6.087 Lecture 3 – January 13, 2010

- Review
- Blocks and Compound Statements
- Control Flow
  - Conditional Statements
  - Loops
- Functions
- Modular Programming
- Variable Scope
  - Static Variables
  - Register Variables

Review: Definitions

- Variable - name/reference to a stored value (usually in memory)
- Data type - determines the size of a variable in memory, what values it can take on, what operations are allowed
- Operator - an operation performed using 1-3 variables
- Expression - combination of literal values/variables and operators/functions

Review: Data types

- Various sizes (**char**, **short**, **long**, **float**, **double**)
- Numeric types - **signed**/**unsigned**
- Implementation - little or big endian
- Careful mixing and converting (casting) types

Review: Operators

- Unary, binary, ternary (1-3 arguments)
- Arithmetic operators, relational operators, binary (bitwise and logical) operators, assignment operators, etc.
- Conditional expressions
- Order of evaluation (precedence, direction)

6.087 Lecture 3 – January 13, 2010

- Review
- Blocks and Compound Statements
- Control Flow
  - Conditional Statements
  - Loops
- Functions
- Modular Programming
- Variable Scope
  - Static Variables
  - Register Variables

Blocks and compound statements

- A simple statement ends in a semicolon:

```
z = foo(x+y);
```

- Consider the multiple statements:

```
temp = x+y;
z = foo(temp);
```

- Curly braces – combine into compound statement/*block*

Blocks

- Block can substitute for simple statement
- Compiled as a single unit
- Variables can be declared inside

```
{
   int temp = x+y;
   z = foo(temp);
}
```

- Block can be empty { }
- No semicolon at end

Nested blocks

- Blocks nested inside each other

```
{
  int temp = x+y;
  z = foo(temp);
  {
    float temp2 = x*y;
    z += bar(temp2);
  }
}
```

6.087 Lecture 3 – January 13, 2010

- Review
- Blocks and Compound Statements
- Control Flow
  - Conditional Statements
  - Loops
- Functions
- Modular Programming
- Variable Scope
  - Static Variables
  - Register Variables

Control conditions

- Unlike C++ or Java, no *boolean* type (in C89/C90)
  - in C99, bool type available (use `stdbool.h`)
- Condition is an expression (or series of expressions)
  e.g. `n < 3` or `x < y || z < y`
- Expression is non-zero ⇒ condition true → NON-ZERO.
- Expression must be numeric (or a pointer)

```
const char str[] = "some text";
if (str) /* string is not null */
  return 0;
```

IMP

similarly while (1) → evaluates to → TRUE
{

}

Conditional statements

- The `if` statement
- The `switch` statement

The `if` statement

```
if (x % 2)
  y += x/2;
```

- Evaluate condition
  `if (x % 2 == 0)`
- If true, evaluate inner statement
  `y += x/2;`
- Otherwise, do nothing

The `else` keyword

```
if (x % 2 == 0)
  y += x/2;
else
  y += (x+1)/2;
```

- Optional
- Execute statement if condition is false

  y += (x+1)/2;
- Either inner statement may be block

The `else if` keyword

```
if (x % 2 == 0)
  y += x/2;
else if (x % 4 == 1)
  y += 2*((x+3)/4);
else
  y += (x+1)/2;
```

- Additional alternative control paths
- Conditions evaluated in order until one is met; inner statement then executed
- If multiple conditions true, only first executed
- Equivalent to nested `if` statements

Nesting `if` statements

```
if (x % 4 == 0)
  if (x % 2 == 0)
    y = 2;
else
  y = 1;
```

To which `if` statement does the `else` keyword belong?

Nesting `if` statements

To associate `else` with outer `if` statement: use braces

```
if (x % 4 == 0) {
  if (x % 2 == 0)
    y = 2;
} else
  y = 1;
```

The `switch` statement

- Alternative conditional statement
- Integer (or character) variable as input
- Considers cases for value of variable

```
switch (ch) {
  case 'Y': /* ch == 'Y' */
    /* do something */
    break;
  case 'N': /* ch == 'N' */
    /* do something else */
    break;
  default: /* otherwise */
    /* do a third thing */
    break;
}
```

Multiple cases

- Compares variable to each case in order
- When match found, starts executing inner code until `break;` reached
- Execution “falls through” if `break;` not included

```
switch (ch) {
  case 'Y':
  case 'y':
    /* do something if
       ch == 'Y' or
       ch == 'y' */
    break;
}
```

No Break.

```
switch (ch) {
  case 'Y':
    /* do something if
       ch == 'Y' */
  case 'N':
    /* do something if
       ch == 'Y' or
       ch == 'N' */
    break;
}
```

The `switch` statement

- Contents of `switch` statement a block
- Case labels: different entry points into block
- Similar to labels used with `goto` keyword (next lecture...)

Loop statements

- The `while` loop
- The `for` loop
- The `do-while` loop
- The `break` and `continue` keywords

The `while` loop

```
while (/* condition */)
  /* loop body */
```

- Simplest loop structure – evaluate body as long as condition is true
- Condition evaluated first, so body may never be executed

The `for` loop

```
int factorial(int n) {
  int i, j = 1;
  for (i = 1; i <= n; i++)
    j *= i;
  return j;
}
```

- The “counting” loop
- Inside parentheses, three expressions, separated by semicolons:
  - Initialization: `i = 1`
  - Condition: `i <= n`
  - Increment: `i++`
- Expressions can be empty (condition assumed to be “true”)

The `for` loop

Equivalent to `while` loop:

```
int factorial(int n) {
  int j = 1;
  int i = 1; /* initialization */
  while (i <= n /* condition */) {
    j *= i;
    i++; /* increment */
  }
  return j;
}
```

The `for` loop

- Compound expressions separated by commas

```
int factorial(int n) {
  int i, j;
  for (i = 1, j = 1; i <= n; j *= i, i++)
    ;
  return j;
}
```

- Comma: operator with lowest precedence, evaluated left-to-right; not same as between function arguments

The do-while loop

```
char c;
do {
  /* loop body */
  puts("Keep going? (y/n) ");
  c = getchar();
  /* other processing */
} while (c == 'y' && /* other conditions */);
```

- Differs from `while` loop – condition evaluated after each iteration
- Body executed at least once
- Note semicolon at end

– Regular while / for DON'T have a (;) at the end.

The `break` keyword

- Sometimes want to terminate a loop early
- **break**; exits innermost loop or `switch` statement to exit early
- Consider the modification of the `do-while` example:

```
char c;
do {
  /* loop body */
  puts("Keep going? (y/n) ");
  c = getchar();
  if (c != 'y')
    break;
  /* other processing */
} while (/* other conditions */);
```

The `continue` keyword

- Use to skip an iteration
- **continue**; skips rest of innermost loop body, jumping to loop condition
- Example:

```
#define min(a,b) ((a) < (b) ? (a) : (b))

int gcd(int a, int b) {
  int i, ret = 1, minval = min(a,b);
  for (i = 2; i <= minval; i++) {
    if (a % i) /* i not divisor of a */
      continue;
    if (b % i == 0) /* i is divisor of both a and b */
      ret = i;
  }
  return ret;
}
```

6.087 Lecture 3 – January 13, 2010

- Review
- Blocks and Compound Statements
- Control Flow
  - Conditional Statements
  - Loops
- Functions
- Modular Programming
- Variable Scope
  - Static Variables
  - Register Variables

Functions

- Already seen some functions, including `main()`:

```
int main(void) {
  /* do stuff */
  return 0; /* success */
}
```

- Basic syntax of functions explained in Lecture 1
- How to write a program using functions?

Divide and conquer

- Conceptualize how a program can be broken into smaller parts
- Let's design a program to solve linear Diophantine equation ($ax + by = c$,x, y: integers):

```
get a, b, c from command line
compute g = gcd(a,b)
if (c is not a multiple of the gcd)
  no solutions exist; exit
run Extended Euclidean algorithm on a, b
rescale x and y output by (c/g)
print solution
```

- Extended Euclidean algorithm: finds integers x, y s.t.

$$ax + by = \mathrm{gcd}(a, b).$$

Computing the gcd

- Compute the gcd using the Euclidean algorithm:

```
int gcd(int a, int b) {
  while (b) { /* if a < b, performs swap */
    int temp = b;
    b = a % b;
    a = temp;
  }
  return a;
}
```

- Algorithm relies on $\gcd(a, b) = \gcd(b, a \bmod b)$, for natural numbers $a > b$.

[Knuth, D. E. The Art of Computer Programming, Volume 1: Fundamental Algorithms. 3rd ed. Addison-Wesley, 1997.]

Extended Euclidean algorithm

Pseudocode for Extended Euclidean algorithm:

```
Initialize state variables (x,y)
if (a < b)
  swap(a,b)
while (b > 0) {
  compute quotient, remainder
  update state variables (x,y)
}
return gcd and state variables (x,y)
```

[Menezes, A. J., et al. Handbook of Applied Cryptography. CRC Press, 1996.]

Returning multiple values

- Extended Euclidean algorithm returns gcd, and two other state variables, x and y
- Functions only return (up to) one value
- Solution: use *global* variables
- Declare variables for other outputs outside the function
  - variables declared outside of a function block are globals
  - persist throughout life of program
  - can be accessed/modified in any function

Divide and conquer

- Break down problem into simpler sub-problems
- Consider iteration and recursion
  - How can we implement gcd(a,b) recursively?
- Minimize transfer of state between functions
- Writing pseudocode first can help

6.087 Lecture 3 – January 13, 2010

- Review
- Blocks and Compound Statements
- Control Flow
  - Conditional Statements
  - Loops
- Functions
- Modular Programming
- Variable Scope
  - Static Variables
  - Register Variables

Programming modules in C

- C programs do not need to be monolithic
- Module: interface and implementation
  - interface: header files
  - implementation: auxilliary source/object files
- Same concept carries over to external libraries (next week. . .)

The Euclid module

- Euclid's algorithms useful in many contexts
- Would like to include functionality in many programs
- Solution: make a module for Euclid's algorithms
- Need to write header file (`.h`) and source file (`.c`)

The source: euclid.c

Implement `gcd()` in `euclid.c`:

```
/* The gcd() function */
int gcd(int a, int b) {
  while (b) { /* if a < b, performs swap */
    int temp = b;
    b = a % b;
    a = temp;
  }
  return a;
}
```

Extended Euclidean algorithm implemented as `ext_euclid()`, also in `euclid.c`

The `extern` keyword

- Need to inform other source files about functions/global variables in `euclid.c`
- For functions: put function prototypes in a header file
- For variables: re-declare the global variable using the `extern` keyword in header file
- `extern` informs compiler that variable defined somewhere else
- Enables access/modifying of global variable from other source files

```
//////////////////// myheader.h  ////////////////////
extern int myvar;

//////////////////// myvar.c  ////////////////////
int myvar = 42;

////////////////////  main.c  ///////////////////
#include <stdio.h>
#include "myheader.h"

int main() {
    printf("The value of myvar is %d\n", myvar);
    return 0;
}


////////////////////  console  ///////////////////
$ gcc -c main.c
$ gcc -c myvar.c
$ gcc -o myprogram main.o myvar.o
$ ./myprogram
The value of myvar is 42
```

extern : informs the compiler to expect the def. to be provided at some point in linking phase.

The header: euclid.h

Header contains prototypes for `gcd()` and `ext_euclid()`:

```
/* ensure included only once */
#ifndef __EUCLID_H__
#define __EUCLID_H__

/* global variables (declared in euclid.c) */
extern int x, y;
```

→ accessing variables in euclid.c

```
/* compute gcd */
int gcd(int a, int b);

/* compute g = gcd(a,b) and solve ax+by=g */
int ext_euclid(int a, int b);

#endif
```

Using the Euclid module

- Want to be able to call `gcd()` or `ext_euclid()` from the main file `diophant.c`
- Need to include the header file `euclid.h`:
  `#include "euclid.h"` (file in ".", not search path)
- Then, can call as any other function:

```
/* compute g = gcd(a,b) */
g = gcd(a,b);

/* compute x and y using Extended Euclidean alg. */
g = ext_euclid(a,b);
```

- Results in global variables `x` and `y`

```
/* rescale so ax+by = c */
grow = c/g;
x *= grow;
y *= grow;
```

Compiling with the Euclid module

- Just compiling `diophant.c` is insufficient
- The functions `gcd()` and `ext_euclid()` are defined in `euclid.c`; this source file needs to be compiled, too
- When compiling the source files, the outputs need to be linked together into a single output
- One call to `gcc` can accomplish all this:

```
athena%[1] gcc -g -O0 -Wall diophant.c
euclid.c -o diophant.o
```

→ order DOESN'T matter.

- `diophant.o` can be run as usual

[1] Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

6.087 Lecture 3 – January 13, 2010

- Review
- Blocks and Compound Statements
- Control Flow
  - Conditional Statements
  - Loops
- Functions
- Modular Programming
- Variable Scope
  - Static Variables
  - Register Variables

Variable scope

- *scope* – the region in which a variable is valid
- Many cases, corresponds to block with variable's declaration
- Variables declared outside of a function have global scope
- Function definitions also have scope

An example

What is the scope of each variable in this example?

```
int nmax = 20;  G

/* The main() function */
int main(int argc, char ** argv) /* entry point */
{
  int a = 0, b = 1, c, n;  local
  printf("%3d: %d\n",1,a);
  printf("%3d: %d\n",2,b);
  for (n = 3; n <= nmax; n++) {  only in this {
    c = a + b; a = b; b = c;                    }.
    printf("%3d: %d\n",n,c);
  }
  return 0; /* success */
}
```

Scope and nested declarations

How many lines are printed now?

```
int nmax = 20;

/* The main() function */
int main(int argc, char ** argv) /* entry point */
{
  int a = 0, b = 1, c, n, nmax = 25;
  printf("%3d: %d\n",1,a);
  printf("%3d: %d\n",2,b);
  for (n = 3; n <= nmax; n++) {
    c = a + b; a = b; b = c;
    printf("%3d: %d\n",n,c);
  }
  return 0; /* success */
}
```

Static variables

- `static` keyword has two meanings, depending on where the static variable is declared
- Outside a function, `static` variables/functions only visible within that file, not globally (cannot be `extern`'ed)
- Inside a function, `static` variables:
  - are still local to that function
  - are initialized only during program initialization
  - do not get reinitialized with each function call

```
static int somePersistentVar = 0;
```

```c
#include <stdio.h>

void increment()
{
  static int x = 0; // static variable
  x++;
  printf("%d\n", x);
}

int main()
{
  increment(); // Output: 1
  increment(); // Output: 2
  increment(); // Output: 3
  return 0;
}
```

Static Variable

- scope → local
- its initialized to $x=0$ only at the first func. call.
- DOES NOT get re-initialized for every func. call

```c
#include <stdio.h>

static int multiply(int a, int b)
{ // static function
  return a * b;
}

int main()
{
  int result = multiply(2, 3);
  printf("%d\n", result); // Output: 6
  return 0;
}
```

Static Function.

- scope is limited to that particular source file.
- can't be extern-ed.

Register variables

- During execution, data processed in *registers*
- Explicitly store commonly used data in registers – minimize load/store overhead
- Can explicitly declare certain variables as registers using `register` keyword
  - must be a simple type (implementation-dependent)
  - only local variables and function arguments eligible
  - excess/unallowed register declarations ignored, compiled as regular variables
- Registers do not reside in addressed memory; pointer of a register variable illegal

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

#define N 1000000000

int main() {

    double elapsed_time_no_reg = 0.0;
    double elapsed_time_with_reg = 0.0;

    for (int k = 0; k < 100; k++) {

        long int sum = 0;
        clock_t start = clock();

        for (long int j = 0; j < N; j++) {
            sum += j;
        }

        clock_t end = clock();
        elapsed_time_no_reg += (double)(end - start) / CLOCKS_PER_SEC;
        //printf("Sum without register keyword: %ld\n", sum);

        register long int sumr = 0;
        start = clock();

        for (register long int i = 0; i < N; i++) {
            sumr += i;
        }

        end = clock();
        elapsed_time_with_reg += (double)(end - start) / CLOCKS_PER_SEC;
        //printf("Sum with register keyword: %ld\n", sumr);
    }

    double avg_time_no_reg = elapsed_time_no_reg / 100.0;
    double avg_time_with_reg = elapsed_time_with_reg / 100.0;

    printf("Elapsed time without register keyword: %f seconds\n", avg_time_no_reg);
    printf("Elapsed time with register keyword: %f seconds\n", avg_time_with_reg);

    return 0;
}
```

```
Sum without register keyword: 499999999500000000
Sum with register keyword: 499999999500000000
Sum without register keyword: 499999999500000000
Sum with register keyword: 499999999500000000
Sum without register keyword: 499999999500000000
Sum with register keyword: 499999999500000000
Sum without register keyword: 499999999500000000
Sum with register keyword: 499999999500000000
Sum without register keyword: 499999999500000000
Sum with register keyword: 499999999500000000
Sum without register keyword: 499999999500000000
Sum with register keyword: 499999999500000000
Sum without register keyword: 499999999500000000
Sum with register keyword: 499999999500000000
Sum without register keyword: 499999999500000000
Sum with register keyword: 499999999500000000
Sum without register keyword: 499999999500000000
Sum with register keyword: 499999999500000000
Sum without register keyword: 499999999500000000
Sum with register keyword: 499999999500000000
Elapsed time without register keyword: 2.176025 seconds
Elapsed time with register keyword: 2.168012 seconds
```

This is NOT always the case.

time without 'register' < time with 'register'

Example

Variable scope example, revisited, with `register` variables:

```
/* The main() function */
int main(register int argc, register char ** argv)
{
  register int a = 0, b = 1, c, n, nmax = 20;
  printf("%3d: %d\n",1,a);
  printf("%3d: %d\n",2,b);
  for (n = 3; n <= nmax; n++) {
    c = a + b; a = b; b = c;
    printf("%3d: %d\n",n,c);
  }
  return 0; /* success */
}
```

Summary

Topics covered:

- Controlling program flow using conditional statements and loops
- Dividing a complex program into many simpler sub-programs using functions and modular programming techniques
- Variable scope rules and `extern`, `static`, and `register` variables

6.087 Practical Programming in C
January (IAP) 2010