18600 Boot camp: C- Review

9/6/2016

Today - All About C!

- Longer lecture today
- Ask questions at any time!

Some basic facts about C

- C was invented to write an operating system called UNIX
- The UNIX OS was completely written in C
- Today C is the most widely used and popular System Programming Language.
- Example use cases of C: Operating Systems, Compilers, Interpreters, Databases, Assemblers, Text editors, Device Drivers
- C is a compiled language. The most frequently used and free available compiler is the GNU C/C++ compiler. Eg: gcc foo.c

Basic C Program Structure

Hello World.c

```
#include <stdio.h>

int main(void) {
   /* my first program in C */
   int a = 18600;
   printf("Hello! Welcome to %d \n", a);

   return 0;
}
```

Notice the following components:

- Preprocessor commands
- Functions
- Variables
- Comments
- Statements
- Parameters, return values

Data Types in C

- Basic Types
  - Integer: char, int, long, double, float (both signed and unsigned)
- Void Types
  - Indicate no value: Eg: void main(void) {....}
- User Defined Data Types / Data Structures
  - Arrays, Structures
- Special Data Types
  - Enum, Unions

Basic Data Types

Type	Storage size (x86-64 compiler specific)	Range of values	Precision
char	1 byte	0 - 255 (unsigned), -128-127 (signed)	--NA--
int	4 bytes	0 to 4,294,967,295 (unsigned) -2,147,483,648 to 2,147,483,647 (signed)	--NA--
long long	8 bytes	0 to 18,446,744,073,709,551,615 (unsigned) −9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (signed)	--NA--
float	4 bytes	1.2E-38 to 3.4E+38	6 decimals
double	8 bytes	2.3E-308 to 1.7E+308	15 decimals
long double	10 bytes	3.4E-4932 to 1.1E+4932	19 decimals

Aggregate Data Types : Arrays/Strings

- Arrays: Fixed sized sequential collection of data of the same type
  - Array declaration: type arrayName[size]. Eg: int array[10], char array[10]
  - Array definition: int array[5] = {0,1,2,3,4};
  - Accessing an array element: int secElem = array[1]
  - Multi-dimensional array: 2-dimensional arrays are most common
    - 2-dimensional array is a list of 1-dimensional arrays
    - Eg: int array[4][4], char array[3][2]
- Strings: Null terminated ('\0') terminated character array
  - Null-character tells us where the string ends
  - All standard C library functions on strings assume null-termination.

Aggregate Data Types: Struct

- Collection of values placed under one name in a single block of memory
  - Can put structs, arrays in other structs
  - Can have arrays of structures too
- Given a struct instance, access the fields using the '.' operator
- Given a struct pointer, access the fields using the '->' operator

```
struct foo_s {
    int a;
    char b;
};
```

```
struct bar_s {
    char ar[10];
    struct foo_s baz;
}
```

```
struct bar_s biz; // bar_s instance
biz.ar[0] = 'a';
biz.baz.a = 1;
struct bar_s* boz = &biz; // bar_s ptr
boz->baz.b = 'b';
```

Pointers in C

- A pointer is a variable which stores the address of a value in memory
  Syntax: type *ptr
    - Eg: int *ptr, char *ptr, void *ptr
- Get the address of a value in memory with the '&' operator
    - Eg: int a = 10; ptr = &a;
- Access the value by dereferencing using the * operator; can be used to read value or write value to given address
    - Eg: int b = *ptr; *ptr = 3;
    - Dereferencing NULL causes a runtime error
        - Eg: int *ptr = NULL; *p = 0; // Runtime error !!!!

Pointer Arithmetic

- Can add/subtract from an address to get a new address
  - Only perform when absolutely necessary (i.e., malloc)
  - Result depends on the pointer type
- Pointer to type 'a' references a block of sizeof(a) bytes. Any arithmetic operations therefore moves in steps of these block sizes
- Examples:
  - A+i, where A is a pointer = 0x100, i is an int (x86-64)
    - int* A: A+i = 0x100 + sizeof(int) * i = 0x100 + 4 * i
    - char* A: A+i = 0x100 + sizeof(char) * i = 0x100 + i
    - int** A: A + i = 0x100 + sizeof(int*) * i = 0x100 + 8 * i
- Rule of thumb: cast pointer explicitly to avoid confusion. More on this in later slides
  - Prefer (char*)(A) + i vs A + i, even if char* A

Pointers: Let’s try some examples…

```
#include <stdio.h>

int main ()
{

   int  var;
   int  *ptr;
   int  **pptr; // Pointer to a pointer
   // Array of pointers
   char *names[] = {"Tom", "Dick", "Harry"};

   var = 3000;

   /* take the address of var */
   ptr = &var;

   /* take the address of ptr using address of operator & */
   pptr = &ptr;

   printf("Value of var = %d\n", var );
   printf("Value available at *ptr = %d\n", *ptr );
   printf("Value available at pointer after increment = %d\n", ++*ptr);
   printf("Value available at **pptr = %d\n", **pptr);
   printf("First student is %s\n", names[0]);

   return 0;
}
```

Functions in C

- Call-by-value: Changes made to arguments passed to a function aren’t reflected in the calling function
- Call-by-reference: Changes made to arguments passed to a function are reflected in the calling function

```
#include <stdio.h>

/* function declaration */
void swap(int x, int y);

int main () {

   /* local variable definition */
   int a = 100;
   int b = 200;

   printf("Before swap, value of a : %d\n", a );
   printf("Before swap, value of b : %d\n", b );

   /* calling a function to swap the values */
   swap_by_val(a, b);

   printf("After swap, value of a : %d\n", a ); // 100
   printf("After swap, value of b : %d\n", b ); // 200

   swap_by_ref(&a, &b);

   printf("After swap, value of a : %d\n", a ); // 200
   printf("After swap, value of b : %d\n", b ); // 100

   return 0;
}
```

```
/* function definition to swap the values */
void swap_by_val(int x, int y) {

   int temp;

   temp = x; /* save the value of x */
   x = y;    /* put y into x */
   y = temp; /* put temp into y */

   return;
}
```

```
/* function definition to swap the values */
void swap_by_ref(int *x, int *y) {

   int temp;
   temp = *x;    /* save the value at address x */
   *x = *y;      /* put y into x */
   *y = temp;    /* put temp into y */

   return;
}
```

Function calls in C

Ensure that the called function is defined (see func_call1.c) or at least declared (see func_call2.c) before the calling function. Else, the compiler will complain about an undefined reference to that function.

```
#include <stdio.h>
// Definition of a function
int sum(int a, int b)
{
    return a+b;
}
void main() {

   int  a = 3, b=4;

   printf(“%d”, sum(a, b));
}
```

func_call1.c

```
#include <stdio.h>

// Declaration of a function
int sum(a, b);

main() {

   int  a = 3, b=4;

   printf(“%d”, sum(a, b));
}

// Definition of a function
int sum(int a, int b)
{
    return a+b;
}
```

func_call2.c

Typedef in C

(Not strongly recommended in this course)

- The C programming language provides a keyword called **typedef**, which you can use to give a type, a new name.
- Typedefs are used to give a more meaningful/readable/shorter name to the data type used.
- Simple Example: typedef unsigned char BYTE; BYTE b1, b2;

```
struct list_node {
   int x;
};

/* You can typedef basic data types */
typedef int pixel;
typedef unsigned char BYTE;

/* You can typedef structures */
typedef struct list_node node;

/* You can typedef function prototypes */
typedef int (*cmp)(int e1, int e2);


pixel x;                    //  int type
BYTE b1;                    // char type
node foo;                   // struct list_node type
cmp int_cmp;                // int (*cmp)(int e1, int e2) type
```

Variable Scope and Qualifiers

- Every variable is associated with a scope and storage duration
- Scope determines where a variable can be accessed and storage duration determines when a variable is created and destroyed
  - Global Variables are defined outside functions. Use ‘extern’ to use global variables in other files
    - Scope: Across all files, Storage: Start and end of a program
  - Local variables are defined within functions
    - Scope: Within a function, Storage: Entry and exit of a function
- Variable qualifiers
  - Const Variables: For variables that won’t change
  - Static Variables:
    - Globals: usable/viewable only from within the current file: More on this next slide
    - Locals: For locals, keeps value between invocations
  - Volatile Variables: Variable values subject to change

Illustrating Variable Scope

```
#include <stdio.h>

int count ;
static int local_ref;
extern void write_extern();

// there can be only one main function among the compiled
// programs
main() {
   count = 5;
   local_ref = count;
   write_extern();
   local_fn();  // Compile time error
}
```

main.c

```
#include <stdio.h>

extern int count;

void write_extern(void) {
   printf("count is %d\n", count);
   printf("local_ref  is %d\n", local_ref); // Compile time error
 }

static void local_fn(void) {
   printf("Scope is restricted to this file\n");
}
```

support.c

gcc main.c support.c

Type Casting

- Type casting is a way to convert a variable from one data type to another data type.
- Typically used when dealing with operations between different data types
- When values of different data types are operated on each other, all variables are converted to a type that is highest among them
- Integer Type Casting:
  - signed <-> unsigned: change interpretation of most significant bit
  - smaller signed -> larger signed: sign-extend (duplicate the sign bit)
  - smaller unsigned -> larger unsigned: zero-extend (duplicate 0)
- Cautions:
  - C implicitly typecasts, which can lead to errors. It is a good practice to explicitly typecast.
  - never cast to a smaller type; will truncate (lose) data
  - never cast a pointer to a larger type and dereference it, this accesses memory with undefined contents

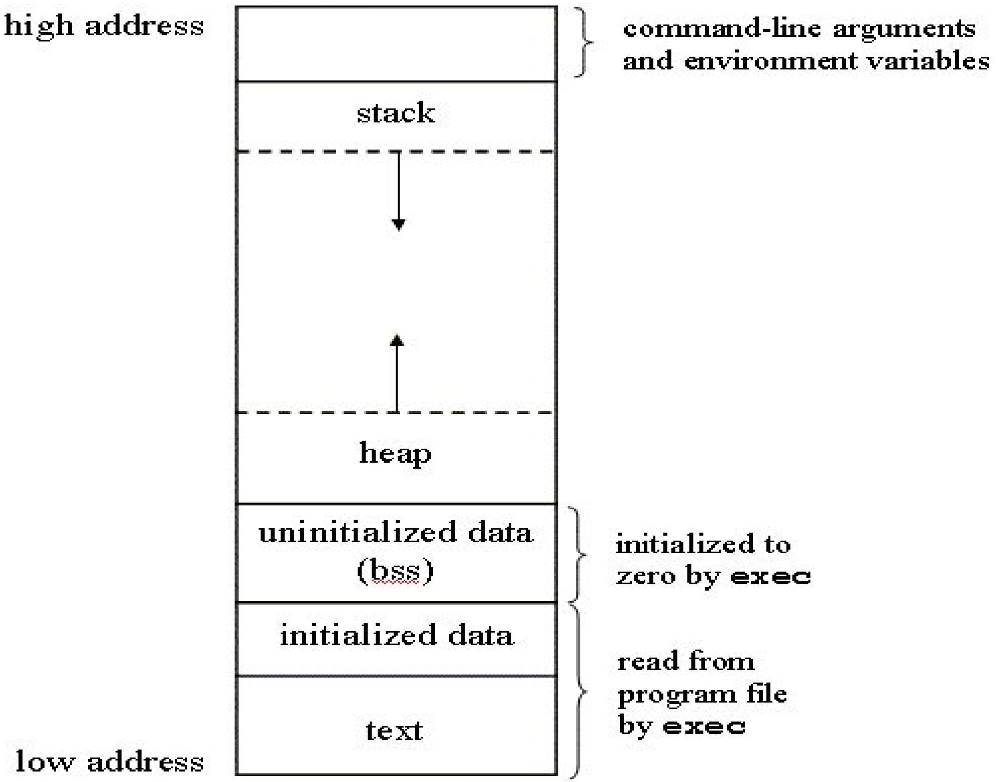
Void pointers

- void* type is C's provision for generic types
  - Raw pointer to some memory location (unknown type)
  - Can't dereference a void* (what is type void?)
  - Must cast void* to another type in order to dereference it
- Used by functions which work only with the pointer and not the contents of the pointer. Eg: push() and pop() routines below
- Can cast back and forth between void* and other pointer types

```
// stack implementation:

typedef void* elem;

stack stack_new();
void push(stack S, elem e);
elem pop(stack S);
```

```
// stack usage:

int x = 42; int y = 54;
stack S = stack_new():
push(S, &x);
push(S, &y);
int a = *(int*)pop(S);
int b = *(int*)pop(S);
```

C Program Memory Layout



Stack vs Heap vs Data

- Local variables and function arguments are placed on the stack
  - deallocated after the variable leaves scope
  - do not return a pointer to a stack-allocated variable!
  - do not reference the address of a variable outside its scope!
- Memory blocks allocated by calls to malloc/calloc are placed on the heap
- Globals, constants are placed in data section
- Example:
  - // a is a pointer on the stack to a memory block on the heap
  - int* a = malloc(sizeof(int));

Macros

- Fragment of code given a name; replace occurrence of name with contents of macro
  - No function call overhead, type neutral
- Uses:
  - defining constants (INT_MAX, ARRAY_SIZE)
  - defining simple operations (MAX(a, b))
  - 122-style contracts (REQUIRES, ENSURES)
- Warnings:
  - Use parentheses around arguments/expressions, to avoid problems after substitution
  - Do not pass expressions with side effects as arguments to macros

```
#define INT_MAX 0x7FFFFFFF
#define MAX(A, B) ((A) > (B) ? (A) : (B))
#define REQUIRES(COND) assert(COND)
#define WORD_SIZE 4
#define NEXT_WORD(a) ((char*)(a) + WORD_SIZE)
```

Header Files

- Includes C declarations and macro definitions to be shared across multiple files
- Only include function prototypes/macros; no implementation code!
- Usage: #include <header.h>
  - #include <lib> for standard libraries (eg #include <string.h>)
  - #include “file” for your source files (eg #include “header.h”)
- Never include .c files (bad practice)

```
// list.h
struct list_node {
   int data;
   struct list_node* next;
};
typedef struct list_node* node;

node new_list();
void add_node(int e, node l);
```

```
// list.c
#include “list.h”

node new_list() {
   // implementation
}

void add_node(int e, node l) {
   // implementation
}
```

```
// stacks.h
#include “list.h”
struct stack_head {
   node top;
   node bottom;
};
typedef struct stack_head* stack

stack new_stack();
void push(int e, stack S);
```

Header Guards

- Double-inclusion problem: include same header file twice

```
//grandfather.h
```

```
//father.h
#include “grandfather.h”
```

```
//child.h
#include “father.h”
#include “grandfather.h”
```

Error: child.h includes grandfather.h twice

- Solution: header guard ensures single inclusion

```
//grandfather.h
#ifndef GRANDFATHER_H
#define GRANDFATHER_H


#endif
```

```
//father.h
#ifndef FATHER_H
#define FATHER_H


#endif
```

```
//child.h
#include “father.h”
#include “grandfather.h”
```

Okay: child.h only includes grandfather.h once

Preprocessing in C

- A C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation
- Handling of header files and macros is done during the preprocessing stage

```
#define MAX_ARRAY_LENGTH 20          // For standard values

#include <stdio.h>                   // include header files

#ifndef __HEADER__                   // Used in header files to avoid duplication
#define __HEADER__
#endif

__FILE__, __LINE__, __func__         // Predefined macros

#define  message_for(a, b)  \        // When continuing macro definitions on multiple lines
   printf(#a " and " #b ": We love you!\n")

#define square(x) ((x) * (x))        // Parameterized macros: Simulate functions using macros
```

C - Command Line Arguments

- It is possible to pass some values from the command line to your C programs when they are executed.
- These values are called command line arguments, they allow you to control your program from outside instead of hard coding those values inside the code.

```
#include <stdio.h>

int main( int argc, char *argv[] )  {
   // argc: Number of command line arguments
   // argv: Array of pointers to each argument
   if( argc == 2 ) {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if( argc > 2 ) {
      printf("Too many arguments supplied.\n");
   }
   else {
      printf("One argument expected.\n");
   }
}
```

C Memory Management

- Memory can be **statically** allocated or **dynamically** allocated
- Memory is said to be statically allocated when it is reserved at the time of compilation
- Memory is said to be dynamically allocated when it is reserved at the time of program execution. Eg: Using c library functions such as malloc(), calloc(), realloc()
- Statically allocated memory is freed automatically at the end of a function call or program execution depending on the scope of the variable
- Dynamically allocated memory has to be freed explicitly using the free() system call
- IMPORTANT
  - Number mallocs = Number frees
  - Never free a malloced block twice
  - Free only what you malloc and malloc only what you free

Why We Need Malloc

- Something that students new to the language often get confused about
- i.e. What is wrong with the following program?

```
/* Very bad program! Will compile and run though! */
int main(int argc, char *argv[]) {
        int N;
        if (argc >= 2) {
                N = atoi(argv[1]);
                char mystr[N];          char *mystr = malloc(N*sizeof(char));
                myfunc(mystr);
        }
        return 0;
}
```

- What is the size of mystr? **Ans: Undefined**
- **Malloc allows us to obtain memory *during* program execution**

System calls and error conditions

- A System Call is a mechanism in which the **user application requests the service of the kernel** **(why do we need to do this?)**
- May be called directly or indirectly through c library functions (e.g. fopen() calls open())
- System calls **may not always succeed.** It is therefore important to check the status of the return values from these calls before proceeding
- List of commonly used system calls include: **open(), read()/write(), pipe(), fork(), exec(), time(), waitpid()**
- A system call **sets the global variable errno** with the error code, which can be printed using strerror(). The various error codes are defined in error.h

```c
// Program showing how to read error codes
#include <stdio.h>
#include <errno.h>
#include <string.h>

extern int errno ;

int main () {

   FILE * pf;
   int errnum;
   pf = fopen ("unexist.txt", "rb");

   if (pf == NULL) {

      errnum = errno;
      fprintf(stderr, "Value of errno: %d\n", errno);
      perror("Error printed by perror");
      fprintf(stderr, "Error opening file: %s\n", strerror( errnum ));
   }
   else {

      fclose (pf);
   }

   return 0;
}
```

```c
// Program demonstrating how to return exit status
#include <stdio.h>
#include <stdlib.h>

main() {

   int dividend = 20;
   int divisor = 5;
   int quotient;

   if( divisor == 0) {
      fprintf(stderr, "Division by zero! 
Exiting...\n");
      exit(EXIT_FAILURE);
   }

   quotient = dividend / divisor;
   fprintf(stderr, "Value of quotient : %d\n", 
quotient );

   exit(EXIT_SUCCESS);
}
```

C Standard Library

- Many basic housekeeping functions are available to a C program in form of standard library functions.
- To call these, a program must #include the appropriate .h file.
- You can use ‘man’ commands on these functions to learn about their usage.
- Most commonly used header files:
    - stdio.h:
        - File I/O: fopen(), fclose(), fscanf(), fprintf()
        - Command line argument parsing: getopt()
    - string.h string operations
        - char * strcpy(char *dst, char *src)
        - char * strcat(char *dst, char *src)
        - size_t strlen(char *str)
        - int strcmp(char *str1, char *str2)
    - stdlib.h
        - Dynamic memory allocation functions: malloc(), calloc(), free()
        - exit(int status): terminate program and return exit status to the parent

Compilation

GCC, Make Files

GCC

- Used to compile C/C++ projects
- List the files that will be compiled to form an executable
- Specify options via flags
- Important Flags:
  - -g: produce debug information (important; used by GDB/valgrind)
  - -Werror: treat all warnings as errors (this is our default)
  - -Wall/-Wextra: enable all construction warnings
  - -pedantic: indicate all mandatory diagnostics listed in C-standard
  - -O0/-O1/-O2: optimization levels
  - -o <filename>: name output binary file ‘filename’
- Example:
  - gcc -g -Werror -Wall -Wextra -pedantic foo.c bar.c -o baz

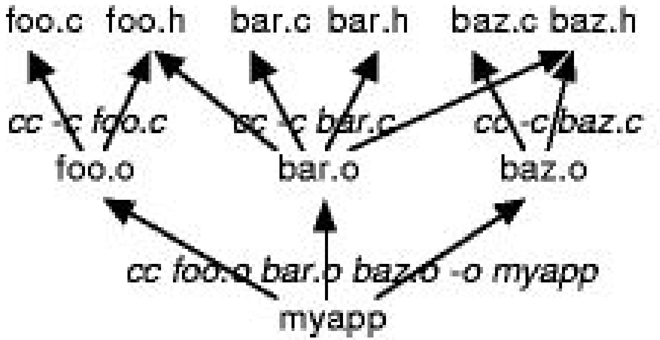
Make Files

- Command-line compilation becomes inefficient when compiling many files together
- Solution: use make-files
- Single operation - ‘make’ to compile files together
- Only recompiles updated files

```
# Makefile for the malloc lab driver
#
CC = gcc
CFLAGS = -Wall -Wextra -Werror -O2 -g -std=gnu99

OBJS = mdriver.o memlib.o

all: mdriver

mdriver: $(OBJS)
	$(CC) $(CFLAGS) -o mdriver $(OBJS)

mdriver.o: mdriver.c memlib.h
	$(CC) $(CFLAGS) mdriver.c
memlib.o: memlib.c memlib.h
	$(CC) $(CFLAGS) memlib.c

clean:
	rm -f *~ *.o mdriver
```

Makefile Rules

- Comments start with a ‘#’, Commands start with a TAB.
- Common Make File Format:
- target: source(s)
  TAB: command
  TAB: command
- Macros: similar to C-macros, find and replace:
- CC   = gcc
  CCOPT = -g -DDEBUG -DPRINT
  foo.o: foo.c foo.h
          $(CC) $(CCOPT) -c foo.c
- See http://www.andrew.cmu.edu/course/15-123-kesden/index/lecture_index.html for more details



Questions?

Appendix

Declaration vs Definition in C

- There can be multiple declarations of an external function or variable
- But there can be only one definition of a function or a variable. I.e. function names/variable names cannot be duplicated

```
#include <stdio.h>

// Unique definition of count
int count ;
// Multiple declarations  of write_extern()
extern void write_extern();

// there can be only one main function among the compiled
// programs
main() {
   count = 5;
   write_extern();
}
```

main.c

```
#include <stdio.h>

// Multiple declaration of count
extern int count;


void write_extern(void) {
   printf("count is %d\n", count);
}
```

support.c

```
#include <stdio.h>

# Multiple declarations
extern int count;
extern void write_extern();

// ERROR: Duplicate definitions of write_extern!!!!
void write_extern(int a) {
    printf(“input var is %d\n”, a);
}
```

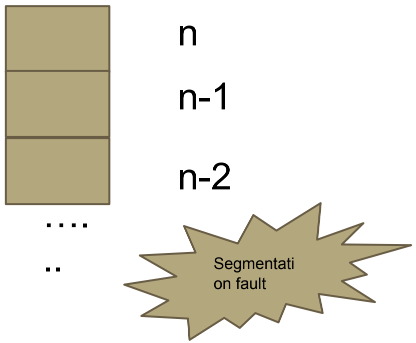
foo.c

gcc main.c support.c foo.c

Recursive Function calls

- Every function call creates a new stack for the called function
- Always remember to have a base case at which the function call returns
- Avoid recursion when you know that the input parameter can be large

```
void recursive_fn(n)
{
        recursive_fn(n-1);
}
```




```
void recursive_fn(n)
{
        If (n==1)
            return;
        recursive_fn(n-1);
}
```