**第 6周作业 C语言程序的转化**

1. 什么是应用程序二进制接口？ 举出三种不同的接口示例。

**答：**应用程序二进制接口（Application Binary Interface ，ABI）是定义编译后的二进制程序组件之间交互规则的底层接口规范。它确保应用程序的不同模块、库函数、操作系统能够正确协同工作。

ABI是二进制程序交互的“隐形契约”，涉及从寄存器使用到内存布局的底层细节。它是软件生态中二进制兼容性的基石，直接影响程序的移植性、库的复用性及系统的稳定性。理解ABI有助于解决跨平台开发、库依赖管理等复杂问题。  
 接口示例：

函数调用约定：包括参数传递方式是使用堆栈传递参数，还是使用寄存器传递参数；参数是从右到左的顺序放入堆栈，还是从左到右的顺序放入堆栈；各个参数分别放在什么寄存器中；返回值存储在什么位置；寄存器是由调用者保存（保护）还是由被调用者保存（保护）等等。

数据类型表示的约定：包括基本类型的大小（如int为4字节）、对齐方式（如结构体成员对齐到自然边界）、复杂类型（结构体、联合体）的内存布局、浮点数格式（如IEEE 754标准）等等。

系统调用约定：包括操作系统通过内核和运行时库定义系统调用、动态链接等底层接口。

在32位、64位、Windows、linux系统中，这些约定并不相同。ABI标准包括Microsoft x64 ABI（Windows）、System V AMD64 ABI（Unix）；操作系统相关的ABI 有Linux Standard Base (LSB)、Microsoft ABI；编程语言相关的ABI标准有C ABI、Microsoft C++ ABI等。

1. 程序运行时，检测缓冲区溢出的常用手段是什么？

**答：**

（1）栈保护（Stack Canary）

编译器在函数栈帧中插入一个随机值（Canary），位于缓冲区与返回地址之间。函数返回前检查该值是否被修改。

（2）动态内存检测工具

AddressSanitizer (ASan)是一种内存错误检测工具，通过编译时插桩和运行时检查，监控内存访问。使用“红区”（Red Zones）隔离缓冲区，通过影子内存（Shadow Memory）标记内存状态。ASan会监测内存访问，一旦发现内存访问错误，比如越界访问、释放后再次访问等，会立即输出错误信息并中断程序执行，同时提供详细的报告帮助开发者定位问题的源头。

（3）使用安全增强的库函数，防范缓冲区溢出

替换危险函数（如 strcpy、sprintf）为带边界检查的版本（如 strncpy、snprintf 、strcpy\_s 等）。

（4）操作系统级防护缓冲区溢出攻击

随机化内存地址（如栈、堆、库的基址），即地址空间布局随机化 (Address Space Layout Randomization， ASLR)，增加攻击者预测难度。

1. 程序运行时，如何检测变量未初始化就被使用？

答：

代码内显式跟踪状态，为变量添加初始化标记（为被监测的变量增加一个匿名变量，用于记录是否进行过初始化），在给变量赋值时，匿名变量就改为已初始化；在使用被监测变量时，检查对应的匿名变量，看是否进行过初始化。

Valgrind (Memcheck)：检测未初始化内存的读取

AddressSanitizer (ASan)：编译时添加标志检测未初始化使用

1. 使用递归程序，存在什么风险？

答：

（1） 栈溢出（Stack Overflow）

每次递归调用都会在内存的调用栈中占用空间。如果递归深度过大（例如无限递归或未正确终止），栈空间可能被耗尽，导致程序崩溃，出现 StackOverflow错误。

（2）性能低下

每次递归调用需要保存函数状态（参数、返回地址等），时间和空间复杂度增长，性能低下。

1. 设有 int x; char \*ptr;

mov eax, dword ptr [ptr];

mov ebx, dword ptr [eax]

mov dword ptr [x], ebx  
上述 三条机器指令对应 的 C语句是什么？

下面三条机器指令对应 的 C语句又是什么？

mov eax, dword ptr [ptr];

movsx ebx, byte ptr [eax]

mov dword ptr [x], ebx

答：

前三条汇编语句对应的C语句为 ： x = \*(int \*)ptr;

后三条汇编语句对应的C语句为 ： x = (int) \*ptr; 或者 x=\*ptr;

1. 将C 语句补充完整？  
    **char str1[12] = "\_\_\_\_\_\_\_\_\_\_\_\_\_"; char str1[12] = "abcdefghijk";**

00D815E3 mov eax,dword ptr [0D85B30h]   


00D815E8 mov dword ptr [str1],eax

00D815EB mov ecx,dword ptr ds:[0D85B34h]

00D815F1 mov dword ptr [ebp-8],ecx

00D815F4 mov edx,dword ptr ds:[0D85B38h]

00D815FA mov dword ptr [ebp-4],edx

**char str2[12];**

**for (int i = \_\_\_\_\_\_\_;\_\_\_\_\_\_\_\_\_; \_\_\_\_\_\_\_\_\_)**

**for (int i =0;i<11; i++)**

00D815FD mov dword ptr [ebp-1Ch],0

00D81604 jmp func+3Fh (0D8160Fh)

00D81606 mov eax,dword ptr [ebp-1Ch]

00D81609 add eax,1

00D8160C mov dword ptr [ebp-1Ch],eax

00D8160F cmp dword ptr [ebp-1Ch],0Bh

00D81613 jge func+59h (0D81629h)

**\_\_\_\_\_\_\_\_\_ = \_\_\_\_\_\_\_\_\_\_; str1[i]=str2[i] & 0x0F;**

00D81615 mov eax,dword ptr [ebp-1Ch]

00D81618 movsx ecx,byte ptr str1[eax]

00D8161D and ecx,0Fh

00D81620 mov edx,dword ptr [ebp-1Ch]

00D81623 mov byte ptr str2[edx],cl

00D81627 jmp func+36h (0D81606h)

**\_\_\_\_\_\_\_\_\_\_\_\_\_; str2[11]=0;**

00D81629 mov eax,1

00D8162E imul ecx,eax,0Bh

00D81631 mov byte ptr str2[ecx],0

1. 写出与第 6题功能等价的优化代码（汇编语言写，可以直接用变量名、标号、不要用数字化的指令地址）。解释代码提高运行速度的原因。注意，有多种优化方案（多种优化策略；写出多种策略者加分）

方案1：

; 给 str1 数组赋值

mov rax, 6867666564636261h

mov qword ptr [str1], rax; ; qword ptr, 是 4字(8字节)类型

mov ebx, 006b6a69h

mov dword ptr [str1+8], ebx ; dword ptr, 是 2字(4字节)类型

; 给 str2 数组赋值

mov rcx, 0f0f0f0f0f0f0f0fh  
 and rax, rcx

mov qword ptr [str2], rax

and ebx, 000f0f0fh

mov dword ptr [str2+8], ebx  
优化策略：

① 循环展开，消除了循环操作。优化前的循环体中有11条机器指令，要循环11次，共计执行 121条指令；此外还要加上最后一次跳出循环的几条指令以及循环结束后的指令。优化程序只会执行 9条指令。  
 ② 使用了位数更多的寄存器。使用 rax 等 8字节寄存器，一次就可以操作8个字节，比一次一个字节的操作，大幅度节省了操作次数。  
 ③ 变量与寄存器绑定。两次都涉及到 str1，将其前8个字节的内容与rax绑定，节省了从内存读数据的开销。

方案2：寄存器和变量绑定，用 EAX 对应 变量 i。循环程序可简化为：  
 (循环体 由 11条语句变成 7条语句)

for (int i =0;i<11; i++)

mov eax,0

jmp condition\_judge

modify:

add eax,1

condition\_judge:

cmp eax,0Bh

jge for\_over

movsx ecx,byte ptr str1[eax]

and ecx,0Fh

mov byte ptr str2[eax],cl

jmp modify

**for\_over:**

mov byte ptr str2[eax],0