NWEN 241
C Fundamentals

Qiang Fu

School of Engineering and Computer Science
Victoria University of Wellington

Victoria
UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui
CAPITAL CITY UNIVERSITY

This Lecture

- Background about C
- A glimpse of C program structures
- GNU C complier (gcc) and GNU debugger (gdb)

Comparing C, C++, Java

- The C Family of Languages: Interview with Dennis Ritchie, Bjarne Stroustrup, and James Gosling:
  - http://www.gotw.ca/publications/c_family_interview.htm

Comparing C, C++ and Java

- C is the basis for C++ and Java
  - C evolved into C++
  - C++ transmuted into Java
  - The “class” is an extension of “struct” in C
- Similarities
  - Java uses a syntax similar to C++ (for, while, …)
  - Java supports OOP as C++ does (class, inheritance, …)
- Differences
  - Java does not support pointer
  - Java frees memory by garbage collection
  - Java is more portable by using bytecode and virtual machine
  - Java does not support operator overloading
  - ….

Background and Characteristics

- Designed by Dennis Ritchie of Bell Labs in the 1970s
- An outgrowth of B also developed at Bell Labs
- ANSI/ISO standard in early 1990s.
- Bridging the gap between machine language and high-level languages
  - Low-level features: fast/efficient (systems programming)
  - High-level features: structured programming (applications programming)

Applications

- Operating systems
- Distributed systems
- Network programming
- Database applications
- Real-time and engineering applications
- Any application where efficiency is *paramount*

Program Structure

- A C program consists of one or more *functions*
- A C program must have a `main` function

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

- Execution begins with the `main` function
- Java vs. C
  - C uses stand-alone functions
  - No stand-alone functions in Java
  - No global functions in Java

Program Structure

- Each function must contain:
  - A function *heading*, (return type, function name, an *optional* list of *arguments*)
  - A list of argument *declarations*, if arguments are included in heading
  - A *compound statement*

```
int function_name(int x, int y)
{
  ...
}
```

Program Structure

- An example (single function)

```
/* A simple program */              /* comment */

#include <stdio.h>                  /* library file access */

int main(void)                      /* function heading   */
{
   printf("Hello world\n");         /* output statement */

   return 0;                         /* return statement */
}
```

Program Structure

- An example (single function)

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

#include <stdio.h>                  /* library file access */
#define PI 3.14         /* macro definition – symbolic constant */
#define SQ(x) ((x)*(x))             /* macro with arguments */

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

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

   area = 3.14 * radius * radius; /* assignment statement */
   printf("Area1 = %f\n", area);  /* output statement */

   area = PI * SQ(radius);          /* use macros */
   printf("Area2 = %f\n", area);  /* output statement */

   return 0;                        /* return statement */
}
```

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…

```
% 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”)

Summary

- C / C++ / Java
- C program structure
- gcc
- gdb

Next Lecture

- More on C fundamentals
- We will look at data types, operators, input/output and control constructs