Introduction

Week 01 – Lecture 1

C Language

Team

Instructors

Giancarlo Succi

Joseph Brown

Teaching Assistants

Vladimir Ivanov

Stanislav Litvinov

Alexey Reznik

Munir Makhmutov

Hamna Aslam

Sources

- These slides have been adapted from the original slides of the adopted book:
  - Tanenbaum & Bo, Modern Operating Systems: 4th edition, 2013
    Prentice-Hall, Inc.

  and customised for the needs of this course.
- Additional input for the slides are detailed later

The C Language (1)

- Operating systems are normally large C (or sometimes C++) programs consisting of many pieces written by many programmers
- It is important to know some of the key differences between C and languages like Python and especially Java
- The primitive data types in C are:
  - integers (including short and long ones)
  - characters
  - floating-point numbers

The C Language (2)

- Composite data types can be constructed using:
  - arrays
  - structures
  - unions
- The control statements in C are the next statements (similar to Java):
  - if
  - switch
  - for
  - while

The C Language (3)

- Pointer is the one feature of C that Java and Python do not have
- A **pointer** is a variable that points to (i.e., contains the address of) a variable or data structure
- Consider the following example:

```
char c1, c2, *p;
c1 = 'c';
p = &c1;
c2 = *p;
```

The C Language (3)

- *c1* and *c2* are character variables
- *p* is a variable that points to (i.e., contains the address of) a character
- The first assignment (c1 = ’c’;) stores the ASCII code for the character “c” in the variable *c1*
- The second one (p = &c1;) assigns **the address of *c1*** to the **pointer variable *p***
- The third one (c2 = *p;) assigns the contents of the variable pointed to by *p* to the variable *c2*, so after these statements are executed, c2 also contains the ASCII code for “c”

The C Language (4)

Address	Type	Value	Name
0x29ff10	end address		
0x29ff2a	char	-3 '\375'	c1
0x29ff2b	char	127 '\177'	c2
0x29ff2c	char *	0x400080 "PE"	p
0x29ff38	start address		

Stack after execution of

char c1, c2, *p;

The C Language (5)

Address	Type	Value	Name
0x29ff10	end address		
0x29ff2a	char	99 'c'	c1
0x29ff2b	char	127 '\177'	c2
0x29ff2c	char *	0x400080 "PE"	p
0x29ff38	start address		

Stack after execution of

c1 = ’c’;

The C Language (6)

Address	Type	Value	Name
0x29ff10	end address		
0x29ff2a	char	99 'c'	c1
0x29ff2b	char	127 '\177'	c2
0x29ff2c	char *	0x29ff2a "c\177*\377)"	p
0x29ff38	start address		

Stack after execution of

p = &c1;

The C Language (7)

Address	Type	Value	Name
0x29ff10	end address		
0x29ff2a	char	99 'c'	c1
0x29ff2b	char	99 'c'	c2
0x29ff2c	char *	0x29ff2a "cc*\377)"	p
0x29ff38	start address		

Stack after execution of

c2 = *p;

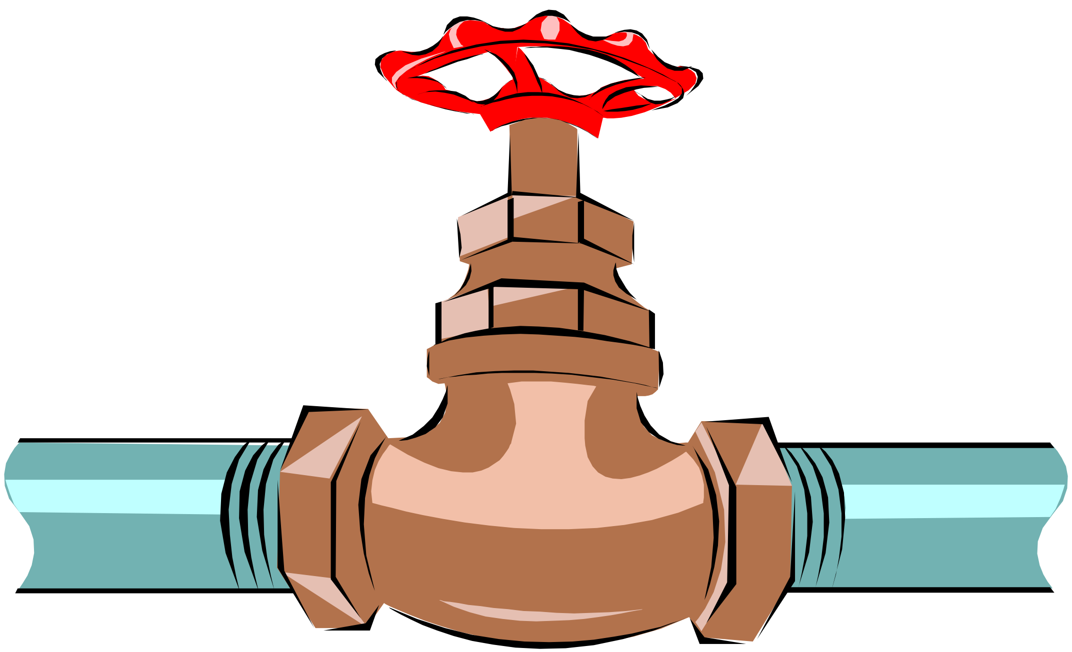
The C Language (8)

- Some things that C **does not have:**
  - built-in strings
  - threads
  - packages
  - classes
  - objects
  - type safety
  - garbage collection

The C Language (9)

- All storage in C is either static or explicitly allocated and released by the programmer, usually with the library functions *malloc()* and *free()*
- Programmer have full control over memory
- Along with explicit pointers it makes C attractive for writing OSs
- OSs are basically real-time systems. When an interrupt occurs, the operating system may have only a few microseconds to perform some action or lose critical information
- Having the garbage collector kick in at an arbitrary moment is intolerable

Preprocessor directives (1)



Preprocessor directives are special commands which are evaluated before compilation

Preprocessor directives (2)

- Preprocessor directives start with a hash symbol ("#")
- They must be written at the beginning of the line, they are not an instruction of the language
- A semi colon (;) is NOT required at the end
- If the entire command does not fit one line, you have to inform the preprocessor that the command continues on the next line by adding a backslash (\) at the end of the line

Preprocessor directives (3)

- Reasons to use preprocessing
  - Source code can be made clearer (using #define)
  - Source code can be split into multiple modules easily (using #include)
  - Sections of code can be shared by several different programs (using #include)
  - It can reduce the work of the compiler by automatically excluding unnecessary sections of code (using #if) or replacing symbolic names with real values (using #define)

#include (1)

- It allows to separate the code into separate units in order to modularize the code. This allows to:
  - Develop the code separately
  - "Divide and conquer" approach: clearly separate the problem into smaller parts and solve the sub-problems (Reduction of complexity)
  - Modules can be tested and verified separately
  - Single modules can be exchanged and extended
- This requires clear interfaces between the different modules. This is the aim in using header files.
- Including the .h file means to include the interface of the file, which used to be the unit of modularization.

#include (2)

- Format:

```
#include <myclass.h>
#include "myClass.h"
```

- The filename has to be written in double quotes (**"xxx"**) if it is in the same directory or in the user-specified include path
- The filename has to be written in angle brackets (**<xxx>**) if it is in the system include path
- The include statement is replaced with the content of the given header file

#include (3)

Before preprocessing	After preprocessing

```c
#include <stdio.h>
int main()
{
   int a, b;
   int s = a + b
   return 0;
}
```

Content of file
"stdio.h"

```c
int main()
{
   int a, b;
   int s = a + b
   return 0;
}
```

#include before and after preprocessing

#define (1)

- Why use #define in C?
- To define constants in C
  - To make the program easier to read
  - If you use a variable throughout the program and then decide to change it, the #define directives eliminate the need to manually change each statement and therefore help to avoid errors
- Why not use public variables instead of #define?
  - The variable could be inadvertently changed during the execution of the program
  - The compiler can generate faster and more compact code for constants that for variables

#define (2)

- Format:

```
#define identifier token-string
```

- Example:

```
#define one 1
```

- The #define directive substitutes a given identifier with the given token-string
- The token-string consists of a series of tokens, such as keywords, constants, or complete statements

#define (3)

Before preprocessing	After preprocessing

```
#define one 1
int main()
{
  int a, b;
  // read a and b
  ...
  int s = a + b + one ;
  return 0;
}
```

```
int main()
{
  int a, b;
  ...
  int s = a + b + 1 ;
  return 0;
}
```

#define before and after preprocessing

#define (4)

Before preprocessing	After preprocessing

Before preprocessing:

```c
# define ABC 123
int main()
{
  printf(“ABC%d\n", ABC);
}
```

After preprocessing:

```c
int main()
{
  printf("ABC%d\n", 123);
}
```

Result: ABC123

#define - Text literals (again constants)

#define (5)

Before preprocessing	After preprocessing

```
# define ABC 123
int ABC;
int main()
{
  printf(“ABC%d\n", ABC);
}
```

```
int 123;
int main()
{
  printf("ABC%d\n", 123);
}
```

Compiler error

#define - Text literals (again constants)

#define (6)

```
#define BEGIN {
#define END }
using namespace std;
int main()
BEGIN
   printf("ABC");
END
```

#define – dangerous step: altering the syntax of C++

#define (7)

- #define – macros as kind of functions
- Format:

```
#define identifier(identifier, ...)
token-string
```

- Example:

```
#define plus(x, y) x+y
```

- In this case a function-like macro is created where identifier plus parameters are replaced in source code
- In the given example, this means that whenever the preprocessor finds the macro *plus* with two parameters, it replaces it with *x+y*

#define (8)

- Why use #define macros ?
  - It is more efficient from a memory usage perspective because the macro is only stored once in the executable even if it appears many times in the code
  - #define macros can be used to make the code easier to read
  - Arguments may be of any data type
  - Avoids overhead of a function call

#define (9)

Before preprocessing	After preprocessing

```
#define double(x) 2*x
int main()
{
   int a;
   // read a
   ...
   int s = double(a);
   return 0;
}
```

```
int main()
{
   int a;
   ...
   int s = 2*a;
   return 0;
}
```

#define – function like macros

#define (10)

Before preprocessing	After preprocessing

```
#define plus(x,y) x+y
int main()
{
  int a, b;
  // read a and b
  ...
  int s = plus(a,b);
  return 0;
}
```

```
int main()
{
  int a, b;
  ...
  int s = a + b;
  return 0;
}
```

#define – function like macros, but…

#define (11)

- Macros are inserted into the code by purely replacing the macro body with the defined keyword
- No interpretation occurs
- This leads to several problems as we will see...

#define (12)

Before preprocessing	After preprocessing

```
#define plus(x,y) x+y
int main()
{
  int a, b;
 // read a and b
  ...
  int s = plus(a,b)*2;
return 0;
}
```

```
int main()
{
  int a, b;
  ...
  int s = a+b*2;
  return 0;
}
```

Which does not match the intention…

Better use:

#define plus(x,y) (x+y)

but…

#define – Pitfalls (1)

#define (13)

Before preprocessing	After preprocessing

```
#define times(x,y)  (x*y)
int main()
{
  int a, b;
 // read a and b
  ...
  int s = times(a+5,b-2);
  cout << "Result: " << s <<
    endl;
  return 0;
}
```

```
int main()
{
  int a, b;
  ...
  int s = (a+5*b-2);
  return 0;
}
```

Which does not match the intention...

Better use:

#define times(x,y) ((x)*(y))

but...

#define – Pitfalls (2)

#define (14)

Before preprocessing	After preprocessing

```
#define double(x)  ((x)+(x))
int main()
{
  int a,b;
  b = 2;
  a = double(b++);
  return 0;
}
```

```
int main()
{
  int a,b;
  b = 2;
  a = ((b++)+(b++))
  return 0;
}
```

returns 2 and *b* is increased to 3

returns 3 and *b* is increased to 4

At the end
a=5 and *b=4*

Which does not match the intention…

#define – Pitfalls (3)

#define (15)

Before preprocessing	After preprocessing

```
#define double(x)  ((x)+(x))
int main()
{
   int a,b;
   b = 2;
   a = double(++b);
   return 0;
}
```

```
int main()
{
   int a,b;
   b = 2;
   a = ((++b)+(++b))
   return 0;
}
```

b is increased to 3 and returns 3

b is increased to 4 and returns 4

At the end
a=7 and b=4

Which does not match the intention…

#define – Pitfalls (4)

Header Files (1)

- While *.c* files contain code of OS, *.h* header files contain declarations and definitions used by one or more code files
- They can also include **macros**, for example,

```
#define BUFFER_SIZE 4096
```

- It allows programmer to name the constant. BUFFER_SIZE is replaced by 4096 everywhere in the code during compilation

Header Files (2)

- Macros can have parameters. For example,

```
#define max(a, b) (a > b ? a : b)
```

- It allows programmer to write

```
i = max(j, k+1)
```

and get

```
i = (j > k+1 ? j : k+1)
```

Header Files (3)

- Headers can also contain conditional compilation:

```
#ifdef X86
intel_int_ack();
#endif
```

- it compiles into a call to the function intel_int_ack **only if the macro X86 is defined**
- Conditional compilation is used to isolate architecture-dependent code. It ensures that certain code is inserted only when the system is compiled on the X86, other code is inserted only when the system is compiled on a SPARC, and so on

Header Files (4)

- A .c file can include zero or more header files using the #include directive.
- There are many header files that are common to nearly every .c and are stored in a central directory

Large Programming Projects (1)

- To build the system, each .c file is compiled into an object file by the C compiler
- Object files, which have the suffix .o, contain binary instructions for the target machine
- They will later be directly executed by the CPU
- There is nothing like Java byte code or Python byte code in the C world

Large Programming Projects (2)

- The first pass of the C compiler is called the **C preprocessor**
  - It reads each .c file
  - Every time it hits a #include directive, it goes and gets the header file named in it and processes it:
    - expands macros
    - handles conditional compilation
    - passes the results to the next pass of the compiler as if they were physically included

Large Programming Projects (3)

- Having to recompile the entire code base every time one file is changed would be unbearable
- However, changing a key header file that is included in thousands of other files does require recompiling those files
- It is possible to keep track of which files need to be recompiled and which don't
- On UNIX systems, there is a program called *make* that reads the *Makefile* – special file that tells which files are dependent on which other files

Large Programming Projects (3)

- *make* sees which object files are needed to build the OS binary
- For each file it checks if any of the files it depends on have been modified subsequent to the last time the object file was created
- If some of the files were modified, that object file has to be recompiled
- When *make* has determined which *.c* files have to recompiled, it invokes the C compiler to recompile them
- In large projects, creating the *Makefile* is error prone, so there are tools that do it automatically

Large Programming Projects (4)

- Once all the .o files are ready, they are passed to a program called **the linker** to combine all of them into a **single executable binary file:**
  - any library functions called are included
  - interfunction references are resolved
  - machine addresses are relocated as need be
- When the linker is finished, the result is an executable program, traditionally called *a.out* on UNIX systems (Fig. 1-30)

Large Programming Projects (5)

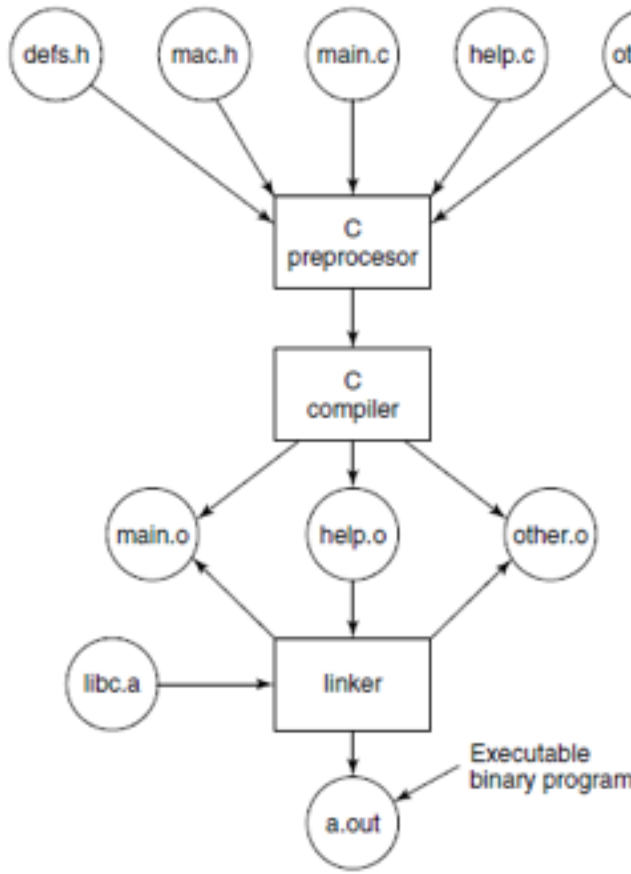


Figure 1-30. The process of compiling C and header files to make an executable.

The Model of Run Time (1)

- Once the OS binary has been linked, the computer can be rebooted and the new operating system started
- It may then dynamically load pieces that were not statically included in the binary such as device drivers and file systems
- At run time the operating system may consist of multiple segments:
  - the text (the program code): this segment is normally immutable, not changing during execution
  - the data: it starts out at a certain size and initialized with certain values, but it can change and grow as need be
  - the stack: is initially empty but grows and shrinks as functions are called and returned from

The Model of Run Time (2)

- Often the text segment is placed near the bottom of memory, the data segment just above it, with the ability to grow upward, and the stack segment at a high virtual address, with the ability to grow downward, but different systems work differently
- In all cases, the OS code is directly executed by the hardware, with no interpreter and no just-in-time compilation, as it is normal with Java

Metric Units

Exp.	Explicit	Prefix	Exp.	Explicit	Prefix
10^{-3}	0.001	milli	10^{3}	1,000	Kilo
10^{-6}	0.000001	micro	10^{6}	1,000,000	Mega
10^{-9}	0.000000001	nano	10^{9}	1,000,000,000	Giga
10^{-12}	0.000000000001	pico	10^{12}	1,000,000,000,000	Tera
10^{-15}	0.000000000000001	femto	10^{15}	1,000,000,000,000,000	Peta
10^{-18}	0.000000000000000001	atto	10^{18}	1,000,000,000,000,000,000	Exa
10^{-21}	0.000000000000000000001	zepto	10^{21}	1,000,000,000,000,000,000,000	Zetta
10^{-24}	0.000000000000000000000001	yocto	10^{24}	1,000,000,000,000,000,000,000,000	Yotta

Figure 1-31. The principal metric prefixes.

End

Week 01 – Lecture 1

References

- Tanenbaum & Bo, Modern Operating Systems: 4th edition, 2013 Prentice-Hall, Inc.