APS105 Lecture 11
5.5-5.6

Dan Zingaro

February 12, 2010

Feedback from Reading Quiz

Some confusion on question 1. Let's talk about it!

```
#include <stdio.h>

int i;

void print_one_row(void)
{
  for (i = 1; i <= 10; i++)
    printf("*");
}

int main(void)
{
  for (i = 1; i <= 10; i++) {
    print_one_row();
    printf("\n");
  }
  return 0;
}
```

Midterm

- Content
  - First five chapters of textbook
  - Focus on concepts from lecture and reading quizzes
  - Questions I give in class always relate to important points
- Types of questions
  - Tracing code, writing code, putting code in the correct order
  - Explaining concepts **without writing code** (like our concepTests!)

Review: Functions that Change Variables

- For each variable we want a function to change, put a * in front of it in the function's first line
- Each time we want to access the value pointed to by one of these parameters, prefix it with a *
- When calling the function, pass pointer variables directly or use & on an `int`, `float`, etc.

```
void swap (int *m, int *n) {
  int temp = *m;
  *m = *n;
  *n = temp;
}
```

ConcepTest

What is printed by the following code?

```
void addOne (int *i, int *j) {
  *i = *i + 1;
  *j = *j + 1;
}

int main (void) {
  int a = 3;
  addOne (&a, &a);
  printf ("%d\n", a);
  return 0;
}
```

- A. 3
- B. 4
- C. 5
- D. Error!

Variables Declared in Functions

- A variable declared in the body of a function is local to that function
- **Scope**: section of program text in which a variable can be used
- **Block**: section of code enclosed by { } braces
- Each local variable is declared in a block
- Scope of local variable starts at the identifier's declaration and extends until the closing } of that block (block scope)

```
int func (void) {
  int i = 4; /*only i here*/
  if (i == 2) {
    int j; /*i and j here*/
    ...
  }
  /*now only i again*/
}
```

ConcepTest

Which variables can be accessed at the line with the comment?

```
void sample(void) {
  int i = 3;
  if (i == 3) {
    int j;
  }
  {
    int i;
  }
  int k;
  //What can I access here?
}
```

- A. `i`
- B. `i, j`
- C. `i, j, k`
- D. `i, k`
- E. `k`

Types of Local Variables

- By default, local variables have **automatic storage duration**
  - This means that a variable is destroyed when the end of the scope is reached, and re-created next time the variable is in scope
  - Variable must be initialized each time it is created (or it will contain garbage)
  - Think of parameters as automatic local variables
- We can also declare local variables with the keyword `static`; they have **static storage duration**
  - Variable is not destroyed when it goes out of scope
  - Next time the function is called, the variable has the same value it had before
  - Initialization of `static` local variables is done only once, prior to program execution

Example: Static Local Variable

```
#include <stdio.h>

void doSomething (void) {
  static int counter = 0;
  counter++;
  printf ("Call %d.\n", counter);
}

int main (void) {
  doSomething();
  doSomething();
  doSomething();
  return 0;
}
```

Variables Declared Outside of Functions

- We can also declare variables outside of functions (i.e. not in any `{ }` block)
- These are often called global variables
- They have static storage duration, and their scope is from their declaration to the end of the file (**file scope**)
- e.g. we can write `const double E = 2.71828;` outside of any function to define a global constant `E`

Variables Declared Outside of Functions...

- We usually stay away from declaring variables outside of functions
- In large programs, it's not obvious which functions are using a global variable
- If we change the value or name of a global variable, we don't know which functions we're affecting
- Using only parameters, the variables used by each function are clear

ConcepTest

What is printed by this program fragment?

```
...
int doSomething (int x) {
  static int y = 1;
  y += x;
  return y;
}

int main (void) {
  doSomething (3);
  doSomething (4);
  printf ("%d\n", doSomething (1));
  ...
```

A. 7 | B. 8 | C. 9 | D. 1 | E. 2

Organizing Larger Programs

- We do not have to stuff all of our functions into the same `.c` file
- What we can do is create multiple `.c` files, each with related function definitions
- For each `.c` file, we create a `.h` file that contains the prototypes for some of the functions in that `.c` file
- We only write prototypes in the `.h` file for the functions we want other `.c` files to be able to use
- Why do this?
  - Makes the organization of the program clear
  - Allows us to more easily reuse functions in other programs

Example: Program Organization

- We often use random-number functions in our code
  - A function to initialize the random-number generator
  - A function to get a random integer between lower bound `i` and upper bound `j`
- If we create a `.c` and a `.h` file for these functions, we can easily use them in our programs
- `randfuncs.h` gives the prototypes for the functions that we will write
- `randfuncs.c` gives the function definitions

Example: Program Organization... (randfuncs.h)

```
/*Initialize random number generator*/
void initRand (void);

  /*Return a random integer in the range i..j*/
int randBetween (int i, int j);
```

Example: Program Organization... (randfuncs.c)

```
#include <stdlib.h>
#include <time.h>
#include "randfuncs.h"

/*Initialize random number generator*/
void initRand (void) {
  srand (time(0));
}

/*Return a random integer in the range i..j*/
int randBetween (int i, int j) {
  int nums = j - i + 1;
  return rand() % nums + i;
}
```

Example: Program Organization... (randmain.c)

We can use these functions to write a program that rolls three dice.

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

int main (void) {
  initRand();
  for (int i = 1; i <= 3; i++)
    printf ("Rolled a %d\n", randBetween (1, 6));
  return 0;
}
```