NWEN 241
C Fundamentals

Winston Seah

School of Engineering and Computer Science
Victoria University of Wellington

This Lecture

- Background about C
- Development environment & C program structures
- GNU C complier (gcc) and GNU debugger (gdb)

Program Structure

- Another example (multiple functions)

```
/* Program to calculate the area of a circle */

#include <stdio.h>                    /* library file access */
#define PI 3.1415926      /* macro definition - symbolic constant */

float sq(float);          /* square function - function prototype */

int main(void)                        /* function heading  */
{
   float radius, area;                /* variable declarations */

   printf("Radius = ");               /* output statement (prompt)*/
   scanf("%f", &radius);              /* input statement */

   area = PI * sq(radius);            /* use square function */
   printf("Area = %f\n", area);       /* output statement */
   return 0;                          /* return statement */
}

float sq(float r)
{  return (r * r);          /* square function - function definition*/
}
```

GNU C Compiler (gcc)

- gcc does:
  - preprocessing,
  - compilation,
  - assembly, and
  - linking
- Normally all done together, but you can get gcc to stop after each stage.

```
% gcc circle.c /* default output name a.out */
```

or

```
% gcc -o circle circle.c
```

Preprocessing

- Execute preprocessor directives
- Preprocessor directives begin with a #
- Text substitution - macro substitution, conditional compilation and inclusion of named files

```
#define PI 3.14
```

  - PI will be replaced by 3.14

```
#define SQ(x) ((x) * (x))
```

  - SQ(x) will be replaced by ((x)*(x))

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

  - File stdio.h will be copied

Preprocessing

- To make gcc stop after preprocessing, use –E

```
% gcc -E circle.c
```

  - Output goes to standard output

```
% gcc -E -o circle.i circle.c
```

  - Output goes to circle.i
  - .c files become .i files.
- Does Java support preprocessing?
  - Java does not have a preprocessor
  - No header files
  - Constant data members used in place of #define

Compilation

- Compile, but don't assemble.
- Output from this stage is assembler code (symbolic representation of the numeric machine code).
- To make gcc stop after compilation, use -S.

```
% gcc -S circle.i
```

  - Output goes to circle.s

```
% gcc -S -o circleC.s circle.c
```

  - Output goes to circleC.s
  - .c and .i files become .s files.

Assembly

- Assemble, but don't link.
- Output from this stage is object code.
- To make gcc stop after assembly, use -c.

```
% gcc -c circle.s
```

  - Output goes to circle.o

```
% gcc -c circle.c -o circleC.o
```

  - Output goes to circleC.o
  - .c, .i and .s files become .o files.

Linking

- Link, and produce executable.
  - Bring together multiple pieces of object code and arrange them into one executable.

```
% gcc circle.o -o circle
% ./circle
```

Linking

- Another example (source code in multiple files)

```
% gcc -c circlelink.c sq.c
```

  – Output goes to circlelink.o and sq.o

```
% gcc -o circle circlelink.o sq.o
% ./circle
```

  Or,

```
% gcc circlelink.o sq.o
% ./a.out
```

  Think about… go and try…

```
% gcc circlelink.o
% gcc sq.o
```

GNU Debugger (gdb)

- gdb is used to fix program errors.
- gdb allows a programmer to:
  - observe the execution of a program
  - determine when and if specific lines of code are executed
  - step through a program line by line

GNU Debugger (gdb)

- How gdb works:

```
% gcc -g circle.c
```

  - -g tells gcc we are going to debug a.out
  - circle.c is compiled without optimisation (rearrangement of code)
  - a symbol table is created to store additional information (e.g., variables used)

```
% gdb a.out
```

  - Shell prompt (%) → debugger prompt ((gdb))

GNU Debugger (gdb)

- Useful gdb commands:
  - run (start to execute the program)
  - q/quit (exit the debugger)
  - break 10 (stop at line 10)
  - print x (show variable x)
  - display x (show variable x when the program is paused)
  - step (step through the program line by line)
  - next (execute next line)
  - continue (resume the execution until next breakpoint)
  - help

GNU Debugger (gdb)

- An example (crash)

```
int main(void)
{ int x, y;
  y = 1234;
  for (x = 5; x>=0; x--)
    y = y/x;                    /* crash occurs here */
  printf("%d\n", y);
  return 0;
}
```

```
(gdb) run
```

  - You will see SIGFPE sent to the program (erroneous arithmetic operation)

```
(gdb) print x
```

  - You will see x=0 (denominator cannot be “0”)

Data Types

- Programming is about describing data and algorithms
- How data is represented in memory?

Data Types

- Programming is about describing data and algorithms
- How data is represented in memory?
- Four basic data types:
  - int (integer quantity)
  - char (single character)
  - float (floating-point number)
  - double (double-precision floating-point number)

  Note: There are also qualifiers associated with the types: short / long, and signed / unsigned.
- Data types for Java (any difference?)

Data Types

- Two groups of types
  - Integral types: int and char
    - Can be used to hold integer values
  - Floating types: float and double
    - Can be used to hold real values

Data Types

- Integral types

```
int i;
char c;
for (i = 65; i <= 90; i++)     /* 'A'=65, 'Z'=90 */
  printf("%c ", i);      /* what is i? */
                         /* what is printed? */

for (c = 'A'; c <= 'Z'; c++)   /* 'A'=65, 'Z'=90 */
  printf("%d ", c);      /* what is c? */
                         /* what is printed? */
```

Data Types

- Integral types

```
int i;
char c;
for (i = 65; i <= 90; i++)     /* 'A'=65, 'Z'=90 */
  printf("%c ", i);      /* print an int into a char */
                         /* A B C … Z is printed */

for (c = 'A'; c <= 'Z'; c++)  /* 'A'=65, 'Z'=90 */
  printf("%d ", c);      /* print a char into an int */
                         /* 65 66 67 … 90 is printed */
```

Data Types

- Floating types
  - How floating-point number represented in memory
  - 123.45 = 1111011.01110011 = 0.111101101110011 * 2^7
    - Mantissa: 111101101110011
    - Exponent: 7
    - Mantissa and exponent are stored separately
  - 123.75 = 1111011.11000000 = 0.111101111000000 * 2^7
  - 123.45 cannot be perfectly expressed in binary notation

Data Types

- Floating types
  - How floating-point number represented in memory
  - 123.45 = 1111011.01110011 = 0.11110110111001 1 * 2^7
    - Mantissa: 11110110111001 1
    - Exponent: 7
    - Mantissa and exponent are stored separately
  - 123.75 = 1111011.11000000 = 0.11110111100000 0 * 2^7
  - 123.45 cannot be perfectly expressed in binary notation
    - float t = 123.45
    - t = 123.449997
    - Use double

Data Types

- Sizes of different types
  - Use sizeof() to find out
  - The sizes may vary from machine to machine
  - The following rules are always guaranteed:
    - sizeof(char) = 1
    - sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long)
    - sizeof(signed) = sizeof(unsigned) = sizeof(int)
    - sizeof(float) <= sizeof(double) <= sizeof(long double)
  - Does Java have varied sizes between systems?

Data Types

- Type casting
  - C does automatic type casting

```
int i = 2;
double d = 2.5;
i = (int)d;  /* explicit type casting */

i = d;
```

Data Types

- Type casting
  - C does automatic type casting

```
int i = 2;
double d = 2.5;
i = (int)d;  /* explicit type casting */

i = d;       /* d is converted to an int
              * and then assigned to i.
              */
```

  - Info losing type casting must be made explicitly in Java

Data Types

- Constants
  - integer constants
  - floating-point constants
  - character constants
  - string constants
  - enumeration constants (does Java have this?)
- Naming constants
  - Use the const qualifier (Java uses the **final** keyword)

```
const float pi = 3.14;  /* declares a "read-only" variable
  */
```

  - Use the preprocessor (Java does not have this)

```
#define PI 3.14  /* macro definition
                  * PI to be substituted by 3.14
                  */
```

Problems with macros

```
#define SQ(x) x * x
(int)SQ(r);         /* (int)r * r */
SQ(r1 + r2);        /* r1 + r2 * r1 + r2 */
```

Problems with macros

```
#define SQ(x) x * x
(int)SQ(r);          /* (int)r * r */
SQ(r1 + r2);         /* r1 + r2 * r1 + r2 */
```

– Solution: `#define SQ(x) ((x) * (x))`

Problems with macros

```
#define SQ(x) x * x
(int)SQ(r);          /* (int)r * r */
SQ(r1 + r2);         /* r1 + r2 * r1 + r2 */
```

- Solution: `#define SQ(x) ((x) * (x))`
- Is it safe now?

Problems with macros

```
#define SQ(x) x * x
(int)SQ(r);          /* (int)r * r */
SQ(r1 + r2);         /* r1 + r2 * r1 + r2 */
```

- Solution: `#define SQ(x) ((x) * (x))`
- Is it safe now?

```
SQ(++r);     /* r is incremented twice? */
SQ(f());     /* f() called twice before the
              * multiplication
              */
```

- Be careful when defining and calling macros

Data Types

- More data types later on ….