

# 系统及软件体系设计作业 • Unit 0

王旭东

2014302580343

2016 年 9 月 21 日

## 1 ANSI C

ANSI C 是美国国家标准协会 (ANSI) 对 C 语言发布的标准。使用 C 的软件开发被鼓励遵循 ANSI C 文档的要求, 因为它鼓励使用跨平台的代码。

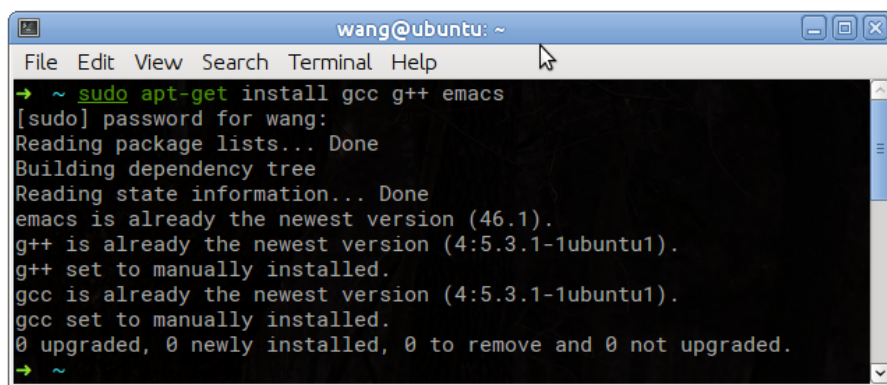
C 的第一个标准是由 ANSI 发布的。虽然这份文档后来被国际标准化组织 (ISO) 采纳并且 ISO 发布的修订版也被 ANSI 采纳了, 但名称 ANSI C (而不是 ISO C) 仍被广泛使用。一些软件开发使用名称 ISO C, 还有一些使用中立的名称 Standard C。

ANSI C 现在被几乎所有广泛使用的编译器支持。现在多数 C 代码是在 ANSI C 基础上写的。任何仅仅使用标准 C 并且没有任何硬件依赖假设的代码实际上能保证在任何平台上用遵循 C 标准的编译器编译成功。如果没有这种预防措施, 多数程序只能在一种特定的平台或特定的编译器上编译, 例如, 使用非标准库, 例如图形用户界面库, 或者有关编译器或平台特定的特性例如数据类型的确切大小和字节序。

常见的支持 ANSI C 的编译器包括 GCC、Microsoft Visual C++、clang 等。

## 2 开发环境安装

Debian 系的 GNU/Linux 使用 apt 软件管理器安装软件, 图 1 是安装编译器和编辑器的截图 (事实上已安装好)。



```
wang@ubuntu: ~  
File Edit View Search Terminal Help  
→ ~ sudo apt-get install gcc g++ emacs  
[sudo] password for wang:  
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
emacs is already the newest version (46.1).  
g++ is already the newest version (4:5.3.1-1ubuntu1).  
g++ set to manually installed.  
gcc is already the newest version (4:5.3.1-1ubuntu1).  
gcc set to manually installed.  
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.  
→ ~
```

图 1: 安装 gcc、g++ 和 Emacs 编辑器

## 3 预处理、编译、汇编、链接

为排版美观考虑, 没有将图片集中一处。此题截图为图 2、3、4、5、6。详细说明见图片脚注。

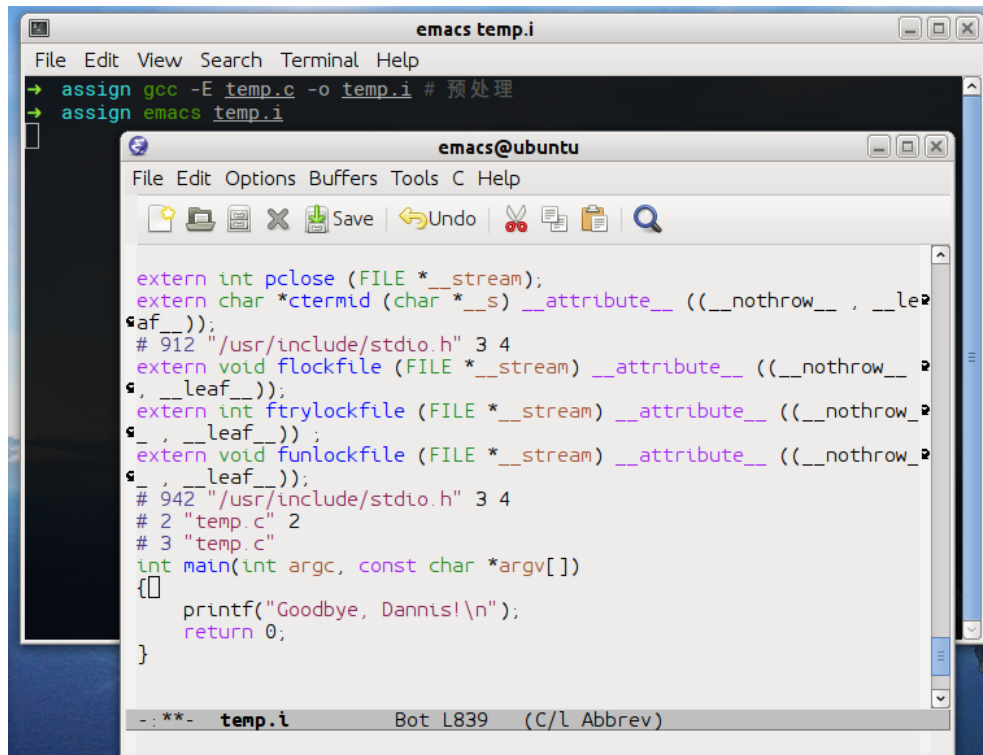


图 2: 预处理: 此时进行宏展开

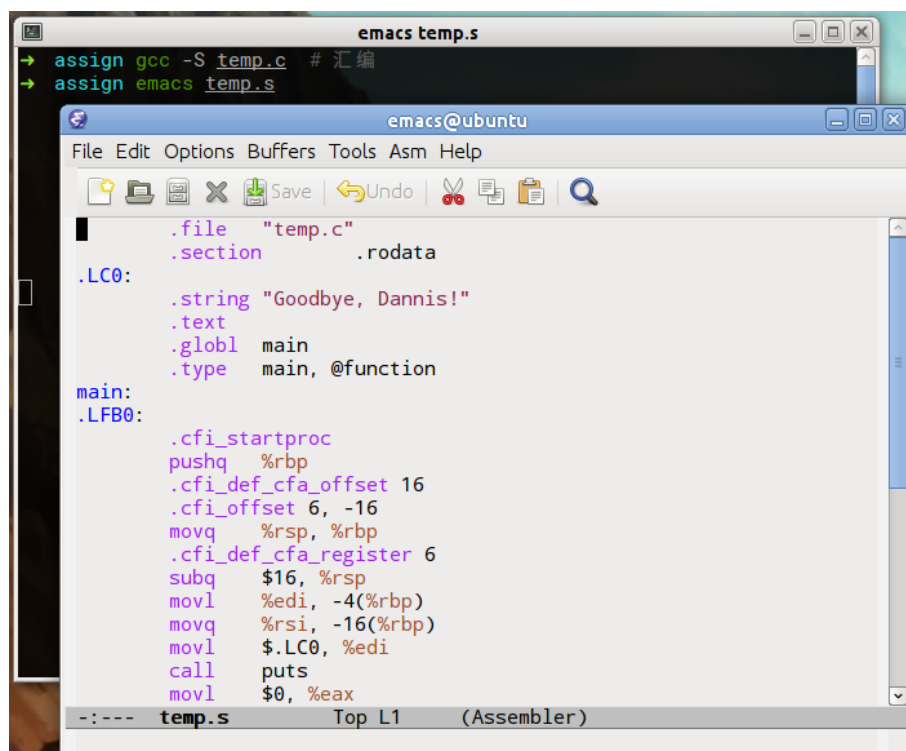
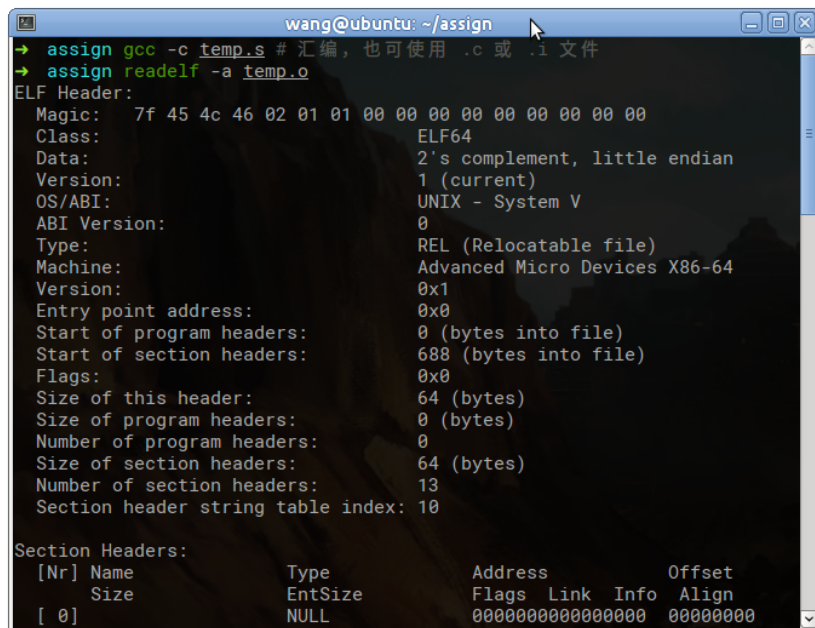
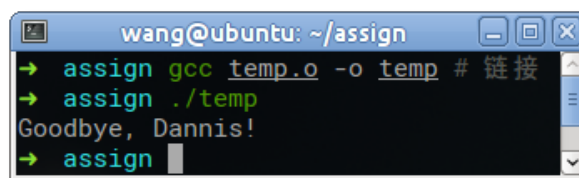


图 3: 编译: 把 C 代码翻译成汇编代码



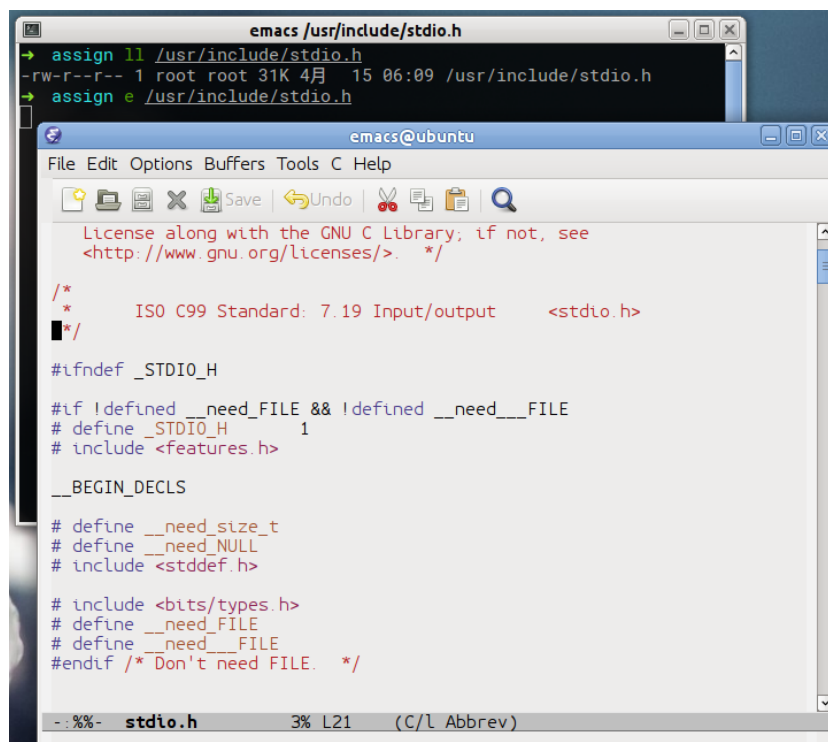
```
wang@ubuntu: ~/assign
→ assign gcc -c temp.s # 汇编, 也可使用 .c 或 .i 文件
→ assign readelf -a temp.o
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:          ELF64
  Data:            2's complement, little endian
  Version:         1 (current)
  OS/ABI:          UNIX - System V
  ABI Version:     0
  Type:            REL (Relocatable file)
  Machine:         Advanced Micro Devices X86-64
  Version:         0x1
  Entry point address: 0x0
  Start of program headers: 0 (bytes into file)
  Start of section headers: 688 (bytes into file)
  Flags:           0x0
  Size of this header: 64 (bytes)
  Size of program headers: 0 (bytes)
  Number of program headers: 0
  Size of section headers: 64 (bytes)
  Number of section headers: 13
  Section header string table index: 10

Section Headers:
[Nr] Name           Type              Address            Offset
     Size            EntSize          Flags Link Info  Align
[ 0]                NULL              0000000000000000  00000000
```

图 4: 汇编: 把汇编代码转换成目标文件 (例如 *ELF* 格式文件)


```
wang@ubuntu: ~/assign
→ assign gcc temp.o -o temp # 链接
→ assign ./temp
Goodbye, Dannis!
→ assign
```

图 5: 链接: 把目标文件链接为可执行文件



```
emacs /usr/include/stdio.h
→ assign ll /usr/include/stdio.h
-rw-r--r-- 1 root root 31K 4月 15 06:09 /usr/include/stdio.h
→ assign e /usr/include/stdio.h

File Edit Options Buffers Tools C Help
License along with the GNU C Library; if not, see
<http://www.gnu.org/licenses/>. */

/*
 * ISO C99 Standard: 7.19 Input/output <stdio.h>
 */

#ifndef _STDIO_H
#define _STDIO_H

#if !defined __need_FILE && !defined __need___FILE
#define _STDIO_H 1
#include <features.h>

__BEGIN_DECLS

#define __need_size_t
#define __need_NULL
#include <stddef.h>

#include <bits/types.h>
#define __need_FILE
#define __need___FILE
#endif /* Don't need FILE. */

-: %%- stdio.h 3% L21 (C/l Abbrev)
```

图 6: *stdio.h* 中的部分代码

## 4 const、static 等概念

### 4.1 赋值操作

```
1 char *cp;
2 const char *ccp;
3 ccp = cp;
4 /* cp = ccp; */
```

这段代码当中，cp 是指向有 const 限定符的 char 的指针，cp 是一个指向没有限定符的 char 指针。因此任何对 cp 允许的操作，对 ccp 也可以合法地操作，故第三条语句可行。反之，若允许第四条语句，则可以通过 cp 修改 ccp 指向的 const char 类型的变量，因此被禁止。

### 4.2 const 使用

```
1 const int limit = 10;
2 const int *limitp = &limit;
3 int i = 27;
4 limitp = &i;
5 // limit = 20;
```

这里 limitp 指向的是 int 常量，而非 limitp 本身是常量。所以 limitp = &i 这个重新绑定指向对象的操作没有问题。

但是最后一行不能通过编译，因为 limit 被声明为 const int，所以它不能被重新被赋值。

### 4.3 赋值失败原因

```
1 void foo(const char **p) {}
2 main(int argc, char **argv) { foo(argv); }
```

赋值操作失败，因为 char \*\* 是指向 char \* 指针，无限定符；而 const char \*\* 是指向 const char \* 的指针，其本身同样无限定符。但是他们指向的类型不一样，因此他们不相容。

### 4.4 整形提升

```
1 int array[] = { 23, 34, 12, 17, 204, 99, 16 };
2 #define TOTAL_ELEMENTS (sizeof(array) / sizeof(array[0]))
3 int main() {
4     int d= -1, x;
5     if (d <= TOTAL_ELEMENTS-2)
6         x = array[d+1];
7     /* ... */
8 }
```

此题有两个点值得注意：

1. 当 signed int 和 unsigned int 做二元运算时，总是先把有符号数转换为无符号数再作运算；
2. 当两个整形长度不一样时，总是把窄的拓宽再运算。

这里，`sizeof(array) / sizeof(array[0])` 这个表达式的类型是 `size_t`，它在实质上是 unsigned long int。注意在 64 位机器上，long 为 64 位，而 int 为 32 位，所以这里会产生整形提升 (integral promotion)。

当以上两点同时出现时，依据 C 标准，应该先把 int 提升为 long int，然后再把 long int 和 unsigned long 比较。因此，在我的机器上，这里的 -1 先是提升到 64 位，依然是 -1，内存中表示为 0xffffffffffffffff，如果将其解释为无符号数，则为 18446744073709551615，远远大于 7-2。因此上述代码片段中表达式求值为假，if 当中的语句不执行。

## 4.5 Fall through

```

1  switch (2) {
2  case 1: printf("case_1_\n");
3  case 2: printf("case_2_\n");
4  case 3: printf("case_3_\n");
5  case 4: printf("case_4_\n");
6  default: printf("default_\n");
7  }
```

case 语句后没有接 break 语句，导致 case 2: 执行后，将后面的语句全部执行，这就是所谓的 fall through。事实上，C 语言将 fall through 设计为 switch 的缺省行为是不合理的，因为这通常不是程序员想得到的行为。

## 4.6 break 的使用

```

1  void network_code() {
2      switch (line) {
3          case THING1: doit1(); break;
4          case THING2: if (x == STUFF) {
5                          do_first_stuff();
6                          if (y == OTHER_STUFF) break;
7                          do_later_stuff();
8                      } /* coder meant to break to here... */
9                      initialize_modes_pointer();
10                     break;
11         default:      processing();
12     } /* ...but actually broke to here! */
13     use_modes_pointer();
14     /* leaving the modes_pointer uninitialized */
15 }
```

这段代码出现问题的原因在于，break 跳出的是 switch 的 case 块，而不是 if。因此提前结束了整个流程。

## 4.7 static 的使用

```

1  void generate_initializer(char *string) {
2      static char separator = '';
3      printf("%c%s_\n", separator, string);
4      separator = ',';
```

5 }

静态变量和全局变量分配在内存的相同区域，他们只初始化一次，后来不再初始化。且不会随着函数返回而消亡。利用这个特性，此函数实现了这样一个特性：依次传一个字符串给此函数，它会打印出逗号分隔的一个字符串列表，形如“aaa, bbb, ccc”。

## 5 漏洞测试

### 5.1 gets()、fgets() 的使用

我的测试代码如下：

```
1 #include <stdio.h>
2 int main() {
3     char s[5];
4     gets(s); puts(s);
5     return 0;
6 }
```

由于 gets() 函数不对输入长度和缓冲区大小作检查，因为当输入过长时，可能会覆盖程序中的其他变量，比如函数返回地址。我做了两个测试：

图 7 是测试 gcc 普通编译结果，当输入超过 8 个字符时，会破坏金丝雀 (canary) 值。此时操作系统检测到金丝雀值的变化，说明栈帧被破坏，因此提前终结程序。图 8 是加上 -fno-stack-protector 参数编译的结果，这个参数用来取消金丝雀的植入。可以看到没有金丝雀的保护时，输入若达到 24 个字符，会导致程序崩溃。

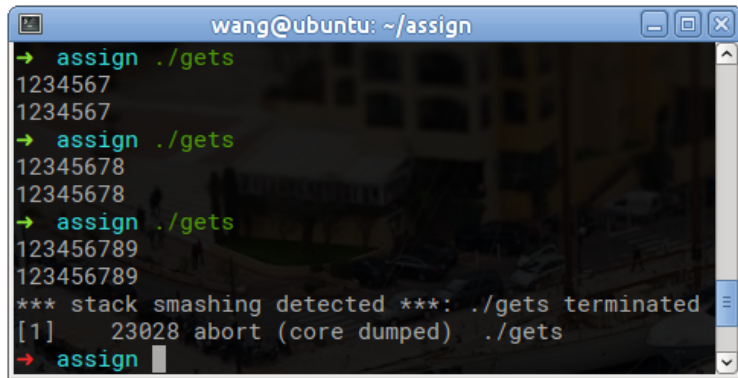


图 7: 默认编译结果测试

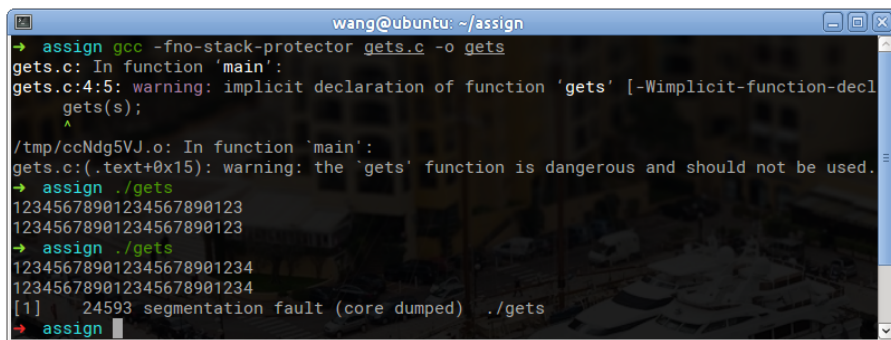


图 8: 加 -fno-stack-protector 参数编译结果测试

## 5.2 局部指针修改方案

```
1  /* Convert the source file timestamp into a localized date string */
2  char *localized_time(char *filename) {
3      struct tm *tm_ptr;
4      struct stat stat_block;
5      char buffer[120];
6      /* get the sourcefile's timestamp in time_t format */
7      stat(filename, &stat_block);
8      /* convert UNIX time_t into a struct tm holding local time */
9      tm_ptr = localtime(&stat_block.st_mtime);
10     /* convert the tm struct into a string in local format */
11     strftime(buffer, sizeof(buffer), "%a_%b_%e_%T_%Y", tm_ptr);
12     return buffer;
13 }
```

这段代码是无法正常工作的。原因在于函数内的自动生命周期变量的内存分配在栈 (stack) 上，在函数返回后，它的原先地址上所存的数据是随机的。一些解决方法包括：

1. 使用一个全局的 `char buffer[120]`。这样此变量可以在调用者和 `localized_time()` 之间共享；
2. 使用静态变量，即 `static char buffer[120]`。静态变量实质上 and 全局变量在内存的同一区域，不会因函数的结束而消亡；
3. 在堆 (heap) 上动态分配 120 个字节。即 `char *buffer = malloc(120)`；
4. 将函数原型声明为 `char *localized_time(char *buffer, size_t len, char *filename)`，然后要求函数调用者传入一块可用的内存作为储存空间，这是我认为最好的办法；
5. 返回字面常量。但实际上这个方法在此处不可行，因为结果很多无法穷举。