Lecture 5: C Programming
Misc. C topics, File I/O

Dr. Naureen Nizam
nnizam@cs.toronto.edu

CSCB09H3: Software Tools and Systems Programming (SY 110)
Department of Computer and Mathematical Science
Feb 4/6, 2019

Tomorrow is created here.
UNIVERSITY OF TORONTO SCARBOROUGH
1265 Military Trail, Toronto, Ontario M1C 1A4

Administrivia

- **Assignments**
  - A2 Posted
  - Due: March 3, 2019
- **Labs**
  - Lab 3 solutions posted
  - Lab 4 posted – C Pointers, Structs and Linked Lists
- **Midterm**
  - Monday, February 25th (7 PM SW143, SW128, SW319)
  - Wednesday, February 27th (12 PM SW143) – Makeup (only those who are approved)

Last Week

- Strings — KING Chapters 13
- Dynamic memory management — KING Chapters 17
- Structures — KING Chapters 16

The plan for today

- Today:
  - Tricks and Tips
    - Passing arguments to functions
    - Passing command line args to program
    - typedef
    - Organizing your program
    - Makefiles
    - Pre-processor directives
    - Error handling
  - File I/O

KING Chapters 13, 14, 15, 16, 17, 22, 24

Argument passing in C

- Modifying the arguments of a function:

```
void ChangeStudentYear (struct student s, int y) {
     s.year = y;
};

void SwapNumbers (int x, int y) {
     int temp = x;
     x = y;
     y = temp;
};
```

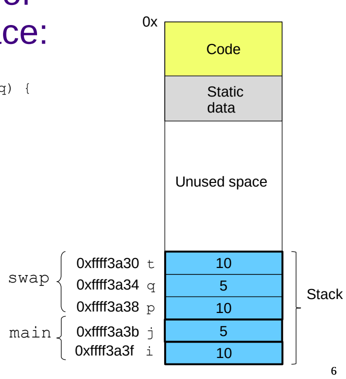
What's wrong with these examples?

Recall use of address space:

```
void swap(int p, int q) {
    int t = p;
    p = q;
    q = t;
}

int main() {

    int i = 10;
    int j = 5;
    swap(i, j);
    return 0;
}
```

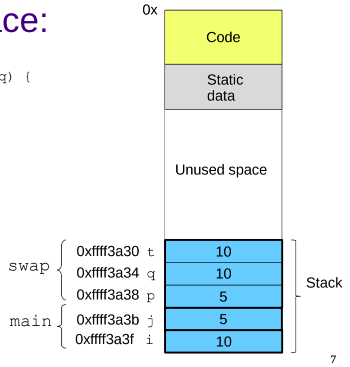


Recall use of address space:

```
void swap(int p, int q) {
    int t = p;
    p = q;
    q = t;
}

int main() {

    int i = 10;
    int j = 5;
    swap(i, j);
    return 0;
}
```



Argument passing in C

```
// WRONG:
void ChangeStudentYear (struct student s, int y) {
     s.year = y;
};

// WRONG:
void swap(int p, int q) {
    int t = p;
    p = q;
    q = t;
}
```

- Parameters in C are *passed by value*, i.e. a copy of each argument is passed to the function.
- The functions works on the copy, not the original variable.

Pointers and functions

- Idea: Pass a pointer to a variable!

```
// CORRECT:
void ChangeStudentYear (struct student *s, int y) {
     s->year = y;
};

// CORRECT:
void swap(int *p, int *q) {
    int t = *p;
    *p = *q;
    *q = t;
}
```


Pointers to pointers???

- Functions might want to change the value of a pointer variable
- E.g. the linked list code in the lab:

```
struct student* Push(int s_id, int year, struct student *head) {
       struct student *p = CreateStudent(s_id, year);
       p->next = head;
       return p;
}

int main(void){
   struct student *my_student_list; // pointer to first node
   …
   my_student_list = Push(4, 4, my_student_list);
```


Pointers to pointers?

- Functions might want to change the value
- E.g. the linked list code in the lab:

How can we change Push, to set list head to the new element?

```
head = p;
// WRONG!
```

Solution: Need to pass a pointer to head.

```
struct student* Push(int s_id, int year, struct student *head) {
       struct student *p = CreateStudent(s_id, year);
       p->next = head;
       return p;
}

int main(void){
   struct student *my_student_list; // pointer to first node
   …
   my_student_list = Push(4, 4, my_student_list);
```


Pointers to pointers

- Modify Push to take a pointer to the list head

```
void  Push(int s_id, int year, struct student **head) {
      struct student *p = CreateStudent(s_id, year);
      p->next = *head;
      *head = p;
}

int main(void){
   struct student *my_student_list; // pointer to first node
   …
   Push(4, 4, &my_student_list);
   …
```

The plan for today

- Today:
  - Tricks and Tips
    - Passing arguments to functions
    - Passing command line args to program
    - Typedef
    - Organizing your program
    - Makefiles
    - Pre-processor directives
    - Error handling
  - File I/O

Passing command line arguments to a C program

- Define main to take two arguments:

```
int main (int argc, char *argv[]) {
    int counter;
    for (counter = 0; counter < argc; counter++){
        printf("%s\n", argv[counter]);
    }
    return 0;
}
```

- `argc` is the number of command line arguments
- `argv[0]` is the name of the program
- The remaining elements of `argv` contain the arguments

The plan for today

- Today:
  - Tricks and Tips
    - Passing arguments to functions
    - Passing command line args to program
    - typedef
    - Organizing your program
    - Makefiles
    - Pre-processor directives
    - Error handling
  - File I/O

typedef

- Example struct from A2 code:

```
struct user {
   char *name;
   double balance;
   struct user *next;
};
```

- Now you can use `struct user` like a built-in data type:

```
struct user s;
```

- typedef allows you to define a short-hand:

```
typedef struct user User;
```

- Now you can use `User` like a built-in data type:

```
User s;
void some_function (User *s);
```


typedef, even shorter

```
typedef struct user {
   char *name;
   double balance;
   struct user *next;
} User;
```

- Now you can use `User` like a built-in data type:

```
User s;
void some_function (User *s);
```

- Caution: the above is not the same as:

```
struct user {
   char *name;
   double balance;
   struct user *next;
} User;
// this declares a variable named User of type struct user
```


The plan for today

- Today:
  - Tricks and Tips
    - Passing arguments to functions
    - Passing command line args to program
    - typedef
    - Organizing your program
    - Makefiles
    - Pre-processor directives
    - Error handling
  - File I/O

Organizing your program

- C does not offer classes or similar concepts to organize your code
- Large programs are organized by breaking them down into multiple files

list.h

```
struct node {
      int value;
      struct node * next;
} ;
typedef struct node List;

int isEmpty(List *);
void add(List *, int);
void remove(List *, int)
```

Compile this with:
`gcc main.c list.c -o myprogram`

main.c

```
#include "list.h"

int main()
{
   List *list1 = NULL;
   add(list1, 10);
   isEmpty(list1);
}
```

list.c

```
#include "list.h"

int isEmpty(List *h)
{...}
void add(List *h, int v)
{...}
void remove(List *h, int v)
{...}
```


What really happens in call to gcc

- E.g. `gcc main.c list.c -o myprogram`

- First the ***pre-processor*** runs, it looks for example for #include directives and includes the corresponding .h file
- Then the ***compiler*** runs on each .c file. It produces machine code (or object code) for each .c file and places it in a .o file (main.o, list.o).
- Then the ***linker*** takes all the object files and combines them into one executable (called myprogram in our example).

Why do I have to recompile my program every time I make an edit?

- Compiling translates the C code to machine level code that runs directly on your hardware
- Every time you make an edit a new executable has to be created

- Why did I not have to do this for shell scripts?
- The commands inside a shell script are interpreted by the shell, they are not translated to a machine language program.
- (Scripting languages are interpreted, not compiled)

Problems with manually calling gcc

- The gcc command can grow quite long:

`gcc file1.c file2.c file3.c file4.c ……… -o myprogram`

- Also recompiles every module, even if it has not changed.
  - This could be addressed by separating compilation & linking:

```
gcc -c list.c    # this produces list.o
gcc -c main.c    # this produces main.o
gcc -o myprogram main.o list.o   # produces executable
```

- Now we could recompile only files that changed.
  - Still not very convenient

Makefiles to simplify your life

- Makefiles are processed by a program called `make`
- Contain information about “targets” and “dependencies”
  - E.g. `myprogram` depends on `list.o` and `main.o`
  - E.g. `list.o` depends on `list.c` and `list.h`
- And “rules”
  - E.g. to produce `list.o` run “`gcc -c list.c`”
- `make` looks at timestamps, and only recompiles a target if one or more of its dependencies are newer.

- Makefiles are a powerful tool, not just for compiling C programs
  - The make reference manual is longer than the C reference manual…

Syntax

Target Prerequisite(s)/Dependencies

```
reverse : reverse.c
  gcc -Wall -o reverse reverse.c
```

Rule Actions(s)

- May be many prerequisites
- Rule may have many actions (one per line)

A sample Makefile

- The makefile to compile A2 (`buxfer.c, lists.c, lists.h`)

```
CC = gcc
CFLAGS = -Wall -g

buxfer: buxfer.o lists.o lists.h
    $(CC) $(CFLAGS) -o buxfer buxfer.o lists.o

buxfer.o: buxfer.c lists.h
    $(CC) $(CFLAGS) -c buxfer.c

lists.o: lists.c lists.h
    $(CC) $(CFLAGS) -c lists.c

clean:
    rm buxfer *.o
```


The plan for today

- Today:
  - Tricks and Tips
    - Passing arguments to functions ✓
    - Passing command line args to program ✓
    - typedef ✓
    - Organizing your program ✓
    - Makefiles ✓
    - Pre-processor directives
    - Error handling
  - File I/O

Pre-processor directives

- Include header files:
  `#include <stdio.h>`
- Define macros:
  `#define MAX 100`
- Conditional inclusions (see lists.h on A2 for example):

```
#ifndef LISTS_H
#define LISTS_H
    ….
#endif
```


The plan for today

- Today:
  - Tricks and Tips
    - Passing arguments to functions ✓
    - Passing command line args to program ✓
    - typedef ✓
    - Organizing your program ✓
    - Makefiles ✓
    - Pre-processor directives ✓
    - Error handling
  - File I/O

Error handling

- The return value of library functions tells you if there was an error.
  - E.g. if `malloc` returns NULL.
- There can be multiple reasons for an error
  - E.g. opening a file might fail because of lack of permissions or because the program has reached the max number of open files
- Many library functions use a global variable called `errno` to store more information on what went wrong.

Error Handling

- `errno` is declared in `errno.h`
  `#include <errno.h>`
- At process creation time **errno** is zero.
- When a library call error occurs, **errno** is set.

- Check `man errno` for possible values of errno.
- Some examples:
  - –ENOMEM: “Not enough space”
  - –EDOM: “Domain error”
  - –EACCESS: “Permission denied”

Careful when using errno

```
if (somecall() == -1) {
        printf("somecall() failed\n");
        if (errno == ...) { ... }
}
```

•What’s wrong?

•`printf` might change the value of errno if it encounters an error.

•Need to save the value of errno, before doing any further processing.

perror()

- `void perror( char *str )`
  - perror displays `str`, then a colon(:), then an English description of the error as defined in errno.h.
- Protocol
  - check system calls for a return value of -1 or NULL
  - call perror() for an error description.

```
char *bufptr;
if ((bufptr = malloc(szbuf)) == NULL) {
    perror("malloc");
    exit(1);
}
```


The plan for today

- Today:
  - Tricks and Tips
    - Passing arguments to functions ✓
    - Passing command line args to program ✓
    - typedef ✓
    - Organizing your program ✓
    - Makefiles ✓
    - Pre-processor directives ✓
    - Error handling ✓
  - File I/O

I/O in C

- Have already seen library functions (from stdio.h) in sample programs:
  `int x = 10;`
  `printf(“The value of x is %d\n”, x);`

- Printing a character:
  –`int putchar(int c);`

- Printing a string
  –`int puts(const char *s);`

- These all print to stdout (standard output = screen)
  –What if we want to print to a file?

File I/O in C

- Have already seen library functions (from stdio.h) in sample programs:
  `int x = 10;`
  `printf(“The value of x is %d\n”, x);`
  –`int fprintf(`**`FILE *stream`**`, const char *format, ...);`

- Printing a character:
  –`int putchar(int c);`
  –`int fputc(int c,` **`FILE *stream`**`);`

- Printing a string
  –`int puts(const char *s);`
  –`int fputs(const char *s,` **`FILE *stream`**`);`

- What is `FILE *stream?`

File interfaces in C/Unix

- Two main mechanisms for managing file access:

- File descriptors (low-level):
  - Each open file is identified by a small integer
  - Use for pipes, sockets (will see later what those are …)

- File pointers (regular files):
  - You use a pointer to a file structure (`FILE *`) as handle to a file.
  - The file struct contains a file descriptor and a buffer.
  - Use for regular files

Standard streams

- All programs automatically have three files open:
  - `FILE *stdin;`
  - `FILE *stdout;`
  - `FILE *stderr;`

	stdio name	File descriptor
Standard input	`stdin`	0
Standard output	`stdout`	1
Standard error	`stderr`	2

Comes by default from the keyboard.

Go by default to the screen.

- Those are ready to use:

```
fprintf(stdout, "Hello!\n"); // Identical to printf("Hello!\n");
fputs("Error!\n", stderr);
```


Operating on regular files

- A file first needs to be opened to obtain a `FILE *`

```
FILE *fopen(const char *filename,
  const char *mode);
```

  - `filename` identifies the file to open.
  - `mode` tells how to open the file:
    - "r" for reading, "w" for writing, "a" for appending
  - returns a pointer to a `FILE` struct which is the handle to the file. This pointer will be used in subsequent operations.
  - To close a file: `void fclose(FILE *stream);`

```
FILE *fp = fopen("my_file.txt", "w");
fprintf(fp, "Hello!\n");
fclose(fp);
```


Reading from files

- `char *fgets(char *s, int size, FILE *stream);`
  - Reads until a `\n` or `size-1` characters, whichever is first
  - Stores characters in buffer `s` points to
  - If a newline is read, it is stored into the buffer too.
  - Always null-terminates the string

- How would you use fgets to read from the keyboard?

- There is also another function to read from keyboard:
  `char *gets(char *s);`
  - Reads from keyboard until `\n` and stores results in buffer `s` points to
  - Why should you <u>never</u> use `gets`?

Example for reading from files

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 100

int main() {
   char buffer[MAX];
   FILE *fp = fopen("my_file.txt", "r");

   if( fp == NULL )
   {
      perror("Error while opening the file.\n");
      exit(1);
   }

    while ((fgets(buffer, MAX, fp)) != NULL){
        printf("%s", buffer);
    }
    fclose(fp);
```


Block I/O (aka binary I/O)

- Suppose a program wants to store a large set of numbers to file for later re-use (not human consumption).
- Could use fprintf for that. Why shouldn't we?

- Suppose you store the number 1,999,999 as follows:
  `fprintf(fp, "1999999");`

- How many bytes does this take up in the file?
  00110001 00111001 00111001 00111001 00111001 00111001 00111001
- How many bytes should it really take to store this number?
  00011110 10000100 01111111
- Block I/O allows you to read and write binary data, i.e. you read and write byte-for-byte rather than lines of characters.

Binary I/O

- Recall that `fgets` reads characters.
- By contrast, `fread` and `fwrite` operate on bytes.

```
size_t  fread(void *ptr, size_t size,
              size_t nmemb, FILE *stream);
```

  – read `nmemb` * `size` bytes into memory at `ptr`
  – returns number of items read

```
size_t fwrite(const void *ptr, size_t size,
              size_t nmemb, FILE *stream);
```

  – write `nmemb` * `size` bytes from `ptr` to the file pointer `stream`
  – returns number of items written

Example using
```
size_t fwrite(const void *ptr, size_t size,
              size_t nmemb, FILE *stream);
```

```
/* write an integer to the file */
int num = 1999999;
n = fwrite(&num, sizeof(num), 1, fp);


/* write a struct to the file */
struct rec {
  char name[20];
  int num;
} r;
r.num = 42;
strncpy(r.name, "koala", 20);
n = fwrite(&r, sizeof(r), 1, fp);
```


Example using
```
size_t  fread(void *ptr, size_t size,
              size_t nmemb, FILE *stream);
```

```
/* read an integer from the file */
int num;
n = fread(&num, sizeof(num), 1, fp);

/* read a struct from the file */
struct rec r;
n = fread(&r, sizeof(r), 1, fp);

/* display the contents of the variables */
printf("%d %s %d\n", num, r.name, r.num);
```


That's it for today!