CSCI-243 Exam 2 Review

Presented by the RIT Computer Science Community

February 22, 2015

http://csc.cs.rit.edu

C Preprocessor

1. Consider the following program:

```
1  #include <stdio.h>
2
3  #ifdef WINDOWS
4  #include <windows.h>
5
6  #define WIN_CREATEFILE(a) CreateFile(a, \
7                                       GENERIC_READ, \
8                                       FILE_SHARE_READ, \
9                                       OPEN_ALWAYS, \
10                                      FILE_ATTRIBUTE_NORMAL)
11 #endif
12
13 int main(int argc, char** argv) {
14     // ... some previous initialization stuff
15
16 #ifdef WINDOWS
17     HANDLE file = WIN_CREATEFILE(argv[1]);
18 #else
19     FILE* file = fopen(argv[1], "r");
20 #endif
21
22     // ... do some more stuff
23
24     return 0;
25 }
```

   (a) What flag would you use to enable the Windows specific build?
   `-DWINDOWS` This causes `WINDOWS` to be `#defined` at compile time.

   (b) Why is this useful?
   You can use the same source for building on multiple platforms.

   (c) Why is the macro useful?
   It condenses a long function call into a simple call with standard arguments.

Memory Management

2. Consider the following statement in a larger program:

```
int* x = (int*) malloc(20); //create an array of 20 ints
```

   After testing the program, you notice that the values of $x[5], x[6], \cdots, x[19]$ keep changing unexpectedly.

   (a) Why is this?

   (b) What should the statement actually be?

   The argument to `malloc` should be the number of bytes to allocate, not the number of elements. Since ints are usually 4 bytes long, the array only has enough room to hold 5 ints ($20/4 = 5$).

   A proper (and portable) statement should be: `int* x = (int*) malloc(sizeof(int)*20);`

3. The following program compiles.

(a) Will the program crash at run time? If so, on which line will it crash?

```
1 #include <stdlib.h>
2 int main(int argc, char **argv)
3 {
4     int *x = NULL;
5     int *y = NULL;
6     int *z = NULL;
7
8     x = (int *) malloc(sizeof(int) * 10);
9     y = (int *) malloc(20);
10    x = (int *) malloc(sizeof(char) * 50);
11
12    free(x);
13    free(y);
14    free(z);
15
16    return 0;
17 }
```

No, it runs and terminates normally. `free` will not do anything if the pointer passed to it is `NULL`.

(b) What tool can you use to find memory leaks? What options would you use?

`valgrind --leak-check=full <program name>`

(c) What output would you get from part b? Is there a memory leak? (If so, where?)

Something similar to "40 bytes lost". Yes, the first call to `malloc` is never freed.

4. (a) Given the following:

```
struct Point {
    char label;
    double x;
    double y;
};
```

What (specifically) happens when the following command is executed?

```
struct Point *newPoint = (struct Point *) malloc( sizeof(struct Point) );
```

The `malloc` command allocates space in memory for the entire `Point` struct. Given the known sizes of `char` and `double` (x and y respectively), enough space to hold one `char` and two `doubles` are set aside in memory. Then the memory address of the newly-allocated space is returned and assigned to the `struct Point*`.

(b) Let's add some more information:

```
typedef struct {
    struct Point p1;
    struct Point p2;
    struct Point p3;
} Triangle;
```

What (specifically) happens when the following command is executed?

```
Triangle *tri = (Triangle *) malloc( sizeof(Triangle) );
```

The same thing happens, but now we need to know the size of `Triangle` from the last question. Based on the definitions provided, this means allocating exactly enough space for three `Point` structs. If the size of a `Point` struct is `X` bits, then exactly `3X` bits will be allocated in memory. Then the memory address is returned as before.

Program Translation

5. (a) List the four main steps in the *program* translation process.

   i. Compiler: Translates C code into assembly code.

   ii. Assembler: Translates assembly code into a *relocatable object module*, which are in machine language.

   iii. Linker: Combines object modules and static libraries (archives) into a single load module, which is also in machine language.

   iv. Loader: Part of OS. Links a load module with dynamic link libraries (shared libraries), if applicable, and runs the program.

   (b) List the four phases of *source code* translation, which make up the compilation step of program translation.

   i. Lexical analysis (scanner): Translates source code text into a sequence of tokens.

   ii. Syntax analysis (parser): Validates tokens to see if they are legal with respect to the language's grammar.

   iii. Semantic analysis: Checks and determines the meaning of token sequences and produces output in an intermediate language.

   iv. Code generation: Translates the intermediate output into assembly language, to be processed by the assembler.

Abstract Data Types

6. Generally speaking, ADTs are easier to define and work with in procedural languages like C, as opposed to object-oriented languages like Java or C#. (**True** or **False**). Explain your answer.

   False. C lacks object-oriented features that streamline the creation and use of ADTs in the language. C doesn't have access modifiers (e.g., `private` or `protected`), so hiding the underlying implementations is tougher, and will often involve the use of pointers.

Makefiles

7. Consider the following makefile:

```
1  CFLAGS := -std=c99 -Wall -Wextra
2  me: me.o
3      $(CC) $(LDFLAGS) -o $@ $^ $(LDLIBS)
4  calc: calc.o real.o
5      $(CC) $(LDFLAGS) -o $@ $^ $(LDLIBS)
6  calc.o: calc.c
7      $(CC) $(CFLAGS) -c $<
8  real.o: real.c
9      $(CC) $(CFLAGS) -c $<
10 .PHONY: clean
11 clean:
12     $(RM) me calc *.o
```

   (a) Ralph is annoying. One day, when Ralph makes a particularly unreasonable demand, his friend loses it and shouts, "Why don't you make me???" Always one to take things literally, Ralph pops open his favorite Bourne-compatible shell and types: `make me`. However, he is confronted with the message: `make: 'me' is up to date`.

   Explain the meaning of this message. Which (if any) of the relevant files are now located in Ralph's directory? What numeric values did `Make` compare before outputting this message?

   This output indicates that `Make` did not need to rebuild the `me` executable because:

   - the files `me.o` and `me` both existed already
   - and the modification timestamp of `me.o` was earlier than that of `me`.

   After the command completes, both files are still present in Ralph's directory.

   (b) Ralph is childishly proud of himself, but everyone just groans and tells him to "get real". Coincidentally, Ralph has a library providing functions for working with reals, as well as a calculator program to test the functionality. Ready to win another trivial victory, Ralph types `make clean` and then `make calc`.

   List the exact sequence of commands that are executed as a result of this new invocation.

   Because the `clean` target eliminated all the object files and executables, Make will decide it needs to rebuild everything:

```
$ cc -std=c99 -Wall -Wextra -c calc.c
$ cc -std=c99 -Wall -Wextra -c real.c
$ cc -o calc calc.o real.o
```

File I/O

8. Write a program that searches a file for a provided number on `stdin`. Print out any errors on `stderr`. Example:

```
$ fileSearch file.txt
> 234
found: 234
```

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

int main(int argc, char* argv[]) {
    if(argc != 2) {
        fprintf(stderr, "Usage: fileSearch <filename>");
        exit(EXIT_FAILURE);
    }
    FILE* handle = fopen(argv[1], "r");
    if(!handle) {
        perror("fopen failed:");
        exit(EXIT_FAILURE);
    }
    int input = 0;
    scanf("\%d", &input);
    int check = 0;
    while(fscanf(handle, "\%d", &check) != EOF) {
        if(check == input) {
            printf("found: \%d", input);
            exit(EXIT_SUCCESS);
        }
    }
    fclose(handle);
    printf("\%d not found", input);
    return EXIT_SUCCESS;
}
```

9. (a) The following program is intended to read a text file and outputs (as binary data) the number of characters (including new lines) in each line to a file. However, there is a problem with the code the way it is currently written. Find the problem and explain how to fix it. Note: The code is compiled and linked using GCC.

```
1  #define _GNU_SOURCE
2
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int main(void) {
7      FILE* inputfile = fopen("input.txt", "r");
8      if(!inputfile) {
9          perror("fopen failed for inputfile:");
10         exit(EXIT_FAILURE);
11     }
12     int numchars = 0;
13     char* line = NULL;
14     FILE* outputfile = fopen("output.txt", "w");
15     if(!outputfile) {
16         perror("fopen failed for outputfile:");
17         fclose(inputfile);
18         exit(EXIT_FAILURE);
19     }
20     while((numchars = getline(line, 1024, inputfile)) != -1) {
21         fprintf(outputfile, "%d\n", numchars);
22     }
23     fclose(inputfile);
24     fclose(outputfile);
25     return EXIT_SUCCESS;
26 }
```

The signature of `getline` is `ssize_t getline(char** lineptr, size_t* n, FILE* stream)`. If `*lineptr` is `NULL` and `*n` is 0, then `getline` will dynamically allocate a C string containing the next line of `stream` (including the new line character), and set `*lineptr` to this C string, and `*n` to the size of this new stream. It returns the size of the string read. If the `*lineptr` given to the function is not `NULL`, then it expects `*n` to be the length of `*lineptr`, and it will attempt to use `*lineptr` to hold the next line if it can fit, otherwise it will use `realloc` to allocate enough space for it to fit. This program does not use `getline` correctly, and causes a type error when compiling. See the answer of the next question for how to fix this problem.

(b) Rewrite the code so that it outputs the data using binary streams instead.

```c
#define _GNU_SOURCE

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

int main(void) {
    FILE* inputfile = fopen("input.txt", "r");
    if(!inputfile) {
        perror("fopen failed for inputfile:");
        exit(EXIT_FAILURE);
    }
    ssize_t numchars = 0;
    char* line = NULL;
    FILE* outputfile = fopen("output.txt", "wb");
    if(!outputfile) {
        perror("fopen failed for outputfile:");
        fclose(inputfile);
        exit(EXIT_FAILURE);
    }
    size_t n = 0;
    while((numchars = getline(&line, &n, inputfile)) != -1) {
        int towrite[1] = { numchars };
        if(fwrite((void*)towrite, sizeof(int), 1, outputfile) != 1) {
            perror("fwrite into outputfile failed:");
            fclose(inputfile);
            fclose(outputfile);
            exit(EXIT_FAILURE);
        }
    }
    free(line);
    fflush(outputfile);
    fclose(inputfile);
    fclose(outputfile);
    return EXIT_SUCCESS;
}
```