$\boxed{\text{Burt Rosenberg}}$

Modules in C

13 OCTOBER, 1994

C language was created in order to write operating systems. Operating systems are large and are written by teams. So, C needed good support for breaking programs into small files which can be individually *compiled*, and then combined during *linking-loading* into a single executable. During linking-loading, procedures and variables defined in one file are linked to uses of those procedures and variables another file. This process is called *resolving external references.*

The particular syntax a programmer uses to share functions and variables across files is a matter of individual taste. Very little structure is required by C. Usage is governed more by custom than by rule. In practice, this has been successful. However, the lack of rule has lead to inconsistencies among compilers. Although some of these inconsistencies are true bugs in the compilers, many stem from vague definitions in the C standards. Care must be exercised to produce code which compiles and runs across many compilers.

Static and External

All functions in a file are defined at the *top level*, and are available for externalizing. Variables which are defined at the top level, outside of any pair of braces, are also available for externalization. External functions and variables exist in a single name space shared by all files involved in the link. To prevent the name of a function from being included in the shared name space, declare it *static*, rather than *extern.* If neither static nor extern is explicitly stated, extern is assumed.

To prevent a variable from being external, declare it inside the body of the function which uses the variable. If a variable is declared static inside the body of a function then the variable's scope is only the function body and the variable lives unchanged from function call to function call. A local variable not declared static is considered *auto.* An auto, or automatic variable, is created anew each time the function is entered, and destroyed when the function exits. A local variable not declared otherwise defaults to auto. In fact, rarely do you find the keyword auto in anyone's code.

Consider the following example:

```
> cat four.c
static int private(void){return 4;}
int four(void){return private();}

> cat five.c
static int private(void){return 5;}
int five(void){return private();}

> cat private.c
#include<stdio.h>
extern int four(void) ;
extern int five(void) ;
int main(){ printf("four=%d, five=%d\n",four(),five()) ; }
```

```
> cc -c four.c
> cc -c five.c
> cc private.c four.o five.o -o private
> private
four=4, five=5
```

In files five.c and four.c, extern is understood for the definition of functions four and five. If static were removed from both these files, an error would occur during linking-loading,

```
> cat four.c
int private(void){ return 4; }
int four(void){ return private(); }

> cat five.c
int private(void){ return 5; }
int five(void){ return private() ; }

> cc -c four.c
> cc -c five.c
> cc private.c four.o five.o -o private
ld:
five.o: private: multiply defined
```

Header Files

A simple way of handling multiple files is to create a single header file for your project. Customarily, one derives the name of the header file from the name of the C file by changing the .c (dot-C) extension to .h (dot-H). Our example would have header file private.h. All external names are declared in this private.h file, and every dot-C file includes this dot-H file. Static functions must be explicitly declared static, and will not appear in the dot-H file. External functions do not wear explicit extern tags when defined, but do wear explicit extern tags when declared in the dot-H file.

Here is our example rewritten using the customary dot-H file.

```
> cat private.h
extern int four(void) ;
extern int five(void) ;

> cat four.c
#include"private.h"
static int private(void){ return 4; }
int four(void){ return private(); }

> cat five.c
#include"private.h"
static int private(void){ return 5; }
```

```
int five(void){ return private(); }

> cat private.c
#include<stdio.h>
#include"private.h"
int main(){ printf("four=%d, five=%d\n",four(),five()) ; }

> cc -c four.c
> cc -c five.c
> cc private.c four.o five.o -o private
> private
four=4, five=5
```

Extern variables should be declared extern in the dot-H file, and defined without extern but with an explicit initializer in exactly one of the dot-C files. Variable defined at the top level which do not appear extern'ed in the dot-H, should be tagged static in the dot-C. These recommendations are found in the excellent C reference book by Harbison and Steel. Here is an example.

```
> cat local.h
extern int global ;
extern int local_in(void) ;
extern int local_out(void) ;

> cat local.c
#include<stdio.h>
#include"local.h"

static int local = 1 ;

int main(){
   global = 3 ;
   local_in() ;
   global = 2 ;
   printf("my local=%d, global=%d, other's local=%d\n",
      local, global, local_out() ) ; }

> cat subrou.c
#include"local.h"

int global = 0 ;
static int local = 0 ;

int local_in(void) { local = global; return local; }
int local_out(void) { return local; }
```

```
> cc -c subrou.c
> cc local.c subrou.o -o local
> local
my local=1, global=2, other's local=3
```

Makefiles

Use of many files leads to complicated compilation procedures. The program make automates compilation. Equally important, make provides a standard format in which to document for other programmers how a project should be compiled: what files are required, what libraries are required, what compiler options should be invoked. For our example, the following simple `makefile` is adequate:

```
OBJS= private.o four.o five.o
private: private.h $(OBJS)
	cc $(OBJS) -o private
$(OBJS): private.h
```

Important! A tab must be the character preceeding the "cc". After editing, just type "make". Make will recompile what needs to be recompiled according to the dependencies stated in this makefile.

Archives

Having many files leads to a further organizational problem. The many files and their arrangement in a directory hierarchy becomes part of the programming project. The programs, including the makefile, should be gathered up for storage into a package that can be unpacked, recreating the original directory layout. The program tar, originally for backing up files onto tape, is used for this. To save space, people often compress files. When a file needs to be transferred, savings in space are also savings in time. Email expects text to use only standard characters. Both tar and compress do not limit themselves to standard characters. The program uuencode converts *binary files*, files using any data values, in *ASCII files*, files using only standard characters. Using uuencode, a binary file can be sent through email. If you ftp a file, it does not have to be uuencoded, but you must set the ftp transfer mode to I.

Here is a shell script that uses tar, compress and uuencode to make a email-able version of a directory. Change directory to the parent of the directory you wish to pack up. Then shell the following file, with parameter the name of the directory to pack up.

```
#! /bin/sh
echo -n "tar $1 ... "
tar cf $1.tar $1
echo -n "compress $1.tar ... "
compress -f $1.tar
echo "uuencode $1.tar.Z ... "
cat $1.tar.Z | uuencode > $1.tar.Z.uu
```