CSE102
Computer Programming with C

2016-2017 Spring Semester

Programming in the Large

Largely adapted from J.R. Hanly, E.B. Koffman, F.E. Sevilgen, and others…

Programming in the Large

"By **large programs** we mean systems consisting of many **small programs** (modules), possibly written by different people."*

* DeRemer, Frank; Kron, Hans (1975). *Programming-in-the large versus programming-in-the-small.*

Programming in the Large

"By **large programs** we mean systems consisting of many **small programs** (modules), possibly written by different people."*

So what are the complications . . .

* DeRemer, Frank; Kron, Hans (1975). *Programming-in-the large versus programming-in-the-small.*

Programming in the Large

Large programs are complicated and are challenging for maintainers to understand.

Programming in the Large

Large programs are complicated and are challenging for maintainers to understand.

The modules are designed with precise interfaces. This requires careful planning and documentation.

Programming in the Large

Large programs are complicated and are challenging for maintainers to understand.

The modules are designed with precise interfaces. This requires careful planning and documentation.

Program changing can become more difficult. If a change occurs across module boundaries, it may involve re-doing work of many people.

Programming in the Large

Large programs are complicated and are challenging for maintainers to understand.

The modules are designed with precise interfaces. This requires careful planning and documentation.

Program changing can become more difficult. If a change occurs across module boundaries, it may involve re-doing work of many people.

Modules are designed with high cohesion and loose coupling, so that they will not need altering in the event of changes.

Programming in the Large

Programming in the Large requires ABSTRACTION skills.

Programming in the Large requires MANAGEMENT skills.

Abstraction

Abstraction

Procedural abstraction.

Abstraction

Procedural abstraction.

Data abstraction.

Abstraction

Procedural abstraction.

Data abstraction.

Information hiding.

Abstraction

Procedural abstraction.

Data abstraction.

Information hiding.

Code reuse.

Abstraction

Procedural abstraction.

Data abstraction.

Information hiding.

Code reuse.

Encapsulation.

Personal Libraries

Header Files.

Implementation Files.

Personal Libraries – Header Files

A block comment summarizing the library's purpose.

#include directives for this library's header file

#define directives naming constant macros.

Type definitions.

Block comments stating the purpose of each library function and declarations of the form `extern prototype`.

Personal Libraries – Implementation Files

A block comment summarizing the library's purpose.

#include directives for this implementation file.

#define directives for this implementation file.

Type definitions used only inside this implementation file.

Function definitions including the usual comments.

Creating a Personal Library

C1 – Create a header file containing the interface information for a program needing the library.

C2 – Create an implementation file containing the code of the library functions and other details of the implementation that are hidden from the user program.

C3 – Compile the implementation file. This step must be repeated any time either the header file or the implementation file is revised.

Using a Personal Library

U1 – Include the library header file in the user program through an `#include` directive.

U2 – After compiling the user program, include both its object file and the object file created in **C3** in the command that activates the linker.

Storage Classes

auto – default for local variables

function parameters and local variables

Storage Classes

auto – default for local variables

function parameters and local variables

```
int function(char *buffer, int size)
{
        int result;
        -    -    -    -    -    -
        -    -    -    -    -    -
        return result;
}
```

Storage Classes

extern – visible to linker and other files

extern prototype and extern variable

Storage Classes

extern – visible to linker and other files

extern prototype and extern variable

```
/* main.c */
extern void print_count(int count);
int main()
{
        int count = 0;
        -   -   -   -   -   -
        print_count(count);
        -   -   -   -   -   -
}
```

Storage Classes

extern – visible to linker and other files

extern prototype and extern variable

```
/* count.c */
void print_count(int count)
{
        printf("count is %d", count);
}
```

Storage Classes

global variables – variable at the top level

Storage Classes

global variables – variable at the top level

```
/* main.c */
const int size = 65535;
Int main()
{
        -   -   -   -   -   -
        print_size();
        -   -   -   -   -   -
}
```

Storage Classes

global variables – variable at the top level

```
/* size.c */
extern const int size;
void print_size()
{
        size = 1024;
        printf("size is %d", size);
}
```

Output:

Storage Classes

global variables – variable at the top level

```
/* size.c */
extern const int size;
void print_size()
{
        size = 1024;
        printf("size is %d", size);
}
```

Output: error: assignment of read-only variable '**size**'

Storage Classes

global variables – variable at the top level

```
/* size.c */
extern const int size;
void print_size()
{

        printf(“size is %d”, size);
}
```

Output:

Storage Classes

global variables – variable at the top level

```
/* size.c */
extern const int size;
void print_size()
{

        printf("size is %d", size);
}
```

Output: size is 65535

Storage Classes

const – the value initialized cannot be changed

Storage Classes

const – the value initialized cannot be changed

```
/* main.c */
const int size = 65535;
Int main()
{
        -    -    -    -    -    -
        print_size();
        -    -    -    -    -    -
}
```

Storage Classes

const – the value initialized cannot be changed

```
/* size.c */
extern const int size;
void print_size()
{
        size = 1024;
        printf("size is %d", size);
}
```

Output:

Storage Classes

const – the value initialized cannot be changed

```
/* size.c */
extern const int size;
void print_size()
{
        size = 1024;
        printf(“size is %d”, size);
}
```

Output: error: assignment of read-only variable ‘**size**’

Storage Classes

const – the value initialized cannot be changed

```
/* size.c */
extern const int size;
void print_size()
{

        printf(“size is %d”, size);
}
```

Output:

Storage Classes

const – the value initialized cannot be changed

```
/* size.c */
extern const int size;
void print_size()
{

        printf(“size is %d”, size);
}
Output: size is 65535
```

Storage Classes

typedef – a type definition with no allocation of storage space

Storage Classes

typedef – a type definition with no allocation of storage space

```
typedef struct
{
        char current[3];
        int volts;
        struct node *next;
} node;
```

Storage Classes

typedef – a type definition with no allocation of storage space

```
void free_all()
{
        node *n = head;
        while (n != 0)
        {
                head = (node *)n->next;
                free (n);
                n = head;
        }
}
```

More Storage Classes

static

1. Once allocated and initialized one time, it remains allocated until the program terminates.
2. Retain data from one call to a function to the next.

Example **static**

```
#include <stdio.h>
void func( void ) {
   int count = 5;
   int i = 5;
   i++; count--;
   printf("i is %2d and count is %2d\n", i, count);
}
int main() {
   int i;
   for (i = 0 ; i < 5; i++)
      func();
   return 0;
}
Output:
```

Example **static**

```
#include <stdio.h>
void func( void ) {
   int count = 5;
   int i = 5;
   i++; count--;
   printf("i is %2d and count is %2d\n", i, count);
}
int main() {
   int i;
   for (i = 0 ; i < 5; i++)
      func();
   return 0;
}
```

Output:

```
i is  6 and count is  4
i is  6 and count is  4
i is  6 and count is  4
i is  6 and count is  4
i is  6 and count is  4
```

Example **static**

```
#include <stdio.h>
void func( void ) {
   static int count = 5;   /* local static variable */
   static int i = 5;       /* local static variable */
   i++; count--;
   printf("i is %2d and count is %2d\n", i, count);
}
int main() {
   int i;
   for (i = 0 ; i < 5; i++)
      func();
   return 0;
}
Output:
```

Example **static**

```c
#include <stdio.h>
void func( void ) {
   static int count = 5;   /* local static variable */
   static int i = 5;       /* local static variable */
   i++; count--;
   printf("i is %2d and count is %2d\n", i, count);
}
int main() {
   int i;
   for (i = 0 ; i < 5; i++)
      func();
   return 0;
}
```

Output:

```
i is  6 and count is  4
i is  7 and count is  3
i is  8 and count is  2
i is  9 and count is  1
i is 10 and count is  0
```

More Storage Classes

static

1. Once allocated and initialized one time, it remains allocated until the program terminates.
2. Retain data from one call to a function to the next.

register

1. Notify the compiler that this variable will be used more often than others.
2. Compiler tries to allocate a physical register on the machine for its use.
3. Cannot apply the unary operator &, as it doesn’t have a memory location.

Example **register**

```
int matrix[65535];
register char index;
int step = 10;

for (index = 0; index < 65535; index++)
     matrix[index] = step * index;
```

Example **register**

```
int matrix[65535];
register char index;
int step = 10;

for (index = 0; index < 65535; index++)
     matrix[index] = step * index;
```

Example **register**

```
int matrix[65535];
register unsigned char index;
int step = 10;

for (index = 0; index < 65535; index++)
     matrix[index] = step * index;
```

Example **register**

```
int matrix[65535];
register unsigned char index;
int step = 10;

for (index = 0; index < 256; index++)
     matrix[index] = step * index;
```

Example **register**

```
int matrix[65535];
register unsigned char index;
int step = 10;

for (index = 0; index < 255; index++)
     matrix[index] = step * index;
```

Example **register**

```
int matrix[65535];
register unsigned char index;
int step = 10;

for (index = 0; index < 65535; index++)
     matrix[index] = step * index;
```

Example **register**

```
int matrix[65535];
register short index;
int step = 10;

for (index = 0; index < 65535; index++)
     matrix[index] = step * index;
```

Example **register**

```
int matrix[65535];
register int index;
int step = 10;

for (index = 0; index < 65535; index++)
     matrix[index] = step * index;
```

What is Preprocessing

C Preprocessor is just a text substitution tool.

It instructs the compiler to do the required preprocessing before the actual compilation.

Preprocessing command always begin with a symbol #.

Some Examples

#include

```
#include <stdio.h>
#include "myheader.h"
```

#define

```
#define BUFFER_SIZE 1024
```

#undef

```
#undef BUFFER_SIZE
#define BUFFER_SIZE 65535
```

Some Examples

```
#ifdef
#endif
/* conditional compilation */
#ifdef __DEBUG__
        printf("Value of parameter a = %d\n", a);
#endif
#ifdef TRACE
        printf("Entering the program %s\n", prog_name);
#endif
-   -   -   -   -
#ifdef TRACE
        printf("Leaving the program %s\n", prog_name);
#endif
```

Some Examples

Once-Only Headers

#ifndef

```
#ifndef __UTIL_H__   /* file util.h */
#define __UTIL_H__
        #include <stdio.h>
        void printData(char *array, int size);
        -   -   -   -   -
        -   -   -   -   -
#endif  /* __UTIL_H__ */
```

Some Examples

#if
#elif
#else

```
#if SYSTEM_1
#include “system_1.h”
#elif SYSTEM_2
#include “system_2.h”
#else
#include “system_3.h”
#endif
```

Some Examples

#pragma

Provide additional information to the compiler.

```
typedef struct {
  char a;
  int b;
} PS;
printf("sizeof(PS) = %d\n", sizeof(PS));
sizeof(PS) = _
```

Some Examples

#pragma

Provide additional information to the compiler.

```
typedef struct {
  char a;
  int b;
} PS;
printf("sizeof(PS) = %d\n", sizeof(PS));
sizeof(PS) = 8
```

Some Examples

#pragma

Provide additional information to the compiler.

```
#pragma pack(push, 1)
typedef struct {
  char a;
  int b;
} PS;
printf("sizeof(PS) = %d\n", sizeof(PS));
sizeof(PS) = _
```

Some Examples

#pragma

Provide additional information to the compiler.

```
#pragma pack(push, 1)
typedef struct {
  char a;
  int b;
} PS;
printf("sizeof(PS) = %d\n", sizeof(PS));
sizeof(PS) = 5
```

Macro Definition

object-like macro

#define <macro> <tokens>

```
#define   PI   3.14159
#define   MSG   “Testing macros”
```

function-like macro

#define <macro>(<parameters>) <tokens>

```
#define   square(n)   ((n) * (n))
#define   radtodeg(n)   ((n) * 57.29578)
```

Predefined Macros

__DATE__	The compilation date of the current source file in "MMM DD YYYY" format.
__TIME__	The current time of translation of the preprocessed translation unit in "HH:MM:SS" format.
__FILE__	The name of the current source file
__LINE__	This contains the current line number in the current source file
__STDC__	Defined as 1 when the compiler complies with the ANSI C standard.

Preprocessor Operators

Continuation (\) & Stringification (#) operators

```
#define print_error(error, num) \
        fprintf(stderr, "Error number " #num ": " #error "\n")
```

use:

```
print_error("Cannot open the file", 101);
```

output:

```
Error number 101: "Cannot open the file“
```

Preprocessor Operators

Defined operator

```
#if defined (STACK)
        stack();
#elif defined (QUEUE)
        queue();
#else
        error();
#endif
```

Directories and Files

```
salam@SALAM-PC: ls -l
drwx------+ 1 salam None    0 Dec  3 14:11 bin/
-rw-------+ 1 salam None 1.4K Dec  3 14:11 Makefile
drwx------+ 1 salam None    0 Dec  3 14:11 source/
```

Directories and Files

```
salam@SALAM-PC: ls -l
drwx------+ 1 salam None    0 Dec  3 14:11 bin/
-rw-------+ 1 salam None 1.4K Dec  3 14:11 Makefile
drwx------+ 1 salam None    0 Dec  3 14:11 source/

salam@SALAM-PC: ls -l source
drwx------+ 1 salam None    0 Dec  2 16:51 include/
drwx------+ 1 salam None    0 Dec  3 14:11 ll/
-rw-------+ 1 salam None  769 Dec  2 18:50 main.c
```

Directories and Files

```
salam@SALAM-PC: ls -l
drwx------+ 1 salam None    0 Dec  3 14:11 bin/
-rw-------+ 1 salam None 1.4K Dec  3 14:11 Makefile
drwx------+ 1 salam None    0 Dec  3 14:11 source/

salam@SALAM-PC: ls -l source
drwx------+ 1 salam None    0 Dec  2 16:51 include/
drwx------+ 1 salam None    0 Dec  3 14:11 ll/
-rw-------+ 1 salam None  769 Dec  2 18:50 main.c

salam@SALAM-PC: ls -l source/ll
-rw-------+ 1 salam None 1.3K Dec  2 18:49 ll.c
-rw-------+ 1 salam None  307 Dec  2 18:47 ll.h
```

Directories and Files - Compiling and Linking

```
salam@SALAM-PC: ls -l
drwx------+ 1 salam None    0 Dec  3 14:11 bin/
-rw-------+ 1 salam None 1.4K Dec  3 14:11 Makefile
drwx------+ 1 salam None    0 Dec  3 14:11 source/

salam@SALAM-PC: ls -l source
drwx------+ 1 salam None    0 Dec  2 16:51 include/
drwx------+ 1 salam None    0 Dec  3 14:11 ll/
-rw-------+ 1 salam None  769 Dec  2 18:50 main.c

salam@SALAM-PC: ls -l source/ll
-rw-------+ 1 salam None 1.3K Dec  2 18:49 ll.c
-rw-------+ 1 salam None  307 Dec  2 18:47 ll.h

gcc -Ofast -ansi  -c source/ll/ll.c -o source/ll/ll.o
```

Directories and Files - Compiling and Linking

```
salam@SALAM-PC: ls -l
drwx------+ 1 salam None    0 Dec  3 14:11 bin/
-rw-------+ 1 salam None 1.4K Dec  3 14:11 Makefile
drwx------+ 1 salam None    0 Dec  3 14:11 source/

salam@SALAM-PC: ls -l source
drwx------+ 1 salam None    0 Dec  2 16:51 include/
drwx------+ 1 salam None    0 Dec  3 14:11 ll/
-rw-------+ 1 salam None  769 Dec  2 18:50 main.c

salam@SALAM-PC: ls -l source/ll
-rw-------+ 1 salam None 1.3K Dec  2 18:49 ll.c
-rw-------+ 1 salam None  307 Dec  2 18:47 ll.h

gcc -Ofast -ansi  -c source/ll/ll.c -o source/ll/ll.o

gcc -Ofast -ansi  -c source/main.c -o source/main.o
```

Directories and Files - Compiling and Linking

```
salam@SALAM-PC: ls -l
drwx------+ 1 salam None    0 Dec  3 14:11 bin/
-rw-------+ 1 salam None 1.4K Dec  3 14:11 Makefile
drwx------+ 1 salam None    0 Dec  3 14:11 source/

salam@SALAM-PC: ls -l source
drwx------+ 1 salam None    0 Dec  2 16:51 include/
drwx------+ 1 salam None    0 Dec  3 14:11 ll/
-rw-------+ 1 salam None  769 Dec  2 18:50 main.c

salam@SALAM-PC: ls -l source/ll
-rw-------+ 1 salam None 1.3K Dec  2 18:49 ll.c
-rw-------+ 1 salam None  307 Dec  2 18:47 ll.h

gcc -Ofast -ansi  -c source/ll/ll.c -o source/ll/ll.o

gcc -Ofast -ansi  -c source/main.c -o source/main.o

gcc -Ofast -ansi  source/ll/ll.o source/main.o  -o bin/ll.exe
```

Makefile

```
ROOT   := source

BIN_DIR         := bin
INCLUDES_DIR    := $(ROOT)/include
LL_DIR          := $(ROOT)/ll
MAIN_DIR        := $(ROOT)

CC        := gcc
LD        := gcc
CFLAGS    := -Ofast -ansi
LDFLAGS   := -Ofast -ansi

SRCS       =      $(LL_DIR)/ll.c \
                  $(MAIN_DIR)/main.c

all: $(SRCS) ll.exe

OBJS      := $(SRCS:.c=.o)
```

Makefile

```
ll.exe: $(OBJS)
    @printf   "   LD                    $(BIN_DIR)/$@\n"
    $(LD) $(LDFLAGS) $(OBJS) $(LIBS) -o $(BIN_DIR)/$@

.c.o:
    @printf   "   CC                    $@\n"
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    @printf   "   RM \n $(OBJS)\n"
    $(RM) $(OBJS)

clean_all:
    @printf   "   RM \n $(OBJS) $(BIN_DIR)/ll.exe\n"
    $(RM) $(OBJS) $(BIN_DIR)/ll.exe
```

Directories and Files – Building ll.exe

```
salam@SALAM-PC: ls -l
drwx------+ 1 salam None    0 Dec  3 14:11 bin/
-rw-------+ 1 salam None 1.4K Dec  3 14:11 Makefile
drwx------+ 1 salam None    0 Dec  3 14:11 source/

salam@SALAM-PC: ls -l source
drwx------+ 1 salam None    0 Dec  2 16:51 include/
drwx------+ 1 salam None    0 Dec  3 14:11 ll/
-rw-------+ 1 salam None  769 Dec  2 18:50 main.c

salam@SALAM-PC: ls -l source/ll
-rw-------+ 1 salam None 1.3K Dec  2 18:49 ll.c
-rw-------+ 1 salam None  307 Dec  2 18:47 ll.h
```

```
salam@SALAM-PC: make all
   CC             source/ll/ll.o
gcc -Ofast -ansi -c source/ll/ll.c -o source/ll/ll.o
   CC             source/main.o
gcc -Ofast -ansi -c source/main.c -o source/main.o
   LD             bin/ll.exe
gcc -Ofast -ansi source/ll/ll.o source/main.o -o bin/ll.exe
```