Introduction to C

To Be a Master Programmer

Instructor: Yin Lou

02/18/2011

Course Recap

- Types, operators and expression

Course Recap

- Types, operators and expression
  - int, unsigned int, long double, etc
  - &&, ||; &, |, >>, <<

Course Recap

- Types, operators and expression
  - int, unsigned int, long double, etc
  - &&, ||; &, |, >>, <<
- Control flow

Course Recap

- Types, operators and expression
  - int, unsigned int, long double, etc
  - &&, ||; &, |, >>, <<
- Control flow
  - if-else, switch, while, do-while, for
  - break; continue

Course Recap

- Types, operators and expression
  - int, unsigned int, long double, etc
  - &&, ||; &, |, >>, <<
- Control flow
  - if-else, switch, while, do-while, for
  - break; continue
- Functions, program structure, and project management

Course Recap

- Types, operators and expression
  - int, unsigned int, long double, etc
  - &&, ||; &, |, >>, <<
- Control flow
  - if-else, switch, while, do-while, for
  - break; continue
- Functions, program structure, and project management
  - Recursion
  - Header file
  - Makefile

Course Recap

- Types, operators and expression
  - int, unsigned int, long double, etc
  - &&, ||; &, |, >>, <<
- Control flow
  - if-else, switch, while, do-while, for
  - break; continue
- Functions, program structure, and project management
  - Recursion
  - Header file
  - Makefile
- Pointers and arrays
  - int *p = &x;
  - int a[10];
  - int *p = (int *) malloc(n * sizeof(int));
  - Array names are constant while pointers are usually variables.

Course Recap

- Complex types

Course Recap

- Complex types
  - enum, struct, union
  - typedef
  - function pointers

Course Recap

- Complex types
  - enum, struct, union
  - typedef
  - function pointers
- Preprocessing

Course Recap

- Complex types
  - enum, struct, union
  - typedef
  - function pointers
- Preprocessing
  - #include, #define, #ifndef, #endif

Course Recap

- Complex types
  - enum, struct, union
  - typedef
  - function pointers
- Preprocessing
  - #include, #define, #ifndef, #endif
- Standard I/O

Course Recap

- Complex types
  - enum, struct, union
  - typedef
  - function pointers
- Preprocessing
  - #include, #define, #ifndef, #endif
- Standard I/O
  - printf, sprintf, fprintf
  - scanf, sscanf, fscanf
  - fread, fwrite

Course Recap

- Complex types
  - enum, struct, union
  - typedef
  - function pointers
- Preprocessing
  - #include, #define, #ifndef, #endif
- Standard I/O
  - printf, sprintf, fprintf
  - scanf, sscanf, fscanf
  - fread, fwrite
- Threads

Course Recap

- Complex types
  - enum, struct, union
  - typedef
  - function pointers
- Preprocessing
  - #include, #define, #ifndef, #endif
- Standard I/O
  - printf, sprintf, fprintf
  - scanf, sscanf, fscanf
  - fread, fwrite
- Threads
  - pthread

Course Recap

- Complex types
  - enum, struct, union
  - typedef
  - function pointers
- Preprocessing
  - #include, #define, #ifndef, #endif
- Standard I/O
  - printf, sprintf, fprintf
  - scanf, sscanf, fscanf
  - fread, fwrite
- Threads
  - pthread
- All these can be done with?

Course Recap

- Complex types
  - enum, struct, union
  - typedef
  - function pointers
- Preprocessing
  - #include, #define, #ifndef, #endif
- Standard I/O
  - printf, sprintf, fprintf
  - scanf, sscanf, fscanf
  - fread, fwrite
- Threads
  - pthread
- All these can be done with?
  - The compiler: gcc

Start with Hello World

```
#include <stdio.h>

int main()
{
    printf("Hello World\n");
    return 0;
}
```

Master Programmer vs. Beginner

Can you answer the following question?

- Why do programs need to be compiled before execution?

Master Programmer vs. Beginner

Can you answer the following question?

- Why do programs need to be compiled before execution?
- What does the compiler do during the process of converting C source code to machine code and how?

Master Programmer vs. Beginner

Can you answer the following question?

- Why do programs need to be compiled before execution?
- What does the compiler do during the process of converting C source code to machine code and how?
- What does the executable code look like? What else besides machine code? How do they store and organize?

Master Programmer vs. Beginner

Can you answer the following question?

- Why do programs need to be compiled before execution?
- What does the compiler do during the process of converting C source code to machine code and how?
- What does the executable code look like? What else besides machine code? How do they store and organize?
- What does "#include <stdio.h>" mean? Why we need to include it? What is a C library and how is it implemented?

Master Programmer vs. Beginner

Can you answer the following question?

- Why do programs need to be compiled before execution?
- What does the compiler do during the process of converting C source code to machine code and how?
- What does the executable code look like? What else besides machine code? How do they store and organize?
- What does "#include <stdio.h>" mean? Why we need to include it? What is a C library and how is it implemented?
- Do codes compiled by different compilers (gcc, Microsoft VC) on different machines (x86, ARM) look the same? Why?

Master Programmer vs. Beginner

- How does our "Hello World" run? How does the operating system load it? Where does it start execution? Where does it end execution? What happens before main function? What happens after main function?

Master Programmer vs. Beginner

- How does our "Hello World" run? How does the operating system load it? Where does it start execution? Where does it end execution? What happens before main function? What happens after main function?
- If there's no operating system, can our "Hello World" run? If we need to run "Hello World" on a machine without operating system, how can we make it?

Master Programmer vs. Beginner

- How does our "Hello World" run? How does the operating system load it? Where does it start execution? Where does it end execution? What happens before main function? What happens after main function?
- If there's no operating system, can our "Hello World" run? If we need to run "Hello World" on a machine without operating system, how can we make it?
- How is "printf" implemented? Why it can have variable-length argument list? Why it can output characters to the terminal?

Master Programmer vs. Beginner

- How does our "Hello World" run? How does the operating system load it? Where does it start execution? Where does it end execution? What happens before main function? What happens after main function?
- If there's no operating system, can our "Hello World" run? If we need to run "Hello World" on a machine without operating system, how can we make it?
- How is "printf" implemented? Why it can have variable-length argument list? Why it can output characters to the terminal?
- What does "Hello World" look like in the memory?

What Happens?

```
$gcc hello.c -o hello
$./hello
```

What Happens?

```
$gcc hello.c -o hello
$./hello
```

GCC does this 4 steps automatically for you:

- Preprocessing
- Compilation
- Assembly
- Linking

Preprocessing

- Remove all #define and expand all macros
- Process all #if, #ifdef, #ifndef, #elif, #else, #endif
- Process #include recursively
- Remove all comments (“//”, “/* ... */”)
- Add line number and file name identification, for example, #2 “hello.c” 2, to help compiler output debugging information
- Retain all #program because the compiler needs them.

Preprocessing

- Remove all #define and expand all macros
- Process all #if, #ifdef, #ifndef, #elif, #else, #endif
- Process #include recursively
- Remove all comments ("//", "/* ... */")
- Add line number and file name identification, for example, #2 "hello.c" 2, to help compiler output debugging information
- Retain all #program because the compiler needs them.

```
$gcc -E hello.c -o hello.i
```

Compilation

- Scanner: Lexical analysis
- Parser: Syntactical analysis
- Semantic analysis
- Optimization

Compilation

- Scanner: Lexical analysis
- Parser: Syntactical analysis
- Semantic analysis
- Optimization

```
$gcc -S hello.i -o hello.s
```

Assembly

- Convert assembly code to machine code

Assembly

- Convert assembly code to machine code

```
$as hello.s -o hello.o
or
$gcc -c hello.s -o hello.o
```

Linking

- Why do we need linking?
- Why doesn't the assembler output an executable file rather than a object file?
- What happens when linking?

Linking

- Why do we need linking?
- Why doesn't the assembler output an executable file rather than a object file?
- What happens when linking?

```
$ld -static /usr/lib/crt1.o /usr/lib/crti.o
/usr/lib/gcc/i486-linux-gnu/4.1.3/crtbeginT.o
-L/usr/lib/gcc/i486-linux-gnu/4.1.3 -L/usr/lib -L/lib hello.o --start-group
-lgcc -lgcc_eh -lc --end-group /usr/lib/gcc/i486-linux-gnu/4.1.3/crtend.o
/usr/lib/crtn.o
```

Static Linking

- Routines, external functions and variables which are resolved in a caller at compile-time
- Advantages

Static Linking

- Routines, external functions and variables which are resolved in a caller at compile-time
- Advantages
  - Application can be certain that all its libraries are present and that they are the correct version.
  - No dependency problem.
  - In some cases, static linking can have performance gain.
  - Allows the application to be contained in a single executable file, simplifying distribution and installation.

Example

a.c

```
extern int shared;

int main()
{
    int a = 100;
    swap(&a, &shared);
}
```

b.c

```
int shared = 1;

void swap(int *a, int *b)
{
    *a ^= *b ^= *a ^= *b;
}
```

Example

a.c

```
extern int shared;

int main()
{
    int a = 100;
    swap(&a, &shared);
}
```

b.c

```
int shared = 1;

void swap(int *a, int *b)
{
    *a ^= *b ^= *a ^= *b;
}
```

```
$gcc -c a.c b.c
```

Using ld

```
$ld a.o b.o -e main -o ab
```

Using ld

```
$ld a.o b.o -e main -o ab
```

- -e main: Use “main” as the main point of execution
- -o ab: output a file named “ab”

Using ld

```
$ld a.o b.o -e main -o ab
```

- -e main: Use "main" as the main point of execution
- -o ab: output a file named "ab"

```
$objdump -h a.o
$objdump -h b.o
$objdump -h ab
```

Dynamic Linking

- Loading the subroutines of a library at load time or runtime
- Advantage

Dynamic Linking

- Loading the subroutines of a library at load time or runtime
- Advantage
  - Shared code. Only one copy of the code
  - Can change library (for example, upgrade the latest library) without recompiling the code
- But you need to set LD_LIBRARY_PATH manually.

Example

program1.c

```
#include "myprinting.h"

int main()
{
    foobar(1);
    return 0;
}
```

program2.c

```
#include "myprinting.h"

int main()
{
    foobar(2);
    return 0;
}
```

Example

myprinting.c

```
#include <stdio.h>

void foobar(int i)
{
    printf("Printing from lib.so %d\n", i);
}
```

myprinting.h

```
#ifndef __MYPRINTING_H
#define __MYPRINTING_H

void foobar(int i);

#endif
```

Example

myprinting.c

```
#include <stdio.h>

void foobar(int i)
{
    printf("Printing from lib.so %d\n", i);
}
```

myprinting.h

```
#ifndef __MYPRINTING_H
#define __MYPRINTING_H

void foobar(int i);

#endif
```

```
$gcc -fPIC -shared -o libmyprinting.so myprinting.c
```

What Happens?

- -shared: output a shared object
- -fPIC: output a platform-independent code

What Happens?

- -shared: output a shared object
- -fPIC: output a platform-independent code

```
$gcc -o program1 program1.c ./libmyprinting.so
$gcc -o program2 program2.c ./libmyprinting.so
```

More GCC Options

Makefile

```
CC:=gcc
OPTIONS:=-shared -fPIC -O3 -funroll-loops
INCLUDE_PATH:=-I./lib/imagelib -I./lib/matrix
LIB_PATH:=-L./lib

PROJECT_LIBS:=-limage.x64 -ljpeg -lgfortran

SRC_DIR:=src
DST_DIR:=bin

default:
    $(CC) $(OPTIONS) $(INCLUDE_PATH) $(LIB_PATH) \
    $(SRC_DIR)/*.c -o $(DST_DIR)/libmyproject.so $(PROJECT_LIBS)
```