用户输入在与有符号的值和无符号的值之间的隐式转换进行交互时会产生一些错误，而这些错误经常会导致内存分配函数出现问题。
- 执行任意代码
 - 整数溢出错误，可能导致内存破坏，并可能被用于执行任意代码。
- 缓冲区溢出



◆ 整数溢出/回绕原理

■ 有符号整数溢出

- 若算术运算的计算结果太大而无法在系统位宽度范围内存储时导致整数溢出；当操作数都是有符号数时，溢出就有可能发生，而且溢出的结果是未定义的。当一个运算的结果发生溢出时，所有关于结果如何的假设均不可靠。



Int8		
Sign bit	Value bits	
1	0 0 0 0 0 0 0 0	= -128
-	0 0 0 0 0 0 0 1	= 1
=	0 1 1 1 1 1 1 1	= 127

■ 无符号整数回绕 (部分书籍称之为反转)

- 无符号操作数的计算不会溢出，因为结果不能被无符号类型表示的时候，就会对比结果类型能表示的最大值加1再执行求模操作。



UInt8		
Sign bit	Value bits	
0	1 1 1 1 1 1 1 1	= 255
+	0 0 0 0 0 0 0 1	= 1
=	1 0 0 0 0 0 0 0	= 0

◆ 可能引发有符号整数溢出和无符号整数回绕操作

■ 会引起整数溢出或回绕的操作符

操作符	是否溢出	是否回绕
+	是	是
-	是	是
*	是	是
/	是	否
%	是	否
++	是	是
--	是	是
+ =	是	是
- =	是	是
* =	是	是
/ =	是	否
% =	是	否
<< =	是	是
<<	是	是
一元 -	是	是

■ 来自于系统外部或其它不可信数据参与到左侧运算中的情形，只要将运算结果用于以下之一（包括但不局限于）的用途，都应该添加校验以防止溢出/回绕：

- 作为数组索引
- 指针运算
- 作为对象的长度或者大小
- 作为数组的边界
- 作为内存分配函数的实参
- 作为循环终止判定条件
- 作为拷贝长度

◆ 可能引发有符号整数溢出和无符号整数回绕操作

■ 会引起整数溢出或回绕的操作符

操作符	是否溢出	是否回绕
+	是	是
-	是	是
*	是	是
/	是	否
%	是	否
++	是	是
--	是	是
+=	是	是
-=	是	是
*=	是	是
/=	是	否
%=	是	否
<<=	是	是
<<	是	是
一元 -	是	是

- 来自于系统外部或其它不可信数据参与到左侧运算中的情形，只要将运算结果用于以下之一（包括但不局限于）的用途，都应该添加校验以防止溢出/回绕：
 - 作为数组索引
 - 指针运算
 - 作为对象的长度或者大小
 - 作为数组的边界
 - 作为内存分配函数的实参
 - 作为循环终止判定条件
 - 作为拷贝长度

◆ 典型错误 - 有符号整数运算时可能出现溢出

【问题描述】 有符号整数运算时可能出现溢出，造成未定义行为。

【反例】

```
int opt = ... // 来自用户态
if ((opt < 0) || (opt > (ULONG_MAX / (60 * HZ)))) {
    ... // 错误处理
}
... = opt * 60 * HZ; // 可能出现整数溢出
...
```



【正例】

```
// 定义变量时应考虑用途，而不仅是机械地都定义成int类型
unsigned long opt = ... // 重构 opt 类型
if (opt > (ULONG_MAX / (60 * HZ))) {
    ... // 错误处理
}
... = opt * 60 * HZ;
...
```



【总结说明】

- 如果参与运算的有符号整数来自外部，需要考虑运算后是否出现溢出
- 如果参与运算的有符号整数的绝对值偏大，需要考虑算后是否出现溢出。

◆ 典型错误-整数转换时出现截断错误和符号错误

【问题描述】

有符号整数发生符号错误

【反例】

```
void Func(unsigned char *msg, long len)
{
    unsigned char *buffer = NULL;
    if (len == 0) {
        ... // 处理错误
    }
    // 【错误】如果 len 的值小于0，会发生符号错误
    buffer = (unsigned char *)malloc((size_t)len);
    ...
}
```



【正例】

```
// 【修改】重构函数 len 参数的类型为 size_t 类型
void Func(unsigned char *msg, size_t len)
{
    unsigned char *buffer = NULL;
    if (len == 0 || len > MAX_LEN) {
        ... // 处理错误
    }
    buffer = (unsigned char *)malloc(len);
    ...
}
```



【总结说明】

- 整型数据参与转换时，确保不要出现截断和符号错误。
- 表示对象大小的整数值或变量应当使用size_t类型

◆ 典型错误-除零错误

【问题描述】 当除数为0时，造成未定义行为。除零（被零除）错误容易被忽略。

【反例】

```
size_t a = ReadSize(); // 通过接口读取  
size_t b = 500 % a;    // a 可能是0  
size_t c = 1000 / a;   // a 可能是0  
  
...
```



【正例】

```
size_t a = ReadSize(); // 通过接口读取  
if (a == 0) {  
    ... // 错误处理  
}  
size_t b = 500 % a;    // 已确保 a 不为0  
size_t c = 1000 / a;   // 已确保 a 不为0  
  
...
```



【总结说明】

- 确保除法和取余运算不会导致除零（被零除）错误。

小测验

1. (判断) 有时从有符号整型转换到无符号整型会发生符号错误，但是符号错误并不丢失数据，因此不会造成安全问题。

答案：错误。

虽然没有丢失数据，但在某些情况下，整数符号错误也会造成安全漏洞。

3.3 数组和字符串

▶ 00:00 / 11:39

Note 720P 1.5x 🔍 ↻

◆ 数组操作介绍

■ 数组及操作

- 数组：由一个连续分配的相同类型的对象组成，数组索引的合法范围是[0, 数组元素数 - 1]。

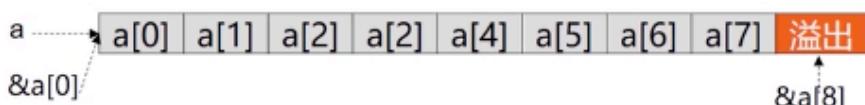
· 数组操作：

- 以数组的方式读写内存缓冲区
- 以数组的方式读写字符串
- ...

■ 数组操作常见问题

- **数组索引越界**：使用外部数据的作为数组索引时未校验其合法范围，导致读写越界问题。
- **差一错误**：当校验数值索引范围时，由于编程疏忽导致的越界读写一个元素。

■ 数组示例：数组 int a[8]，在内存中存放顺序如下：



■ 当一个指针指向数组元素时，可以指向数组末尾元素后一个元素的位置，但是不能读写该位置的内存

```
int *ptr1 = &a[8]; // 符合，指向数组末尾元素后一个元素的地址  
int *ptr2 = &a[9]; // 不符合，产生未定义行为  
int x = *ptr1; // 不符合，读越界
```

■ 数组和指针的转换

- 当表达式中出现数组标识符时，标识符类型要从数组转化为相应类型的指针类型，标识符的值被转换成数组中的第一个元素的地址（sizeof表达式除外，sizeof返回数组的大小）
- 函数参数列表中声明为数组的参数会被调整为相应类型的指针

◆典型错误-数组索引越界

【问题描述】

- 外部数据作为数组索引对内存进行访问时，必须对数据的大小进行严格的校验，否则会导致严重的错误。

【反例】

```
int Func (char *buffer, size_t bufferLen)
{
    ...
    size_t offset = ReadOffsetFromMsg(); // 读外部数据
    char c = buffer[offset];
    ...
}
```



【正例】

```
int Func (char *buffer, size_t bufferLen)
{
    ...
    size_t offset = ReadOffsetFromMsg(); // 读外部数据
    if (offset >= 0 && offset < bufferLen) {
        char c = buffer[offset];
        ...
    }
    ...
}
```



【总结说明】

- 外部数据作为数组索引时必须确保在数组大小范围内。

◆ 典型错误-不同类型对象指针之间的强制转换

【问题描述】 不同的对象类型可能有不同的对齐要求，如果转换后指针未正确对齐，会导致未定义行为。

【反例】

```
int ProcessMessage (unsigned char *data, size_t length)
{
    ... // 校验入参合法性
    Header header;
    Header *tmp = NULL;
    size_t offset = data[OFFSET_VALUE];
    ...
    // 【错误】如果指针不对齐，程序会产生未定义行为
    tmp = (Header *) (data + offset);
    memcpy (&header, tmp, sizeof(header));
    ...
}
```



【正例】

```
int ProcessMessage (unsigned char *data, size_t length)
{
    ... // 校验入参合法性
    Header header;
    size_t offset = data[OFFSET_VALUE];
    ...
    // 【重构】去除指针类型转换，直接赋值数据
    memcpy (&header, data + offset, sizeof(header));
    ...
}
```



【总结说明】

- 当在不同类型的对象指针之间强制转换时，确保指针是正确对齐的。
问题多发于内核升级，平台迁移等。

小测验

1. 指出下列代码中的错误之处:

```
void Func(void)
{
    char dst[ARRAY_SIZE];
    char src[ARRAY_SIZE];
    memset(src, '@', ARRAY_SIZE);

    size_t i;
    for(i = 0; (i < ARRAY_SIZE) && src[i] != '\0'; i++) {
        dst[i] = src[i];
    }
    dst[i] = '\0';
}
```

i 正确取值为[0, ARRAY_SIZE - 1], 当执行到此处时,
i 为ARRAY_SIZE, 导致越界写。

```
void Func(void)
{
    char dst[5];
    char src[] = "0123456789";
    strncpy(dst, src, sizeof(dst));
    printf(dst);
}
```

1. strncpy() 截断了原数据, 导致 dst 内无结束符。
2. printf 打印问题。

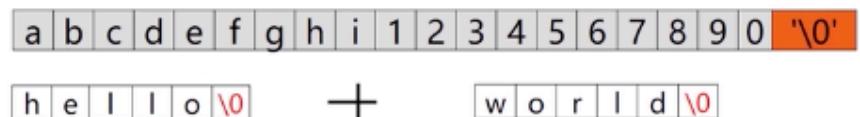
◆ 字符串操作介绍

■ 字符串及操作

- 字符串组成：字符序列 + '\0' 结束符
- 字符操作类型：
 - 字符串拷贝
 - 字符串连接
 - ...

■ 字符串操作常见问题

- 存储空间不足：**做字符串复制或拼接等操作时，没有考虑到目的缓冲区的大小是否能容纳源字符串，导致写越界问题。若两个指针指向的字符串内存空间存在重叠，则复制操作的结果不确定。
- 缺失\0结束符：**当生成的字符序列缺失\0结束符时，导致其他函数产生读越界问题，如strlen函数。例如：strncpy产生的字符串不保证有\0结束符。



◆ 字符串操作不当带来的风险

■ 拒绝服务攻击

- 由于字符串缓冲区被破坏，导致程序崩溃。如果被攻击者通过恶意输入触发这种情况，可造成拒绝服务让合法用户无法正常使用。

■ 执行任意代码

- 在典型的buffer overflow攻击中，攻击者会将大于目标缓冲区的恶意数据传入某个程序，然后程序会将这些恶意数据存储到目标堆栈缓冲区中。结果，目标堆栈上的信息就会被覆盖。如果函数返回地址被恶意代码所覆盖，那个当该函数返回时，函数的控制权便会转移给包含攻击者数据的恶意代码中。

“函数的调用在计算机中是用堆栈实现的”，这为缓冲区的利用提供了技术基础：

- ebp：基址指针寄存器，指向当前堆栈底部
- esp：堆栈(Stack)指针寄存器，指向堆栈顶部
- edi：目的变址寄存器
- esi：源变址寄存器
- ebx：基址(Base)寄存器，以它为基址访问内存

◆ 字符串操作不当带来的风险

■ 拒绝服务攻击

- 由于字符串缓冲区被破坏，导致程序崩溃。如果被攻击者通过恶意输入触发这种情况，可造成拒绝服务让合法用户无法正常使用。

■ 执行任意代码

- 在典型的buffer overflow攻击中，攻击者会将大于目标缓冲区的恶意数据传入某个程序，然后程序会将这些恶意数据存储到目标堆栈缓冲区中。结果，目标堆栈上的信息就会被覆盖。如果函数返回地址被恶意代码所覆盖，那个当该函数返回时，函数的控制权便会转移给包含攻击者数据的恶意代码中。

“函数的调用在计算机中是用堆栈实现的”，这为缓冲区的利用提供了技术基础：

- ebp：基址指针寄存器，指向当前堆栈底部
- esp：堆栈(Stack)指针寄存器，指向堆栈顶部
- edi：目的变址寄存器
- esi：源变址寄存器
- ebx：基址(Base)寄存器，以它为基址访问内存

函数栈变化及利用过程解析

栈

ebp-> ebp 返回地址 <-esp

◆ 字符串操作不当带来的风险

■ 拒绝服务攻击

- 由于字符串缓冲区被破坏，导致程序崩溃。如果被攻击者通过恶意输入触发这种情况，可造成拒绝服务让合法用户无法正常使用。

■ 执行任意代码

- 在典型的buffer overflow攻击中，攻击者会将大于目标缓冲区的恶意数据传入某个程序，然后程序会将这些恶意数据存储到目标堆栈缓冲区中。结果，目标堆栈上的信息就会被覆盖。如果函数返回地址被恶意代码所覆盖，那个当该函数返回时，函数的控制权便会转移给包含攻击者数据的恶意代码中。

“函数的调用在计算机中是用堆栈实现的”，这为缓冲区的利用提供了技术基础：

- ebp：基址指针寄存器，指向当前堆栈底部
- esp：堆栈(Stack)指针寄存器，指向堆栈顶部
- edi：目的变址寄存器
- esi：源变址寄存器
- ebx：基址(Base)寄存器，以它为基址访问内存

函数栈变化及利用过程解析



◆ 字符串操作不当带来的风险

■ 拒绝服务攻击

- 由于字符串缓冲区被破坏，导致程序崩溃。如果被攻击者通过恶意输入触发这种情况，可造成拒绝服务让合法用户无法正常使用。

■ 执行任意代码

- 在典型的buffer overflow攻击中，攻击者会将大于目标缓冲区的恶意数据传入某个程序，然后程序会将这些恶意数据存储到目标堆栈缓冲区中。结果，目标堆栈上的信息就会被覆盖。如果函数返回地址被恶意代码所覆盖，那个当该函数返回时，函数的控制权便会转移给包含攻击者数据的恶意代码中。

“函数的调用在计算机中是用堆栈实现的”，这为缓冲区的利用提供了技术基础：

- ebp：基址指针寄存器，指向当前堆栈底部
- esp：堆栈(Stack)指针寄存器，指向堆栈顶部
- edi：目的变址寄存器
- esi：源变址寄存器
- ebx：基址(Base)寄存器，以它为基址访问内存

函数栈变化及利用过程解析



◆ 字符串操作不当带来的风险

■ 拒绝服务攻击

- 由于字符串缓冲区被破坏，导致程序崩溃。如果被攻击者通过恶意输入触发这种情况，可造成拒绝服务让合法用户无法正常使用。

■ 执行任意代码

- 在典型的buffer overflow攻击中，攻击者会将大于目标缓冲区的恶意数据传入某个程序，然后程序会将这些恶意数据存储到目标堆栈缓冲区中。结果，目标堆栈上的信息就会被覆盖。如果函数返回地址被恶意代码所覆盖，那个当该函数返回时，函数的控制权便会转移给包含攻击者数据的恶意代码中。

“函数的调用在计算机中是用堆栈实现的”，这为缓冲区的利用提供了技术基础：

- ebp：基址指针寄存器，指向当前堆栈底部
- esp：堆栈(Stack)指针寄存器，指向堆栈顶部
- edi：目的变址寄存器
- esi：源变址寄存器
- ebx：基址(Base)寄存器，以它为基址访问内存



◆ 字符串操作不当带来的风险

■ 拒绝服务攻击

- 由于字符串缓冲区被破坏，导致程序崩溃。如果被攻击者通过恶意输入触发这种情况，可造成拒绝服务让合法用户无法正常使用。

■ 执行任意代码

- 在典型的buffer overflow攻击中，攻击者会将大于目标缓冲区的恶意数据传入某个程序，然后程序会将这些恶意数据存储到目标堆栈缓冲区中。结果，目标堆栈上的信息就会被覆盖。如果函数返回地址被恶意代码所覆盖，那个当该函数返回时，函数的控制权便会转移给包含攻击者数据的恶意代码中。

“函数的调用在计算机中是用堆栈实现的”，这为缓冲区的利用提供了技术基础：

- ebp：基址指针寄存器，指向当前堆栈底部
- esp：堆栈(Stack)指针寄存器，指向堆栈顶部
- edi：目的变址寄存器
- esi：源变址寄存器
- ebx：基址(Base)寄存器，以它为基址访问内存

函数栈变化及利用过程解析



◆ 字符串操作不当带来的风险

■ 拒绝服务攻击

- 由于字符串缓冲区被破坏，导致程序崩溃。如果被攻击者通过恶意输入触发这种情况，可造成拒绝服务让合法用户无法正常使用。

■ 执行任意代码

- 在典型的buffer overflow攻击中，攻击者会将大于目标缓冲区的恶意数据传入某个程序，然后程序会将这些恶意数据存储到目标堆栈缓冲区中。结果，目标堆栈上的信息就会被覆盖。如果函数返回地址被恶意代码所覆盖，那么当该函数返回时，函数的控制权便会转移给包含攻击者数据的恶意代码中。

“函数的调用在计算机中是用堆栈实现的”，这为缓冲区的利用提供了技术基础：

- ebp：基址指针寄存器，指向当前堆栈底部
- esp：堆栈(Stack)指针寄存器，指向堆栈顶部
- edi：目的变址寄存器
- esi：源变址寄存器
- ebx：基址(Base)寄存器，以它为基址访问内存

函数栈变化及利用过程解析



◆典型错误-字符串复制时的缓冲区溢出

【问题描述】 拷贝字符串时，源字符串长度可能大于目标数组空间。

【反例】

```
int main(int argc, char *argv[])
{
    char strCtx[MAX_LEN];
    ...
    // 【错误】源字符串长度可能大于目标数组空间，造成缓冲区溢出
    strcpy(strCtx, argv[1]);
}
```



【正例】

```
int main(int argc, char *argv[])
{
    char strCtx[MAX_LEN];
    ...
    // 【修改】使用strcpy_s代替strcpy来进行拷贝指定的长度，防止出现缓冲区溢出
    errno_t ret = strcpy_s(strCtx,
                           MAX_LEN,
                           argv[1]);
}
```



- 【总结说明】**
- 处理任何长度不确定的输入字符串时都要小心。
 - 对字符串复制操作前，要确保缓冲区有足够的空间容纳字符数据和'\0'结束符。

小测验

1. (单选) 下面数字中对字符串 “abcdefghi0” 最少应使用多大字符数组进行存储()

- A. 9
- B. 10
- C. 11
- D. 12

C. 字符串“abcdefghi0”本身长度为10，外加一个结束符 ‘\0’，所以应为11

3.4 函数设计

▶ 0:13 / 2:28

Note 720P 1.5x ↗



实践.2

12 函数设计

14 内存

16 其他

13 函数使用

15 文件

◆ 条款

P. 04 对所有外部数据进行合法性检查

G. FUD. 04 函数的指针参数如果不是用于修改所指向的对象就应该声明为指向const的指针

示例函数：

```
// a.h  
int Foo(const unsigned char *buffer, size_t len);  
...  
  
// a.c  
#include "a.h"  
...  
int Foo(const unsigned char *buffer, size_t len)  
{  
    int ret = 0;  
  
    if (buffer == NULL || len >= MAX_BUFFER_LEN) { // 必须做参数合法性检查  
        // 错误处理  
        ...  
    }  
    ...  
    G. FUD. 08 将字符串或指针作为函数参数时，在函数体中应检查参数是否为NULL  
    size_t nameLen = strlen((const char *)buffer, len); // buffer不一定是'\0'结尾  
    char *name = (char *)malloc(nameLen + 1);  
    if (name != NULL) {  
        memcpy_s(name, nameLen + 1, buffer, nameLen);  
        name[nameLen] = '\0';  
        foo2(name); // foo2内可以直接使用 strlen  
    }  
    ...  
    return ret;  
}
```

G. FUD. 03 函数避免使用 void* 类型参数

P. 05 函数应当合理设计返回值

G. FUD. 05 函数要简短

G. FUD. 06 内联函数避免超过10行

G. FUD. 09 避免修改函数参数的值

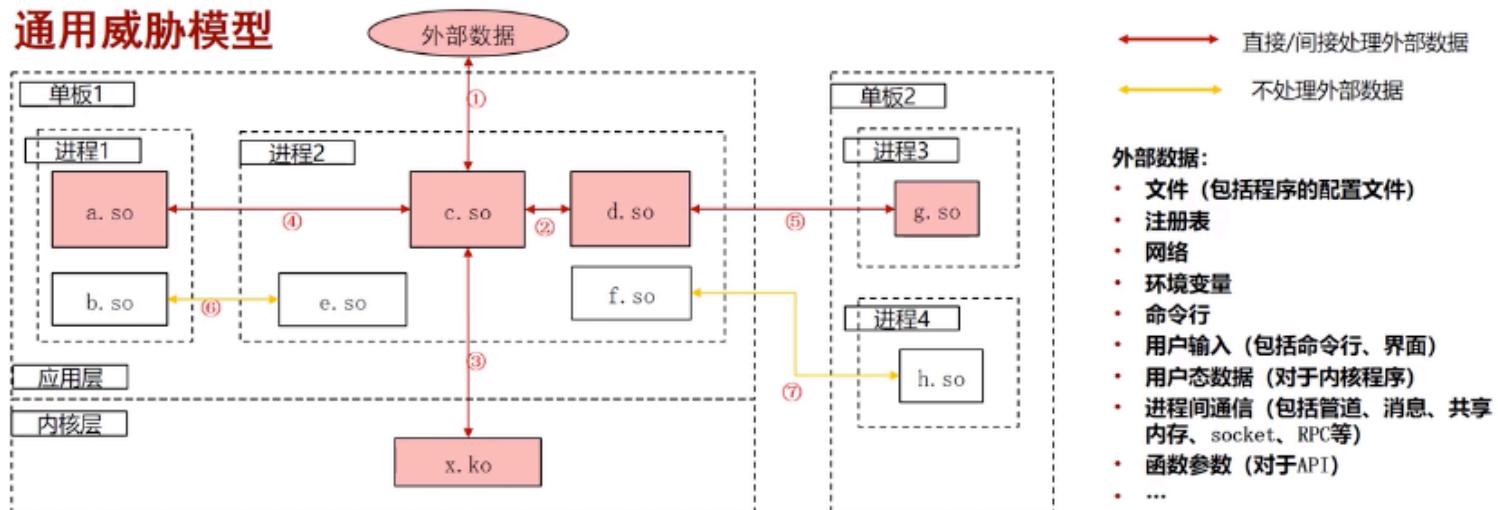
G. FUD. 10 函数只在一个文件内使用时应当使用static修饰符

3.5 函数使用与内存

函数使用与内存

◆ P.04 外部输入校验原则

通用威胁模型



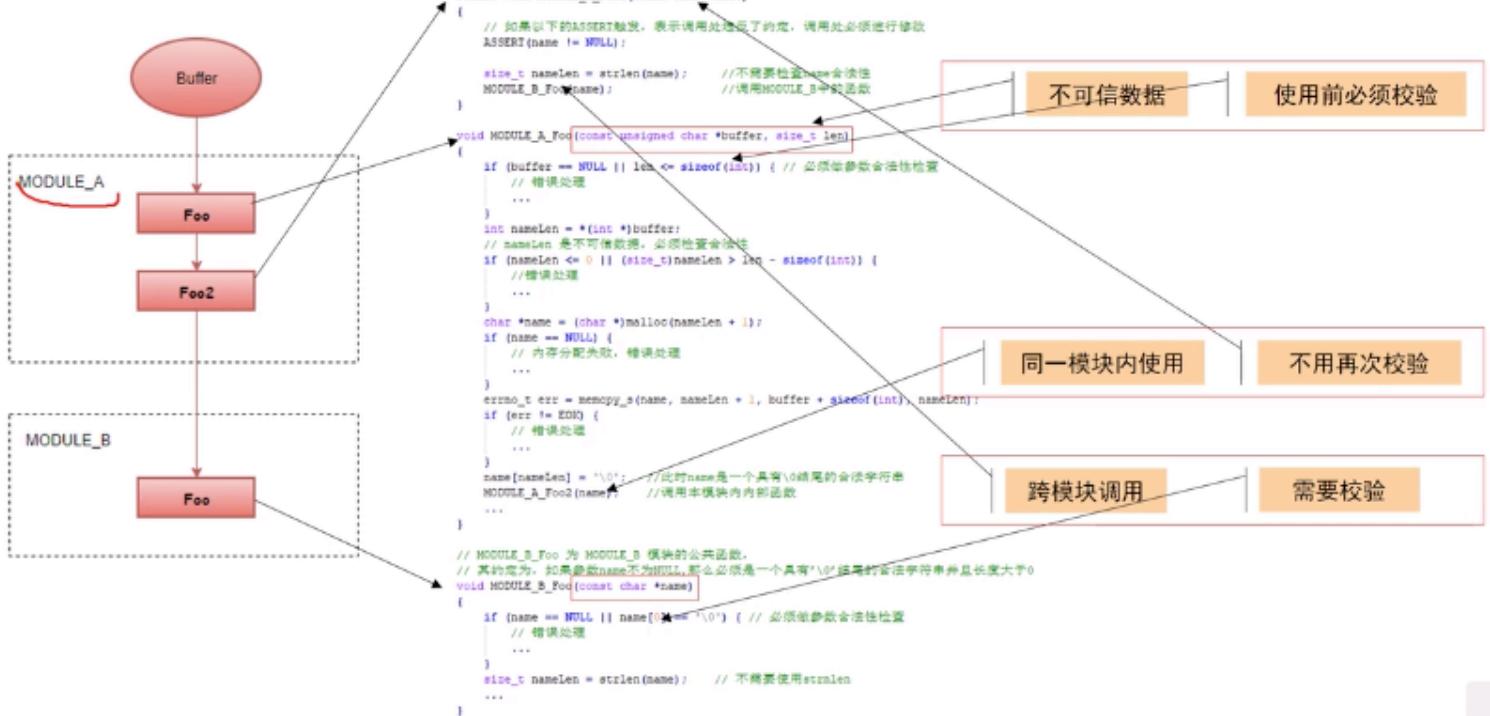
- ① 进程2中，c.so对外开放接口，直接处理外部数据
- ② 在进程2内部，d.so通过回调函数、导出函数等API接口，被c.so调用，处理外部数据
- ③ 内核模块通过ioctl向应用层提供接口，处理部分外部数据
- ④ ⑤ 进程1和进程3开放IPC接口，处理外部数据
- ⑥ ⑦ 内部IPC通信，不处理外部数据

- 外部数据：**
- 文件（包括程序的配置文件）
 - 注册表
 - 网络
 - 环境变量
 - 命令行
 - 用户输入（包括命令行、界面）
 - 用户态数据（对于内核程序）
 - 进程间通信（包括管道、消息、共享内存、socket、RPC等）
 - 函数参数（对于API）
 - ...

- 信任边界：**
- SO（或者dll）之间
 - 进程与进程之间
 - 应用层进程与操作系统内核
 - 可信执行环境内外环境

◆ 外部输入校验举例

外部数据进入到本模块后，必须经过合法性校验才能使用。被校验后的合法数据，在本模块内，后续传递到内部其他子函数，不需要重复校验。



◆ 带边界检查函数 (Bounds-checking)

- 早期内存拷贝、字符串拷贝等函数不当使用会带来缓冲区溢出等严重安全问题，ISO/IEC 9899:2011, ISO/IEC 9899:2018标准中定义了对应的带边界检查函数版本（带后缀_s），推荐使用该版本。
- 举例：常见不安全函数及对应的带边界检查函数（安全增强函数）

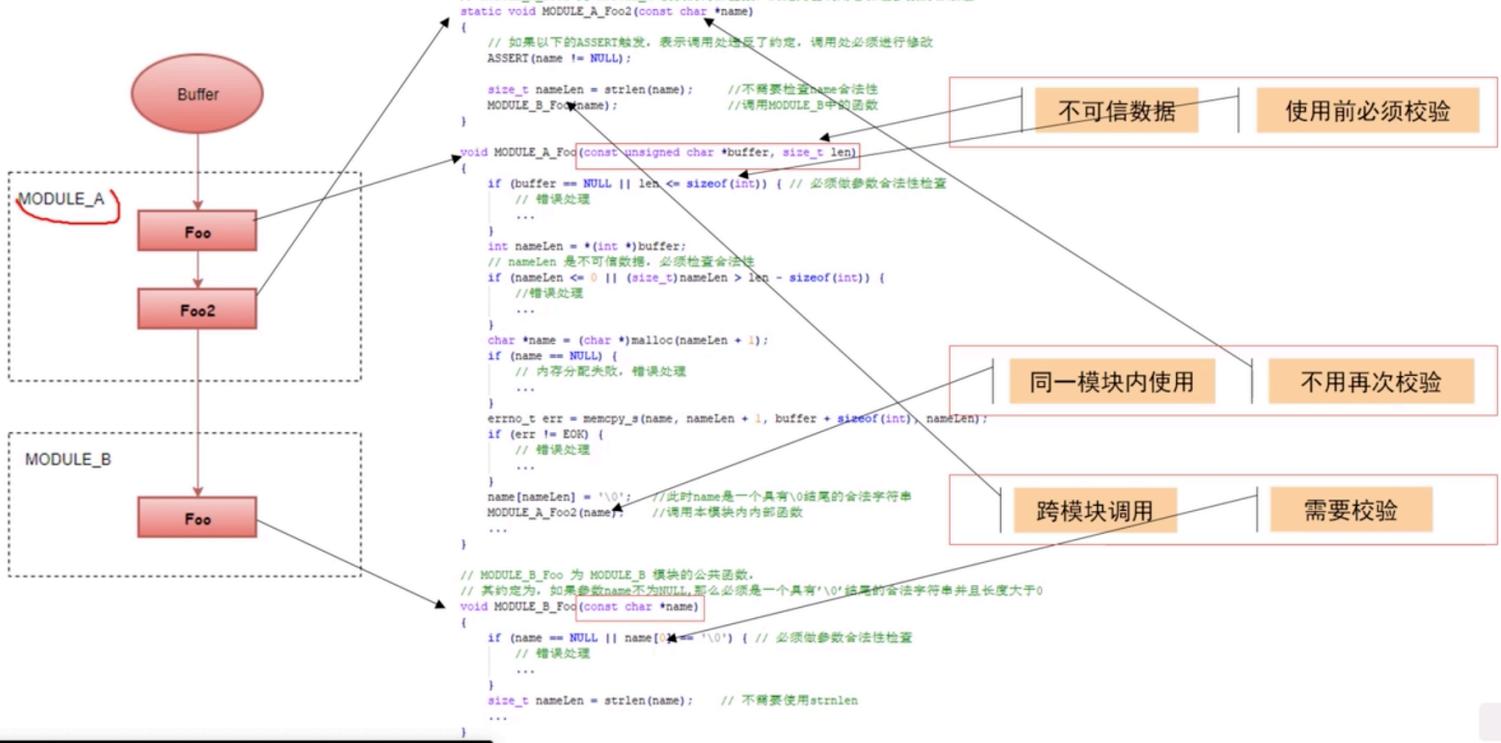
不安全函数	对应的带边界检查函数
memcpy	memcpy_s
strcpy	strcpy_s
memmove	memmove_s

不安全函数	对应的带边界检查函数
strcat	strcat_s
sprintf	sprintf_s
scanf	scanf_s

华为公司开源的带边界检查函数库（安全增强函数库）<https://gitee.com/openeuler/libboundscheck>

◆ 外部输入校验举例

外部数据进入到本模块后，必须经过合法性校验才能使用。被校验后的合法数据，在本模块内，后续传递到内部其他子函数，不需要重复校验。



◆ 格式化输入/输出函数介绍

■ 格式化输入/输出函数回顾

- C99标准中定义的格式化函数有: fprintf()、printf()、sprintf()、snprintf()、vfprintf()、vprintf()、vsprintf()、vsnprintf()等。
- 格式化字符串包括两部分内容: 普通字符和格式说明符; 普通字符将按原样输出; 格式说明符以“%”开始, 后跟一个或几个规定字符, 用来确定输出内容格式
- 参量表是需要输出的一系列参数, 其个数必须与格式化字符串所说明的输出参数个数相同, 各参数之间用“,”分开, 且顺序和类型一一对应

■ 常见问题

- 格式化参数类型不匹配, 例如:
`scanf_s("%d" , &ch); // signed char ch; 应该用 %hd`
- 格式化参数数目不匹配, 例如: `printf("%d, %s" , n);`
- 格式化字符串的长度没有限制, 例如:
`sprintf(buffer, "%s" , userInput); // userInput长度未知`
- 格式化字符串的全部或部分由用户输入, 例如:
`sprintf(buffer, userInput); // userInput来自用户输入`

Code	格式
%c	字符
%d	有符号整数
%i	有符号整数
%e	科学计数法, 使用小写“e”
%E	科学计数法, 使用大写“E”
%f	浮点数
%g	使用%e或%f中较短的一个
%G	使用%E或%f中较短的一个
%o	八进制
%s	一串字符
%u	无符号整数
%x	无符号十六进制数, 用小写字母
%X	无符号十六进制数, 用大写字母
%p	一个指针
%n	参数应该是一个指向一个整数的指针, 指向的是字符数放置的位置
%%	一个'%'符号

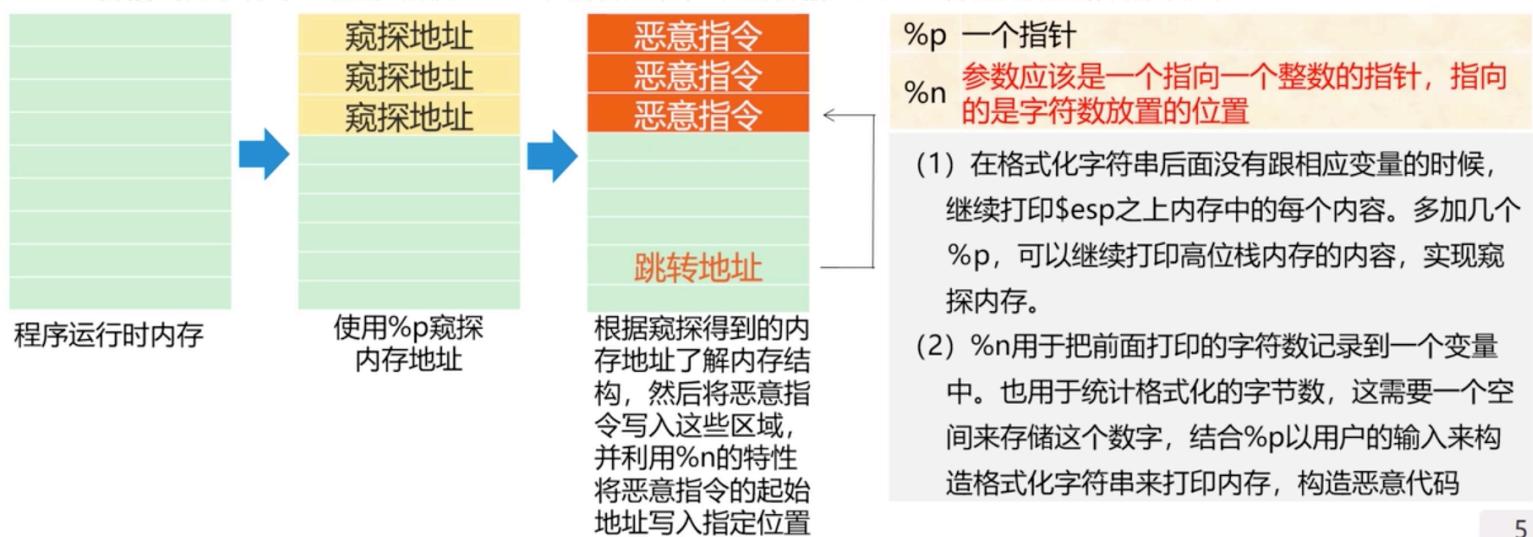
◆ 格式化缺陷的风险

■ 导致程序异常终止（拒绝服务）

- 格式说明符和参数的个数或类型不匹配会导致未定义的行为。大多数情况下，不正确的格式化缺陷可能会使程序异常终止。

■ 执行任意代码

- 若格式化字符串的全部或部分可由攻击者控制，攻击者输入的恶意数据会被当做指令执行。



◆ 格式化缺陷的风险

■ 导致程序异常终止（拒绝服务）

- 格式说明符和参数的个数或类型不匹配会导致未定义的行为。大多数情况下，不正确的格式化缺陷可能会使程序异常终止。

■ 执行任意代码

- 若格式化字符串的全部或部分可由攻击者控制，攻击者输入的恶意数据会被当做指令执行。

程序运行时内存

窥探地址	恶意指令
窥探地址	恶意指令

```
int main()
{
    int pos = 0;
    int n = 0;
    int x = 123;
    int y = 4;
    printf("%d\n%n%d", x, &pos, y);
    printf("n = %d", pos);
    ...
}
```

在Linux下编译，输出：
123 4
n = 4
N中被写入了已经打印出的字符的个数：123和一个空格，也就是4。

%p 一个指针
%n 参数应该是一个指向一个整数的指针，指向的是字符数放置的位置

(1) 在格式化字符串后面没有跟相应变量的时候，继续打印\$esp之上内存中的每个内容。多加几个%p，可以继续打印高位栈内存的内容，实现窥探内存。

(2) %n用于把前面打印的字符数记录到一个变量中。也用于统计格式化的字节数，这需要一个空间来存储这个数字，结合%p以用户的输入来构造格式化字符串来打印内存，构造恶意代码

◆ 典型错误-格式说明符与参数的类型和数量不匹配

【反例1】

```
void ArgMismatch ()  
{  
    char *errorMsg = "Resource not available to user.";  
    int errorType = ABSENT_RESOURCE_ERROR;  
    ...  
    // 【错误】格式化参数类型不匹配  
    printf( "Error (type %d): %s\n" , errorMsg, errorType);  
}
```



【正例1】

```
void ArgMismatch ()  
{  
    char *errorMsg = "Resource not available to user.";  
    int errorType = ABSENT_RESOURCE_ERROR;  
    ...  
    // 【修改】格式化参数类型匹配  
    printf( "Error (type %d): %s\n" , errorType, errorMsg );  
}
```



【反例2】

```
void Func ()  
{  
    char *errorMsg = "Resource not available to user."  
    ...  
    // 【错误】格式化参数个数不匹配  
    printf("Error (type %s)\n");  
}
```



【正例2】

```
void Func ()  
{  
    char *errorMsg = "Resource not available to user."  
    ...  
    // 【修改】使格式化参数个数匹配  
    printf("Error (type %s)\n", errorMsg );  
}
```



总结说明

- 使用格式化字符串应小心，确保格式说明符和参数在类型和数量上的匹配。格式说明符和参数之间的不匹配会导致未定义的行为。大多数情况下，不正确的格式化字符串会导致程序异常终止，甚至可以执行任意代码。

◆ 典型错误 - 使用用户输入来构造格式化字符串

【问题描述】

直接或间接将用户输入作为格式化字符串的一部分或者全部

【反例】

```
void Func (...) {  
    char input[MAX_LEN];  
    if (fgets(input, sizeof(input), stdin) == NULL) {  
        ... // 错误处理  
    }  
    printf(input); // 【错误】将用户输入直接作为了格式字符串。若其中包含 "%s" " %p" 等特殊符号时，会引起程序异常  
    ...  
}
```



【正例】

```
void Func (...) {  
    char input[MAX_LEN];  
    if (fgets(input, sizeof(input), stdin) == NULL) {  
        ... // 错误处理  
    }  
    printf("%s", input); // 【修改】通过%s将格式字符串确定下来  
    ...  
}
```



总结说明

- 禁止将用户输入作为格式化字符串的一部分或全部；
- 否则，用户输入非法字符或字符串时，可导致程序崩溃、甚至执行任意代码。

◆ 存在格式化缺陷的函数，应该谨慎使用

- 格式化输出函数: printf(), fprintf(), **sprintf()**, **snprintf()**, vprintf(), vfprintf(), **vsprintf()**, **vsnprintf()**, asprintf() (GNU扩展函数) , vasprintf() (GNU扩展函数) 及相应宽字节版本;
- 格式化输入函数: **scanf()**, **fscanf()**, **sscanf()**, **vscanf()**, **vsscanf()**, **vfscanf()**及**相应宽字节版本**;
- 格式化错误消息函数: **err()**, **verr()**, **errx()**, **verrx()**, **warn()**, **vwarn()**, **warnx()**, **vwarnx()**, **error()**, **error_at_line()**;
- 格式化日志函数: **syslog()**, **vsyslog()**.

注意: 标红为“**谨慎使用**”函数，可使用对应带边界检查函数版本（安全函数版本）

小测验

1. (判断) 可以直接使用任意用户输入来构造格式化字符串，不会带来安全风险。

错误

用户输入是不可信的，里面可能含有非法字符，因此需要校验之后再使用。

◆ 使用库函数时避免竞争条件

■ 线程不安全函数

- strtok, getenv, strerror ...
- 信号处理程序中的使用异步不安全函数： I/O函数，自定义的异步不安全函数...

■ 线程不安全函数造成危害

- 拒绝服务

◆ 使用库函数时避免竞争条件

■ 线程不安全函数

- strtok, getenv, strerror ...
- 信号处理程序中的使用异步不安全函数: I/O函数, 自定义的异步不安全函数...

■ 线程不安全函数造成危害

- 拒绝服务

使用线程不安全函数, 当程序运行在多线程环境时, 往往会导致变量被多处修改, 从而导致程序运行不正常, 甚至产生crash。

- 执行任意代码

变量被其它线程修改, 若被攻击者写入恶意数据, 则可能会导致任意代码执行。

◆ 危险的堆操作及带来的风险

■ 危险的堆操作举例

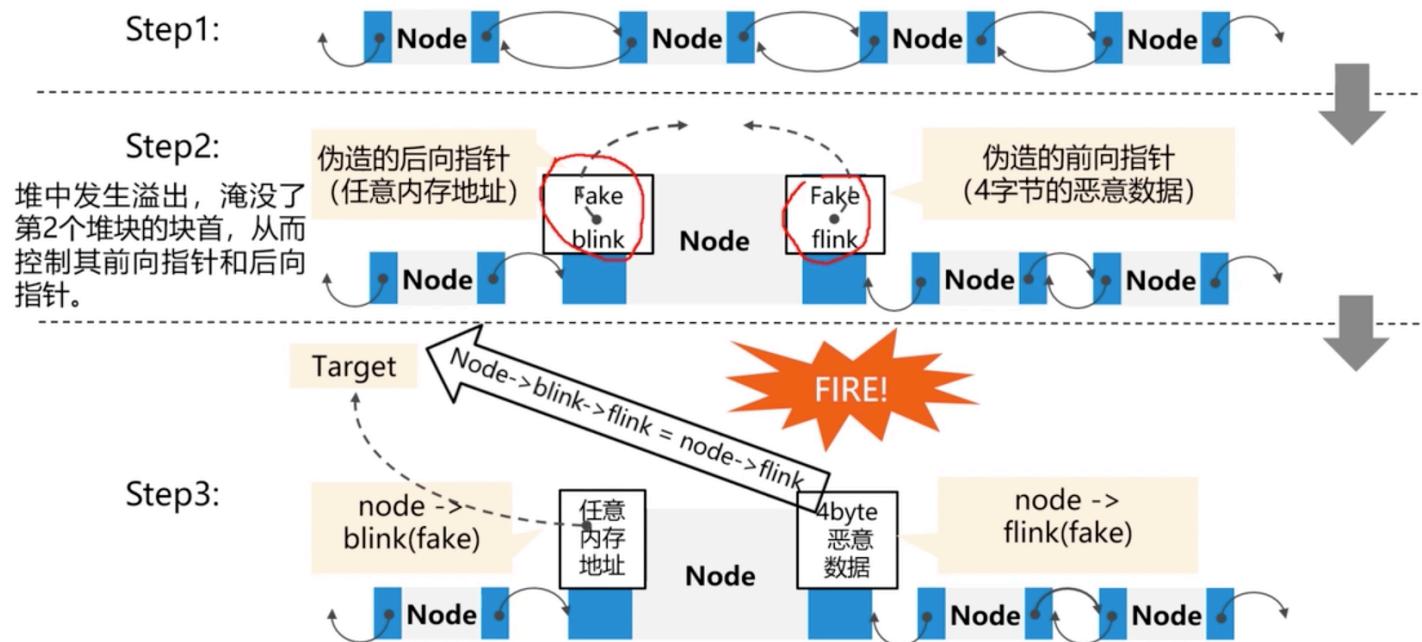
- 内存拷贝时未判断目标内存长度的有效性 (**目标内存过小**)
- 内存申请完毕后未判断空指针 (**空指针引用**)
- 使用已经释放的内存 (**UAF: use after free**)
- 调用不匹配的内存管理操作 (**new, delete与malloc, free混用**)
- 重复释放内存 (**double free**)



■ 堆管理不当带来的风险举例

- **拒绝服务攻击**: 由于堆被破坏, 可导致程序崩溃; 如果被攻击者通过恶意输入控制这种情况, 攻击者可以使程序崩溃从而让合法用户无法继续使用。
- **执行任意代码**: 攻击者输入的恶意数据被当做指令执行。

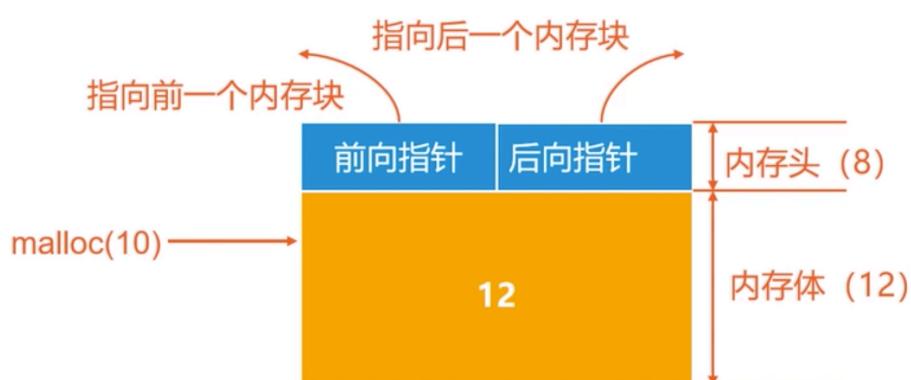
◆ 堆溢出原理 (2)



◆ 堆溢出原理 (1)

- 假设程序员申请10字节内存，windows内存管理器是这样操作的。

- 首先申请内存头 (head)，其中包含前向指针和后向指针（各4个字节，所以总共8个字节）；
- 其次申请内存体 (body)，若按照4字节对齐原则，则实际申请12个字节；
- 所以，虽然程序员只申请了10个字节，但是实际上却耗费了20($12 + 4 \times 2$) 个字节。



◆典型错误-读取未初始化的变量

【反例】

```
typedef struct {
    ...
    int value;
} CustomerMsg;

#define CUSTOMIZED_SIZE 255
void Foo(int condVal)
{
    int data;
    if (condVal > 0) {
        data = CUSTOMIZED_SIZE;
    }

    CustomerMsg msg = {0};
    msg.value = data; // 不符合: data可能未初始化
    ...
}
```



【正例】

```
typedef struct {
    ...
    int value;
} CustomerMsg;

#define CUSTOMIZED_SIZE 255
void Foo(int condVal)
{
    int data = 0; // 初始化0
    if (condVal > 0) {
        data = CUSTOMIZED_SIZE;
    }

    CustomerMsg msg = {0};
    msg.value = data; // 符合, 读取的数据值是确定的
    ...
}
```



- 在引用变量之前需要注意是否已被成功初始化；如果未初始化，变量中的数据是无法预知的，如果直接读取并使用，可能会泄露敏感信息或者其他不符合预期的结果。

- 分配内存初始化，可以消除之前可能存放在内存中的敏感信息，避免敏感信息的泄露。

【总结说明】

◆ 典型错误-访问已经释放的内存

【反例】

```
typedef struct {
    int ctx;
    ...
} MemCb;

...
MemCb *data = NULL;
... // 初始化data结构
if (DealingData(data) == NULL) { // 处理data数据
    DBG_LOG("Dealing Data Error");
    free(data);
}
... // 中间未引用data
ret = ProcessMemCbCtx(data->ctx); // 错误引用了已释放内存data中的成员
...
```



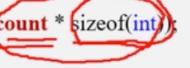
【总结说明】

- 在访问内存前需要注意该内存是否有效，是否已在其他地方被释放过；访问已经释放的内存，是很危险的行为主要分为两种情况：
 - 堆内存：一块内存释放了，归还内存池以后，就不应该再访问。因为这块内存可能已经被其他部分代码所申请，内容可能已经被修改；直接修改释放的内存，可能会导致其他使用该内存的功能不正常。
 - 栈内存：在函数执行时，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，这些存储单元自动释放。如果返回这些已释放的存储单元的地址，可能导致程序崩溃或被恶意代码利用。

◆ 典型错误-未校验申请内存大小的整数值

【反例】

```
int *FuncInterface (size_t count) // 接口函数, count来自外部
{
    /* 【错误】 malloc的参数没有进行校验,
     *      可能会使分配的内存过大或者过小, 产生意想不到的问题
     */
    int *array = (int *)malloc(count * sizeof(int));
    if (array == NULL) {
        return NULL;
    }
    ...
}
```



【正例】

```
#define MAX_COUNT 255
int *FuncInterface (size_t count) // 接口函数, count来自外部
{
    // 校验count
    if (count == 0 || count > MAX_COUNT) {
        return NULL;
    }
    int *array = (int *)malloc(count * sizeof(int));
    if (array == NULL) {
        return NULL;
    }
    ...
}
```



■ 申请内存时没有对指定的内存大小整数作合法性校验，会导致未定义的行为，主要分为3种情况：

- 使用 0 字节长度去申请内存的行为是没有定义的；
- 若申请的内存长度大于最大值，则可能导致内存申请失败，造成拒绝服务；
- 使用负数长度去申请内存，负数会被当成一个很大的无符号整数，从而导致因申请内存过大而出现失败，造成拒绝服务。

【总结说明】

错误

1. (判断) 可以使用free函数释放非动态申请的内存。

非动态申请内存是由内存管理器负责
回收管理，不需要手动释放

3.6 文件及其他

◆ 文件

■ 常见的文件操作问题

- 文件创建时没有指定合适的文件权限 → G. FIL. 01 创建文件时必须显式指定合适的文件访问权限
- 对文件的路径校验不够完全 → G. FIL. 02 使用文件路径前必须进行规范化并校验
- 共享目录中创建临时文件 → G. FIL. 03 不要在共享目录中创建临时文件

■ 文件操作不当带来的安全风险

- 重要文件丢失: 文件操作在程序中是比较复杂的, 某些不当的操作可能会导致相关文件的丢失, 甚至导致程序崩溃(比如文件指针偏移错误, 出现相对于文件头的负偏移)。
- 关键文件被不预期的访问者查看甚至执行: 如果文件的权限设置不当, 文件可能被不预期的用户访问, 引起信息泄露; 而如果是程序文件或者脚本文件, 被不预期的执行, 更会带来严重后果。

◆ 典型错误 - 输入文件路径没有进行标准化

【反例】

```
void Foo(const char *file)
{
    ...
    int fp = open(file);
    if (fp == -1) { // 未标准化, 直接使用
        ... // 错误处理
    }
    ...
    // 直接使用未标准化的文件名, 存在目录遍历的威胁
    ...
    /.../etc/passwd
```



【总结说明】

- 打开外部文件时, 需要对文件路径进行标准化 (即转换成绝对路径), 确保打开的文件是符合预期的。如果必须要使用相对路径 (比如对应用安装目录不确定), 需要对文件名中的特殊字符进行过滤, 比如“.” 和 “/”。
- 内部文件名最好能硬编码参数到应用程序中, 如果出于更高的安全要求, 需要对文件的完整性进行校验。

【正例】

```
void Func(const char *file)
{
    ...
    char *fullPath = realpath(file, NULL); ①
    if (fullPath == NULL) {
        ... // 错误处理
    }
    ② // 标准化之后, 再检查是否是安全的目录
    if (!SecureDirVerify(fullPath)) {
        ... // 错误处理
    }
    int fp = open(fullPath);
    if (fp == -1) {
        ... // 错误处理
    }
    ...
    ... // 清理资源
}
```



路径标准化

小测验

1. (判断) 为了文件操作安全考虑，应当使用文件名做为文件操作的标识。

错误

文件名操作文件极易引入条件竞争问题，从安全角度来讲，不应该使用文件名操作文件。

◆ G.MEM.04 内存中的敏感信息使用完毕后立即清0

■ 常见的敏感信息

- SessionID
- 明文口令
- 密钥

■ 敏感信息使用完毕后不及时清理，可能通过读取缓存，内存映像等方式被非法获取并利用。造成重要资产损失。

- C编译器无法识别内存清理动作，对写后未读的内存指令可能被编译器优化而丧失清理作用。

◆ G.OTh.03 禁用rand函数生成用于安全用途的伪随机数

■ 常见的随机数安全用途

- SessionID的生成
- 挑战认证算法中的随机数生成
- 验证码的随机数生成
- 生成重要随机文件（如存有系统信息等的文件）的随机文件名
- 用于密码算法用途（例如用于生成IV、盐值、密钥等）的随机数生成



■ Rand函数生成的随机数，随机性不够好，随机数易于预测，容易导致安全机制失效

◆ G.OTH.04 禁止在发布版本中输出对象或函数的地址

程序奔溃了，把“内存”
地址打印出来看看

太棒了，日志中居然能发
现目标函数地址，哈哈！



如果程序中主动输出对象或函数的地址，则为攻击者提供了便利条件，根据这些地址以及偏移量计算出其他对象或函数的地址，并实施攻击。另外，由于内存地址泄露，也会造成地址空间随机化的保护功能失效。

周三 23:23 | 开心每一天 😊

Max Score: 100 points Pass: 80 points Answered: 15/15 Time Left: 00:00:00

Name: 梁昊 Account: hao...
Examinee Account: haohfhs

Single-answer Question 12

Q 1 (6 points) Q 2 (6 points) Q 3 (5 points) Q 4 (6 points) Q 5 (6 points) Q 6 (6 points) Q 7 (6 points) Q 8 (5 points) Q 9 (6 points) Q 10 (6 points) Q 11 (6 points) Q 12 (6 points)

1、在32位系统中，如下结构体所占的字节数为0x18(每个成员所占的字节数已经标明)，
struct BBB {
 long lNum; // 4字节
 char *pcName; // 4字节
 short sDate; // 2字节
 char cHa[2]; // 1*2字节
 short sBa[6]; // 2*6字节
} *p;
假如p = 0x100000;那么
p + 0x1,(unsigned long)p + 0x1,(unsigned long *)p + 0x1,(char *)p + 0x1的值分别是多少?

A. 0x100001, 0x100001, 0x100001, 0x100004
 B. 0x100018, 0x100001, 0x100004, 0x100001
 C. 0x100018, 0x100001, 0x100004, 0x100004
 D. 0x100001, 0x100001, 0x100008, 0x100001

Congratulations! Correct Answer: B

Next Back



Chrome 文件 编辑 视图 历史记录 书签 个人资料 标签页 窗口 帮助

周三 23:23 | 开心每一天 😊

exam.edu.huawei.com/exam/#/examinationContent?lang=en_US&examinationId=107307&exam_sessionId=0&exam_sessionName=&identifyName=&ide... 更新

应用 Web Development My CS Accounts Online Course Personal Account Relax Tools ToRead ToWatch Supergeoai: COVI... 新冠加拿大疫情地... 其他书签 阅读清单

Max Score: 100 points Pass: 80 points Answered: 15/15 Time Left: 00:00:00 Submit

软件雏鹰计划 (C语言) C语言进阶综合考试

Name: 梁昊 Account: hao...

Single-answer Question 12

Q 1 (6 points)

Q 2 (6 points) ■

Q 3 (5 points)

Q 4 (6 points)

Q 5 (6 points)

Q 6 (6 points)

Q 7 (6 points)

Q 8 (5 points)

Q 9 (6 points)

Q 10 (6 points)

Q 11 (6 points)

Q 12 (6 points)

2. 在使用默认编译选项的环境中, 如下Foo函数调用后代码输出结果为:

```
int g_y;
void Foo()
{
    int x;
    int *p;
    for (int i = 0; i < 10; i++) {
        p = &i;
    }
    printf("%d %d %d", g_y, x, *p);
    return;
}
```

A. 0 0 10

B. 不确定值 0 9

C. 不确定值 不确定值 不确定值

D. 0 不确定值 不确定值

Sorry ! Correct Answer: D

■ Answered ■ Pending

Chrome 文件 编辑 视图 历史记录 书签 个人资料 标签页 窗口 帮助 (58) | (58) | 目前量 | #3 NL | Comp | Cours | MCIT | MCIT | Univ | Huawei | 综合测 Ex | 总体介 学前小 serve | Vi / V | + | 周三 23:23 | 开心每一天 😊 38% | 38% | 更新 |

exam.edu.huawei.com/exam/#/examinationContent?lang=en_US&examinationId=107307&exam_sessionId=0&exam_sessionName=&identifyName=&ide... 应用 Web Development My CS Accounts Online Course Personal Account Relax Tools ToRead ToWatch Supergeoai: COVI... 新冠加拿大疫情地... 其他书签 阅读清单

软件雏鹰计划（C语言）C语言进阶综合考试

Name: 梁昊 Account: hao...
Examinee Account: haohfhs

Max Score: 100 points Pass: 80 points Answered: 15/15 Time Left: 00:00:00 Submit

Single-answer Question 12

Q 1 (6 points)
Q 2 (6 points)
Q 3 (5 points)
Q 4 (6 points)
Q 5 (6 points)
Q 6 (6 points)
Q 7 (6 points)
Q 8 (5 points)
Q 9 (6 points)
Q 10 (6 points)
Q 11 (6 points)
Q 12 (6 points)

3. char* pszResource[] = {
 "soft disk",
 "hard disk",
 "Cray",
 "on-line drawing routhines",
 "mouse",
 "keyboard",
 "power cables"
};
pszResource[2]指向的是 ()

A. "hard disk"存放的起始地址
 B. "soft disk"中字符f存放的地址
 C. "soft disk"中字符o存放的地址
 D. "Cray"存放的起始地址

Congratulations! Correct Answer: D

Previous Next Back

Answered Pending

任务栏图标：聊天、日历、待办事项、文件夹、代码编辑器、笔记、浏览器、邮件、搜索、地图、设置、垃圾箱。

周三 23:23 | 开心每一天 😊

Max Score: 100 points Pass: 80 points Answered: 15/15 Time Left: 00:00:00

Name: 梁昊 Account: hao...
Examinee Account: haohfhs

Single-answer Question 12

Q 1 (6 points) Q 2 (6 points) Q 3 (5 points) Q 4 (6 points) Q 5 (6 points) Q 6 (6 points) Q 7 (6 points) Q 8 (5 points) Q 9 (6 points) Q 10 (6 points) Q 11 (6 points) Q 12 (6 points)

4. 以下程序的输出结果是 ()

```
#include <stdio.h>
#define PT 3.5
#define S(X) PT *X *X
void main()
{
    int a = 1, b = 2;
    printf("%g\n", S(a + b));
}
```

A. 7.5
 B. 10.5
 C. 14.0
 D. 31.5

Sorry ! Correct Answer: A

Previous Next Back

Answered Pending

<https://exam.edu.huawei.com/exam/>

Chrome 文件 编辑 视图 历史记录 书签 个人资料 标签页 窗口 帮助 1 KB/s 0 KB/s 59° 94% 34% 周三 23:23 | 开心每一天 😊 38% 更新

exam.edu.huawei.com/exam/#/examinationContent?lang=en_US&examinationId=107307&exam_sessionId=0&exam_sessionName=&identifyName=&ide... 应用 Web Development My CS Accounts Online Course Personal Account Relax Tools ToRead ToWatch Supergeoai: COVI... 新冠加拿大疫情地... 其他书签 阅读清单

软件雏鹰计划（C语言）C语言进阶综合考试

Name: 梁昊 Account: hao... Max Score: 100 points Pass: 80 points Answered: 15/15 Time Left: 00:00:00 Examneee Account: haohfhs Submit

Single-answer Question 12

Q 1 (6 points) Q 2 (6 points) Q 3 (5 points) Q 4 (6 points) Q 5 (6 points) Q 6 (6 points) Q 7 (6 points) Q 8 (5 points) Q 9 (6 points) Q 10 (6 points) Q 11 (6 points) Q 12 (6 points)

5、在一个四字节对齐的32位系统中，

```
struct tagSmart {  
    char flag1;  
    int (*left_tree)[3];  
    struct tagSmart *right_tree[2];  
    char flag3;  
    char flag4;  
} SmartFlag[4];
```

sizeof(SmartFlag) = ?

A. 128
 B. 64
 C. 80
 D. 96

Sorry ! Correct Answer: C

Previous Next Back

Answered Pending



Chrome 文件 编辑 视图 历史记录 书签 个人资料 标签页 窗口 帮助 2 1 KB/s 0 KB/s 52° 94% 23% 周三 23:24 | 开心每一天 😊 38% 更新

exam.edu.huawei.com/#/examinationContent?lang=en_US&examinationId=107307&exam_sessionId=0&exam_sessionName=&identifyName=&id... 应用 Web Development My CS Accounts Online Course Personal Account Relax Tools ToRead ToWatch Supergeoai: COVI... 新冠加拿大疫情地... 其他书签 阅读清单

软件雏鹰计划（C语言）C语言进阶综合考试

Name: 梁昊 Account: hao... Exam Score: 100 points Pass: 80 points Answered: 15/15 Time Left: 00:00:00

Examinee Account: haohfhs

Single-answer Question 12

Q 1 (6 points) Q 2 (6 points) Q 3 (5 points) Q 4 (6 points) Q 5 (6 points) Q 6 (6 points) Q 7 (6 points) Q 8 (5 points) Q 9 (6 points) Q 10 (6 points) Q 11 (6 points) Q 12 (6 points)

Multiple choice single answer 6/15Question

6、在64位系统中，
char acNew[] = "Welcome\0To\0Huawei\0"; sizeof(acNew)和strlen(acNew) 的值分别是多少？

A. 19 7
 B. 18 8
 C. 18 7
 D. 8 8

Sorry ! Correct Answer: A

Previous Next Back

Answered Pending

Chrome 文件 编辑 视图 历史记录 书签 个人资料 标签页 窗口 帮助

exam.edu.huawei.com/exam/#/examinationContent?lang=en_US&examinationId=107307&exam_sessionId=0&exam_sessionName=&identifyName=&ide...

Max Score: 100 points Pass: 80 points Answered: 15/15 Time Left: 00:00:00

Name: 梁昊 Account: hao...

Single-answer Question 12

Q 1 (6 points) Q 2 (6 points) Q 3 (5 points) Q 4 (6 points) Q 5 (6 points) Q 6 (6 points) Q 7 (6 points) Q 8 (5 points) Q 9 (6 points) Q 10 (6 points) Q 11 (6 points) Q 12 (6 points)

Multiple choice single answer 7/15Question

7、下列程序执行后的输出结果是（ ）。

```
signed char x = 0xFFFF;
printf("%d \n", x--);
```

A. -32767
 B. FFFE
 C. -2
 D. -32768

Sorry ! Correct Answer: C

Previous Next Back

Answered Pending

Chrome 文件 编辑 视图 历史记录 书签 个人资料 标签页 窗口 帮助

exam.edu.huawei.com/exam/#/examinationContent?lang=en_US&examinationId=107307&exam_sessionId=0&exam_sessionName=&identifyName=&ide...

Max Score: 100 points Pass: 80 points Answered: 15/15 Time Left: 00:00:00

Name: 梁昊 Account: hao...

Single-answer Question 12

Q 1 (6 points) Q 2 (6 points) Q 3 (5 points) Q 4 (6 points) Q 5 (6 points) Q 6 (6 points) Q 7 (6 points) Q 8 (5 points) Q 9 (6 points) Q 10 (6 points) Q 11 (6 points) Q 12 (6 points)

Multiple choice single answer 8/15Question

8、有如下枚举定义：
typedef enum
{
 E12 = 2,
 E13,
 E14 = 5,
 E15 = 5,
}ENUM_N;
请问E12~E15分别是多少

A. 2,3,5,6
 B. 2,3,5,5
 C. 2,3,4,5
 D. 3,4,5,6

Congratulations! Correct Answer: B

Answered Pending

Previous Next Back

Chrome 文件 编辑 视图 历史记录 书签 个人资料 标签页 窗口 帮助

exam.edu.huawei.com/exam/#/examinationContent?lang=en_US&examinationId=107307&exam_sessionId=0&exam_sessionName=&identifyName=&ide...

Max Score: 100 points Pass: 80 points Answered: 15/15 Time Left: 00:00:00

Name: 梁昊 Account: hao...

Single-answer Question 12

Q 1 (6 points) Q 2 (6 points) Q 3 (5 points) Q 4 (6 points) Q 5 (6 points) Q 6 (6 points) Q 7 (6 points) Q 8 (5 points) Q 9 (6 points) Q 10 (6 points) Q 11 (6 points) Q 12 (6 points)

Multiple choice single answer 9/15Question

9、下列程序执行后的输出结果是（ ）。

```
void main()
{
    int a[3][3], *p, i;
    p = &a[0][0];
    for (i = 0; i < 9; i++)
        p[i] = i + 1;
    printf("%d \n", a[1][2]);
}
```

A. 6
 B. 3
 C. 随机数
 D. 9

Congratulations! Correct Answer: A

Arrow keys available to switch questions

Previous Next Back

Chrome 文件 编辑 视图 历史记录 书签 个人资料 标签页 窗口 帮助

exam.edu.huawei.com/exam/#/examinationContent?lang=en_US&examinationId=107307&exam_sessionId=0&exam_sessionName=&identifyName=&ide...

Max Score: 100 points Pass: 80 points Answered: 15/15 Time Left: 00:00:00

Submit

软件雏鹰计划 (C语言) C语言进阶综合考试

Examinee Account: haohfhs

Multiple choice single answer 10/15 Question

Arrow keys available to switch questions

Q 3 (5 points)

Q 4 (6 points)

Q 5 (6 points)

Q 6 (6 points)

Q 7 (6 points)

Q 8 (5 points)

Q 9 (6 points)

Q 10 (6 points)

Q 11 (6 points)

Q 12 (6 points)

Multi-answer Question 3

Q 13 (10 points)

Q 14 (10 points)

Q 15 (10 points)

Congratulations! Correct Answer: C

Previous Next Back

Answered Pending

10. 有一个char *类型的指针, 恰好指向了一个int, 想让这个指针跳过int指向下一个char, 下面的代码可以达到这个目的吗? ()
1) ((int *)p)++;
2) p += sizeof(int);

A. 都不可以
 B. 都可以
 C. 只有2可以
 D. 只有1可以

Chrome 文件 编辑 视图 历史记录 书签 个人资料 标签页 窗口 帮助

exam.edu.huawei.com/exam/#/examinationContent?lang=en_US&examinationId=107307&exam_sessionId=0&exam_sessionName=&identifyName=&ide...

Max Score: 100 points Pass: 80 points Answered: 15/15 Time Left: 00:00:00

Software Eagle Plan (C Language) C Language Advanced Comprehensive Exam

Examinee Account: haohfhs

Multiple choice single answer 11/15 Question

Arrow keys available to switch questions

Q 3 (5 points)

Q 4 (6 points)

Q 5 (6 points)

Q 6 (6 points)

Q 7 (6 points)

Q 8 (5 points)

Q 9 (6 points)

Q 10 (6 points)

Q 11 (6 points)

Sorry! Correct Answer: D

Q 12 (6 points)

Multi-answer Question 3

Previous Next Back

Q 13 (10 points)

Q 14 (10 points)

Q 15 (10 points)

Answered Pending

周三 23:24 | 开心每一天 😊

Max Score: 100 points Pass: 80 points Answered: 15/15 Time Left: 00:00:00

Software Falcon Plan (C Language) C Language Advanced Comprehensive Exam

Examinee Account: haohfhs

Q 3 (5 points)

Q 4 (6 points)

Q 5 (6 points)

Q 6 (6 points)

Q 7 (6 points)

Q 8 (5 points)

Q 9 (6 points)

Q 10 (6 points)

Q 11 (6 points)

Q 12 (6 points)

Multi-answer Question 3

Q 13 (10 points)

Q 14 (10 points)

Q 15 (10 points)

Answered Pending

12. 在X86下, 有下列程序

```
#include <stdio.h>
void main()
{
    union {
        long k;
        char i[2];
    } * s, a;
    s = &a;
    s->i[0] = 0x39;
    s->i[1] = 0x38;
    printf("%x\n", a.k);
}
```

A. 不可预知

B. 3938

C. 3839

D. 380039

Sorry ! Correct Answer: A



Chrome 文件 编辑 视图 历史记录 书签 个人资料 标签页 窗口 帮助

exam.edu.huawei.com/exam/#/examinationContent?lang=en_US&examinationId=107307&exam_sessionId=0&exam_sessionName=&identifyName=&ide...

Max Score: 100 points Pass: 80 points Answered: 15/15 Time Left: 00:00:00

Software Eagle Plan (C Language) C Language Advanced Comprehensive Exam

Examinee Account: haohfhs

Q 3 (5 points) Multiple choice multiple answer 13/15 Question

Q 4 (6 points)

Q 5 (6 points)

Q 6 (6 points)

Q 7 (6 points)

Q 8 (5 points)

Q 9 (6 points)

Q 10 (6 points)

Q 11 (6 points)

Q 12 (6 points)

Multi-answer Question 3

Q 13 (10 points)

Q 14 (10 points)

Q 15 (10 points)

Answered Pending

13. In C language, which of the following is a valid expression?

```
unsigned int a=20;
int b=13;
int k = b-a;
```

A. k<(int)(b+a)
 B. k<(unsigned int)b+a
 C. k>b+a
 D. k<b+(int)a

Sorry! Correct Answer: ACD

Previous Next Back

Chrome 文件 编辑 视图 历史记录 书签 个人资料 标签页 窗口 帮助

exam.edu.huawei.com/exam/#/examinationContent?lang=en_US&examinationId=107307&exam_sessionId=0&exam_sessionName=&identifyName=&ide...

Max Score: 100 points Pass: 80 points Answered: 15/15 Time Left: 00:00:00

Submit

Examinee Account: haohfhs

Multiple choice multiple answer 14/15 Question

Arrow keys available to switch questions

Q 3 (5 points)

Q 4 (6 points)

Q 5 (6 points)

Q 6 (6 points)

Q 7 (6 points)

Q 8 (5 points)

Q 9 (6 points)

Q 10 (6 points)

Q 11 (6 points)

Q 12 (6 points)

Multi-answer Question 3

Q 13 (10 points)

Q 14 (10 points)

Q 15 (10 points)

Answered Pending

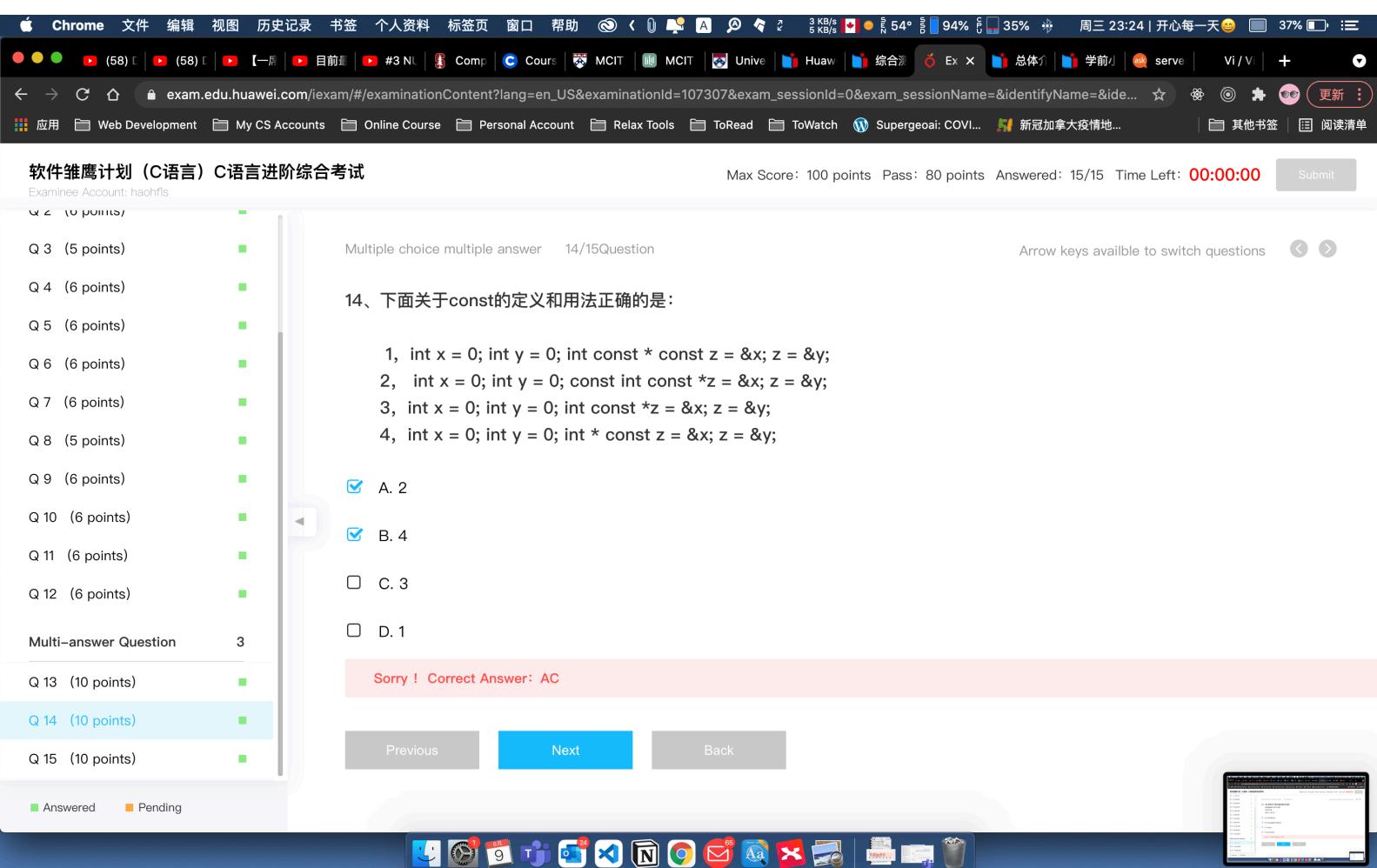
14. 下面关于const的定义和用法正确的是：

1, int x = 0; int y = 0; int const * const z = &x; z = &y;
2, int x = 0; int y = 0; const int const *z = &x; z = &y;
3, int x = 0; int y = 0; int const *z = &x; z = &y;
4, int x = 0; int y = 0; int * const z = &x; z = &y;

A. 2
 B. 4
 C. 3
 D. 1

Sorry ! Correct Answer: AC

Previous Next Back



周三 23:24 | 开心每一天 😊

Max Score: 100 points Pass: 80 points Answered: 15/15 Time Left: 00:00:00

Software Eagle Plan (C Language) C Language Advanced Comprehensive Exam

Examinee Account: haohfhs

Q 1 (5 points)

```
int main()
{
    char *buf = NULL;
    buf = GetMemory();
    printf(buf);
    free(buf);
    return 0;
}
```

Q 2 (5 points)

Q 3 (5 points)

Q 4 (6 points)

Q 5 (6 points)

Q 6 (6 points)

Q 7 (6 points)

Q 8 (5 points)

Q 9 (6 points)

Q 10 (6 points)

Q 11 (6 points)

Q 12 (6 points)

Multi-answer Question 3

Q 13 (10 points)

Q 14 (10 points)

Q 15 (10 points)

Sorry ! Correct Answer: ACD

Previous Back

Answered Pending



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)

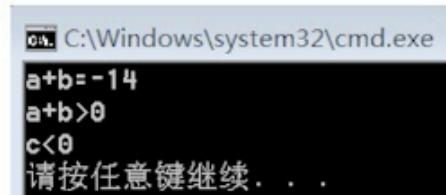
```
void main(void)
{
    unsigned int a=6;
    int b=-20;
    int c=a+b;

    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

```
$00114095@CTU\>
$ 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;



3) 12
typedef struct
{
 U32 a;
 TEST_1 b;
}TEST_3;

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

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

基本概念:

- 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。
- 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。
- 指定对齐值: #pragma pack (value)时指定的对齐value。
- 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:

- 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。
- 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。
- 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



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)

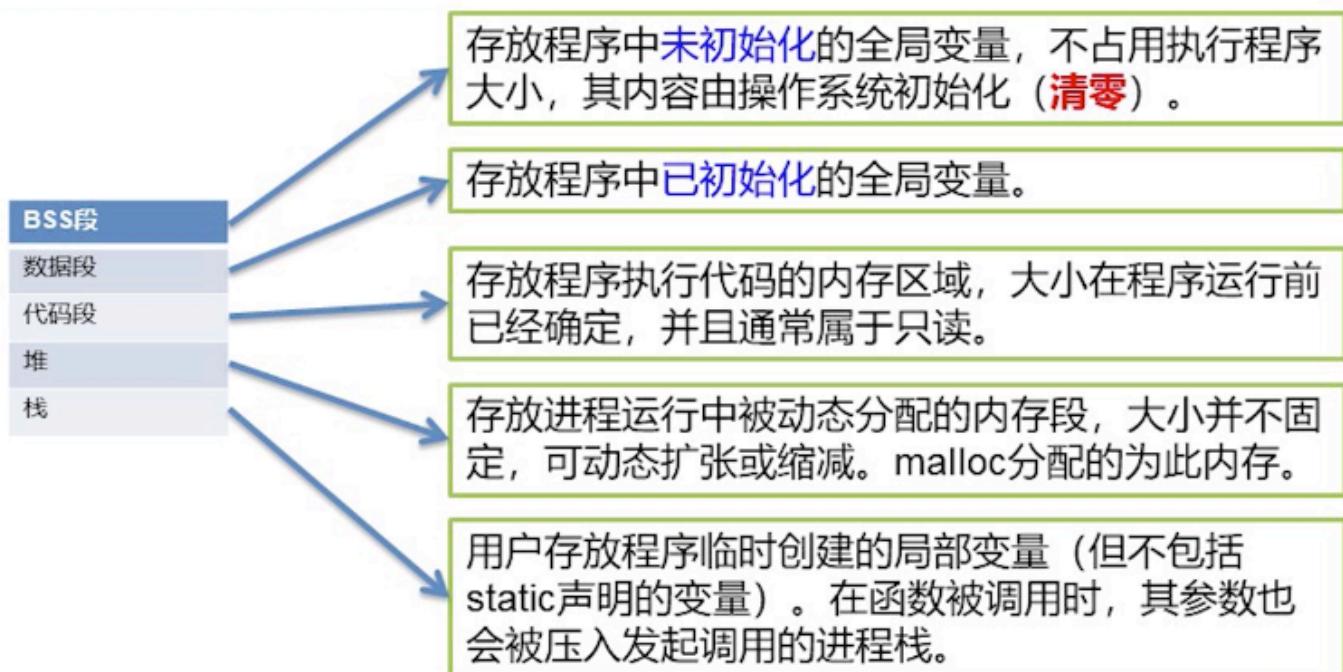
```
1 // Copyright (c) Huawei Technologies Co., Ltd. 2012-2018. All rights reserved.
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6
7 int a = 0x0506; // 全局初始化区 - 数据段
8 char *p0 = "this is test string";
9 char *p1; // 全局未初始化区 --BSS段
10
11 int main()
12 {
13     int b; // 局部变量 - 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 ...@.....

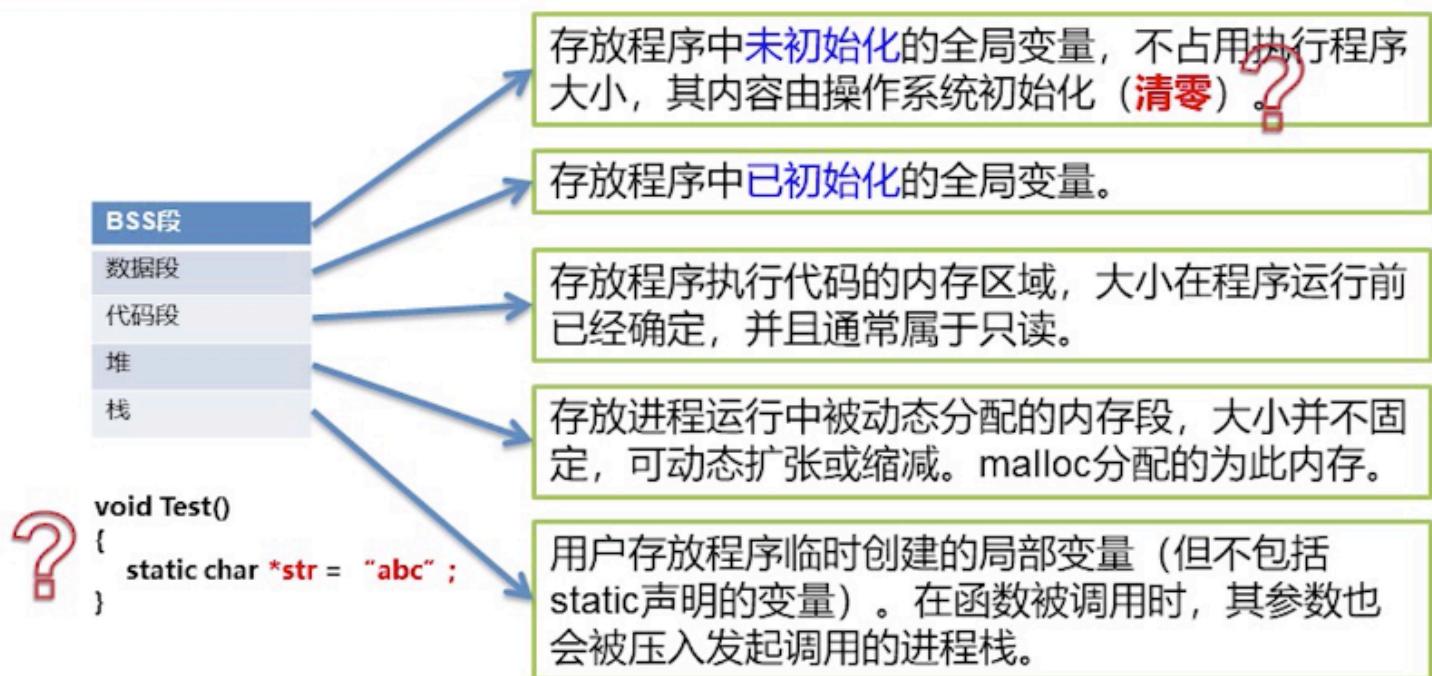
Contents of section .rodata:
4006c0 01000200 74686973 20697320 74657374this is test
4006d0 20737472 696e6700 31323334 00616464 string.1234.add
4006e0 72203d20 25700a00 61626364 00 r = %p..abcd.

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')