CS 201
Exam 1 Review More on Makefiles

Debzani Deb

Announcements

- MSU Book Store: The official last day to buy books is March 20, 2007.
- ACM Meeting: This Thursday at 5pm in the CS conference room.

Exam 1 review of critical Qs

Answer to Q: 1(c)

```
#include <stdio.h>
#include <math.h>
double  Root_product(double x, double y);
int main(void) {
      double first, second;
      double result;
      printf("Please enter two positive real numbers\n");
      scanf("%lf %lf", &first, &second);
      result = Root_product(first, second);
      printf("The square root of their product is %f\n", result);
      return 0;
}
double Root_product (double x, double y) {
      double result;
      result = sqrt(x*y);
      return result;
}
```


Question 2

- Q2(c): Challenging
  - If you execute the code

    ```
    int i = 7;
    printf("%d\n", i++ * i++);
    ```

  - prints 49. Shouldn't it print 56?
  - The behavior of code which contains side effects like this has always been undefined. So avoid them.
  - You can learn more about this in http://c-faq.com/expr/evalorder2.html

Question 3

- Q 3(a): What is the output if the input is 3?

```
scanf("%d", &n);
if (n = 5)
    printf("Equal\n");
else if (n < 5)
    printf("Less\n");
else
    printf("Greater\n");
```

- Condition (n=5) is always true.
- See Lecture 7 for more on this.

Question 4 (a): Tricky

What is printed out by the following C code fragment?

```
int i;
for (i = 1; i <= 100; i++)
    if ((i % 3) && (i % 5))
        printf("%d\n", i);
```

i. All integers between 1 to 100 inclusive.
ii. All integers between 1 to 100 inclusive that are divisible by 3 or 5.
iii. All integers between 1 to 100 inclusive that are divisible by 15.
iv. All integers between 1 to 100 inclusive that are not divisible by 3 or 5.
v. All integers between 1 to 100 inclusive that are not divisible by 15.

- The if condition becomes false when
  - input i is divisible by 3, 5 or both.
- If you are not sure about the output, then try hand tracing for few inputs and apply process of elimination.

Question 4 (b): More Tricky

What is printed out by the following C code fragment?

```
int i;
for (i = 1; i <= 100; i++)
    if (!((i % 3) || (i % 5)))
      printf("%d\n", i);
```

i. All integers between 1 to 100 inclusive.
ii. All integers between 1 to 100 inclusive that are divisible by 3 or 5.
iii. All integers between 1 to 100 inclusive that are divisible by 15.
iv. All integers between 1 to 100 inclusive that are not divisible by 3 or 5.
v. All integers between 1 to 100 inclusive that are not divisible by 15.

- !((i % 3) || (i % 5)) = !(i%3) && !(i%5)
- So if becomes true when input i is divisible by both 3 and 5.

Question 4(c)

```
st = scanf("%d%d%d", &n, &m, &p);
printf("n = %d m = %d st = %d\n", n, m, st);
```

Input data: 10 12 hello

Answer: n = 10 m = 12 st = 2

- Lecture 8 and p. 242 of HK
- If `scanf` faces invalid/insufficient data, the function returns as its value the number of successful scans before encountering the problem.

Question 4(d)

- Write a do-while loop that repeatedly scans integer values until a positive even number is input.
- We want the while (condition) to be **false** when user enters a positive even number**.**
- **while (negative || odd),** the loop should continue.
- Answer:

```
do
scanf("%d", &num);
while ((num <= 0) || (num % 2 != 0));
```


Question 4(e): Tricky

- What is printed out by the following C code fragment

```
int a = 1.5;
if (a == 1.5)
    printf("1\n");
else
    printf("2\n");
```

i. 1
ii. 2
iii. System dependent.
iv. A compilation error occurs.
v. A run-time error occurs.

Question 5(a)

- Complete function `incr` so its effect mimics the prefix increment of an integer variable: that is, both `++i` and `incr(&i)` would add 1 to i.

```
int incr(int *argp)
{
  ++(*argp);
  return(*argp);
}
```

Question 5(c)

- What's wrong with the following code?

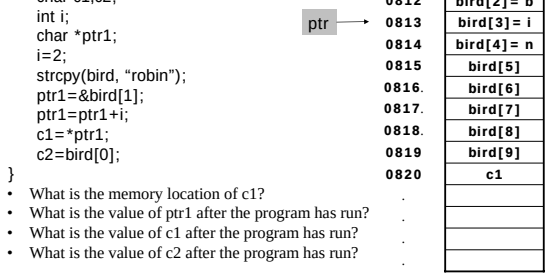
  int *p = 20;

  This is actually declaring a pointer variable p and initializing it with memory address 20 at the same time.
- The point of this question was to test whether you remember the usage of operator * in pointer declaration and in regular use such as *p = 20.
- If you execute this code like this, there is a warning:

  Initialization makes pointer from integer without a cast.
- If you do proper casting such as int *p = (int *) 20; the address of p becomes 0x14(20 in decimal).

Question 5(d)-(g)

```
void main() {
    char bird[10]; /* 0810 */
    char c1,c2;
    int i;
    char *ptr1;
    i=2;
    strcpy(bird, "robin");
    ptr1=&bird[1];
    ptr1=ptr1+i;
    c1=*ptr1;
    c2=bird[0];
}
```



- What is the memory location of c1?
- What is the value of ptr1 after the program has run?
- What is the value of c1 after the program has run?
- What is the value of c2 after the program has run?

Question 5(h)

- Calling program:

  c = funct(&a, &b);
- In the called program

```
int funct(int *fa, int *fb) {
        scanf("...", fa, fb);
}
```

- Since formal parameters `fa` and `fb` are addresses already, if you pass it on to a `scanf`, you don't need the & operator.

Question 5(i)

- We have seen that all pointer variables (i.e. int *, double *) are of same size (in our case all of them occupy 4 bytes of memory and contain plain and simple addresses). Considering this, can we live without declaring the type of a pointer variable? Whether your answer is 'yes'/'no', please explain.
- My Answer: No
- What will happen in case of pointer arithmetic operations such as p++ or p+2? If we do not know the type p is pointing to, how do we know which memory location to go to?

Makefiles

#include "proto.h"

- I like to put all my prototypes in a separate file and then include this file in the main program.
- Why? It makes my modules more convenient for re-using.
- When you do it this way, you need some additional pre-processor directives to avoid duplication.

Sample proto.h

```
/* proto.h – Function Prototypes
  */

#ifndef _PROTO_H
#define _PROTO_H
int fun1(int,int,int);
double fun2(double,int);
float funn(int);
#endif
```

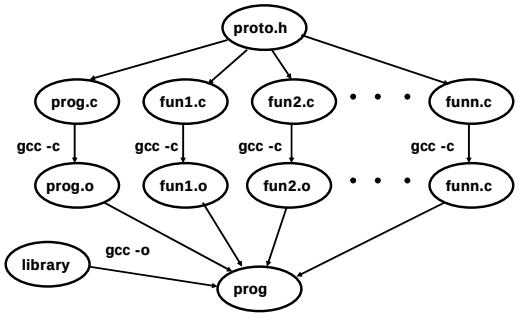

Avoids Duplication

- First it checks to see if the variable _PROTO_H is defined.
- If it IS defined, the entire file is skipped.
- If it ISN'T defined, the first thing done is to define it.
- Then the rest of the file is processed.
- If this file is included twice, the compiler will only see one copy of it!

Why did I use a strange variable name?

- _PROTO_H is used so that it never conflicts with any normal variables in any programs.
- I usually use the file name since that is pretty unique.
- You'll see this done in all sorts of ways.

Using Multiple Source Files



Sample Makefile

```
# Makefile for multiple file example
prog: prog.o fun1.o fun2.o funn.o
        gcc -o prog prog.o fun1.o fun2.o funn.o
prog.o: prog.c proto.h
        gcc -c prog.c
fun1.o: fun1.c
        gcc -c fun1.c
fun2.o: fun2.c proto.h
        gcc -c fun2.c
fun3.o: fun3.c
        gcc -c fun3.c
clean:
        rm *.o prog
```


What does it do?

- `make` is a program that looks for a file called "Makefile".
- `Makefile` contains variables, dependency rules and comments.
- When a `Makefile` is executed, a specific target is executed. A target is just some name that appears at the beginning of a dependency rule.

Dependency rules (1)

- Dependencies are the heart of makefiles; without them nothing would work. They have the following form:

```
dependency1: depA, depB …..depN
  command for dependency1
```

- dependency1 is dependent upon depA, and depB, upto however many dependencies you have.
- The program `make`, looks at dependency1 and sees it is dependent upon depA, so then it looks later in the makefile for how depA is defined.

Dependency rules (2)

- A rule does not have to be used in every invocation of make. Example: the "clean" rule is not normally used.
- A rule does not need to have any dependencies. This means if we tell make to handle its target, it will always execute its command list, as in the "clean" rule below.
- A rule does not need to have any commands. For example, the "all" rule is just used to invoke other rules, but does not need any commands of its own.

```
# top-level rule to create program
all: prog
…
# Cleaning everything that could be automatically
# recreated by "make"
clean:
   rm –f prog *.o
```


Compiler Options

- Depend on the compiler, but many similar ones.
  gcc {…. Options….} file_names;
- You have seen
  - -c : compile only
  - -o : define executable file name.
- Here are few others
  - -g : inserts debugging statements
  - -Wall : Print all warnings. Allows to write cleaner executing code.
  - -E : preprocess only and produce the results of all preprocessor directives.
- If you have problem linking your math library:
  gcc –lm –o file_names;

Have a nice Spring Break!!!