CMSC 246 Systems Programming

Spring 2018
Bryn Mawr College
Instructor: Deepak Kumar

CMSC 246 Systems Programming
Go to class web page...

Goals

• Learn Linux (CLI, not WIMP!)
• Learn C
• Learn Linux tools
Evolution of C

Algol60
Designed by an international committee, 1960

CPL
Combined Programming Language
Cambridge & Univ. of London, 1963
Was an attempt to bring Algol down to earth and retail contact with the realities of an actual computer.
Features:
• Big
• Too many features
• Hard to learn
• Intended for numerical as well as non-numerical applications

BCPL
Basic CPL
Designed by Martin Richards, Cambridge 1967
Intended as a tool for writing compilers.
Designed to allow for separate compilation.
Features:
• Typeless language (only binary words)
• Introduced static variables
• Compact code
• Provides access to address of data objects
• Stream-based I/O

B
Designed by Ken Thompson, Bell Labs 1970
A true forerunner of C
Features:
• Typeless (with floating pt. capabilities
• Designed for separate compilation
• Easily implementable
• Pre-processor facility
• Expensive library
# Evolution of C

<table>
<thead>
<tr>
<th>Evolution</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Algol60</strong></td>
<td>Designed by an international committee, 1960</td>
</tr>
<tr>
<td><strong>CPL</strong></td>
<td>Combined Programming Language Cambridge &amp; Univ. of London, 1963&lt;br&gt;Was an attempt to bring Algol down to earth and retain contact with the realities of an actual computer. Features:&lt;br&gt;• Big&lt;br&gt;• Too many features&lt;br&gt;• Hard to learn&lt;br&gt;• Intended for numerical as well as non-numerical applications</td>
</tr>
<tr>
<td><strong>BCPL</strong></td>
<td>Basic CPL&lt;br&gt;Designed by Martin Richards, Cambridge 1967&lt;br&gt;Intended as a tool for writing compilers.&lt;br&gt;Features:&lt;br&gt;• Typeless language (only binary words)&lt;br&gt;• Introduced static variables&lt;br&gt;• Compact code&lt;br&gt;• Provides access to address of data objects&lt;br&gt;• Stream-based I/O</td>
</tr>
<tr>
<td><strong>B</strong></td>
<td>Designed by Ken Thompson, Bell Labs 1970&lt;br&gt;A true forerunner of C&lt;br&gt;Features:&lt;br&gt;• Typeless (with floating pt. capabilities)&lt;br&gt;• Designed for separate compilation&lt;br&gt;• Easily implementable&lt;br&gt;• Pre-processor facility&lt;br&gt;• Expensive library</td>
</tr>
<tr>
<td><strong>C</strong></td>
<td>1971-72&lt;br&gt;Developed at Bell Laboratories by Ken Thompson, Dennis Ritchie, and others. C is a by-product of UNIX.&lt;br&gt;Ritchie began to develop an extended version of B. He called his language NB (“New B”) at first. As the language began to diverge more from B, he changed its name to C. The language was stable enough by 1973 that UNIX could be rewritten in C.</td>
</tr>
<tr>
<td><strong>K&amp;R C</strong></td>
<td>Described in Kernighan and Ritchie, The C Programming Language (1978)&lt;br&gt;De facto standard&lt;br&gt;Features:&lt;br&gt;• Standard I/O Library&lt;br&gt;• Long int data type&lt;br&gt;• Unsigned int data type&lt;br&gt;• Compound assignment operators</td>
</tr>
<tr>
<td><strong>C89/C90</strong></td>
<td>ANSI standard X3.159-1989&lt;br&gt;Completed in 1988&lt;br&gt;Formally approved in December 1989&lt;br&gt;International standard ISO/IEC 9899-1990&lt;br&gt;A superset of K&amp;R C&lt;br&gt;Heavily influenced by C++, 1979-83&lt;br&gt;• Function prototypes&lt;br&gt;• Void pointers&lt;br&gt;• Modified syntax for parameter declarations&lt;br&gt;• Remained backwards compatible with K&amp;R C</td>
</tr>
<tr>
<td><strong>C99</strong></td>
<td>International standard ISO/IEC 9899:1999&lt;br&gt;Incorporates changes from Amendment 1 (1995)&lt;br&gt;Features:&lt;br&gt;• Inline functions&lt;br&gt;• New data types (long long int, complex, etc.)&lt;br&gt;• Variable length arrays&lt;br&gt;• Support for IEEE 754 floating point&lt;br&gt;• Single line comments using //</td>
</tr>
</tbody>
</table>

---

## Properties of C

- Low-level
- Small
- Permissive
Strengths of C

- Efficiency
- Portability
- Power
- Flexibility
- Standard library
- Integration with UNIX

Weaknesses of C

- Programs can be error-prone.
- Programs can be difficult to understand.
- Programs can be difficult to modify.
Effective Use of C

• Learn how to avoid pitfalls.
• Use software tools to make programs more reliable.
• Take advantage of existing code libraries.
• Adopt a sensible set of coding conventions.
• Avoid “tricks” and overly complex code.
• Stick to the standard.
• Try and adapt the good habits from programming in Java!
• MAKE SURE YOU WRITE YOUR OWN CODE.

First C Program: Hello, World!

```c
#include <stdio.h>

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

• This program might be stored in a file named `hello.c`.
• The file name doesn’t matter, but the `.c` extension is often required.
First C Program: Hello, World!

// Name: Xena W. Princess
// Purpose: My first C Program, prints: Hello, World!
// Written on January 22, 2018

#include <stdio.h>

int main(void) {
    printf("Hello, World!\n");
    return 0;
} // end of main()

• This program might be stored in a file named hello.c.
• The file name doesn’t matter, but the .c extension is often required.

Compilation Process

[xena@codewarrior cs246]$ gcc hello.c

[xena@codewarrior cs246]$ ./a.out
Hello, World!
[xena@codewarrior cs246]$
Excursion to Linux

Learn to use: `pwd`, `ls`, `cd`, `cp`, `cat/less/more`, `mv`

Compilation Process – GNU C Compiler

```
[xena@codewarrior cs246]$ gcc -o hello hello.c
```

```
[xena@codewarrior cs246]$ ./hello
Hello, World!
[xena@codewarrior cs246]$
```
Compilation Process

Compilation is a 3-step process

1. **Preprocessing**
   Source code commands that begin with a # are preprocessed. E.g.,
   
   ```
   #include <stdio.h>
   ```

2. **Compiling**
   Source code is translated into object code (m/c language)

3. **Linking**
   All libraries/modules used by the program are linked to produce an executable object code

Preprocessing is normally integrated into the compiler. Linking is done by a separate program/command. The gcc command, in its simplest form, integrates all three steps.

The **gcc** command, in its simplest form, integrates all three steps.
C Program Structure (for now)

```c
#include <stdio.h>

int main(void) {
    printf("Hello, World!\n");
    return 0;
} // end of main()
```

• Before a C program is compiled, it is first edited by a preprocessor.
• Commands intended for the preprocessor are called directives.
• `<stdio.h>` is a *header* containing information about C’s standard I/O library.
**main()**

- The `main()` function is mandatory.
- `Main()` is special: it gets called automatically when the program is executed.
- `main` returns a status code; the value 0 indicates normal program termination.
- If there’s no `return` statement at the end of the `main` function, many compilers will produce a warning message.

**Printing Strings**

- The statement
  ```c
  printf("To C, or not to C: that is the question.\n");
  ```
  could be replaced by two calls of `printf`:
  ```c
  printf("To C, or not to C: ");
  printf("that is the question.\n");
  ```
- The new-line character can appear more than once in a string literal:
  ```c
  printf("Brevity is the soul of wit.\n    --Shakespeare\n");
  ```
Comments – Two styles /*...*/ or //</br><br>• Begins with /* and end with */.

/* No comment */

• Comments can also be written in the following way:

// No comment

• Advantages of // comments:
  • Safer: there’s no chance that an unterminated comment will accidentally consume part of a program.
  • Multiline comments stand out better.

Another Program (variables, assignment, formatted output)

File: small.c
#include <stdio.h>
int main(void) {
    int A, B, C;
    A = 24;
    B = 18;
    C = A + B;

    printf("C = %d\n", C);
} // main()

[xena@codewarrior cs246]$ gcc -o small small.c
[xena@codewarrior cs246]$ ./small
C = 42
[xena@codewarrior cs246]$
Printing the Value of a Variable

- \%d works only for int variables; use %f to print a float variable
- By default, %f displays a number with six digits after the decimal point.
- To force %f to display \( p \) digits after the decimal point, put \( .\ p \) between % and f.
- To print the line
  Profit: $2150.48
  use the following call of printf:
  printf("Profit: $%.2f\n", profit);
- There’s no limit to the number of variables that can be printed by a single call of printf:
  printf("Height: %d  Length: %d\n", height, length);

Input

- scanf() is the C library’s counterpart to printf.
- Syntax for using scanf()
  scanf(<format-string>, <variable-reference(s)>)
- Example: read an integer value into an int variable data.
  scanf("%d", &data); //read an integer; store into data
- The & is a reference operator. More on that later!
Reading Input

• **Reading a float:**
  
  `scanf("%f", &x);`

• "%f" tells `scanf` to look for an input value in **float** format (the number may contain a decimal point, but doesn’t have to).

Standard Input & Output Devices

• In Linux the standard I/O devices are, by default, the keyboard for input, and the terminal console for output.

• Thus, input and output in C, if not specified, is always from the standard input and output devices. That is,

  `printf()` always outputs to the terminal console

  `scanf()` always inputs from the keyboard

• Later, you will see how these can be reassigned/redirected to other devices.
Program: Convert Fahrenheit to Celsius

- The `celsius.c` program prompts the user to enter a Fahrenheit temperature; it then prints the equivalent Celsius temperature.
- Sample program output:
  Enter Fahrenheit temperature: 212
  Celsius equivalent: 100.0
- The program will allow temperatures that aren’t integers.

```c
#include <stdio.h>

int main(void)
{
    float f, c;
    printf("Enter Fahrenheit temperature: ");
    scanf("%f", &f);
    c = (f - 32) * 5.0/9.0;
    printf("Celsius equivalent: %.1f\n", c);
    return 0;
} // main()
```

Sample program output:
Enter Fahrenheit temperature: 212
Celsius equivalent: 100.0
Improving ctof.c

Look at the following command:
\[ c = (f - 32) \times \frac{5.0}{9.0}; \]

First, 32, 5.0, and 9.0 should be floating point values: 32.0, 5.0, 9.0

Second, by default, in C, they will be assumed to be of type `double`
Instead, we should write
\[ c = (f - 32.0) \times \frac{5.0}{9.0}; \]

What about using constants/magic numbers?

Defining constants - macros

```c
#define FREEZING_PT 32.0f
#define SCALE_FACTOR (5.0f/9.0f)
```

So we can write:
\[ c = (f - FREEZING_PT) \times SCALE_FACTOR; \]

When a program is compiled, the preprocessor replaces each macro by the value that it represents.
During preprocessing, the statement
\[ c = (f - FREEZING_PT) \times SCALE_FACTOR; \]
will become
\[ c = (f - 32.f) \times (5.0f/9.0f); \]

This is a safer programming practice.
Program: Convert Fahrenheit to Celsius
ctof.c

#include <stdio.h>
#define FREEZING_PT 32.0f
#define SCALE_FACTOR (5.0f/9.0f)

int main(void)
{
    float f, c;
    printf("Enter Fahrenheit temperature: ");
    scanf("%f", &f);
    c = (f - FREEZING_PT) * SCALE_FACTOR;
    printf("Celsius equivalent: %.1f\n", c);
    return 0;
} // main()

Sample program output:
Enter Fahrenheit temperature: 212
Celsius equivalent: 100.0

Identifiers

• Names for variables, functions, macros, etc. are called **identifiers**.
• An identifier may contain letters, digits, and underscores, but must begin with a letter or underscore:

```c
times10  get_next_char  _done
```
It's usually best to avoid identifiers that begin with an underscore.
• Examples of illegal identifiers:

```c
10times  get-next-char
```
Identifiers

- C is **case-sensitive:** it distinguishes between upper-case and lower-case letters in identifiers.
- For example, the following identifiers are all different:
  ```
  job  jOB  j0b  jOb  Job  JoB  j0B  JOB
  ```
- Many programmers use only lower-case letters in identifiers (other than macros), with underscores inserted for legibility:
  ```
  symbol_table  current_page  name_and_address
  ```
- Other programmers use an upper-case letter to begin each word within an identifier:
  ```
  symbolTable  currentPage  nameAndAddress
  ```
- C places no limit on the maximum length of an identifier.

Keywords

- The following **keywords** can’t be used as identifiers:
  ```
  auto  enum  restrict*  unsigned
  break  extern  return  void
  case  float  short  volatile
  char  for  signed  while
  const  goto  sizeof  _Bool*
  continue  if  static  _Complex*
  default  inline*  struct  _Imaginary*
  do  int  switch
  double  long  typedef
  else  register  union
  ```
- Keywords (with the exception of _Bool, Complex, and _Imaginary) must be written using only lower-case letters.
- Names of library functions (e.g., `printf`) are also lower-case.
If and Switch statements in C

- A compound statement has the form
  
  ```
  { statements }
  ```

- In its simplest form, the if statement has the form
  
  ```
  if ( expression ) compound/statement
  ```

- An if statement may have an else clause:
  
  ```
  if ( expression ) compound/statement else compound/statement
  ```

- Most common form of the switch statement:
  
  ```
  switch ( expression ) {
    case constant-expression : statements
    ...
    case constant-expression : statements
    default : statements
  }
  ```

Arithmetic Operators

- C provides five binary **arithmetic operators**: 
  
  +   addition  
  –   subtraction  
  *   multiplication  
  /   division  
  %   remainder  

- An operator is **binary** if it has two operands. 

- There are also two **unary** arithmetic operators:
  
  +   unary plus  
  –   unary minus
Logical Expressions

• Several of C’s statements must test the value of an expression to see if it is “true” or “false.”
• In many programming languages, an expression such as $i < j$ would have a special “Boolean” or “logical” type.
• In C, a comparison such as $i < j$ yields an integer: either 0 (false) or 1 (true).

Relational Operators

• C’s relational operators:
  - `<` less than
  - `>` greater than
  - `<=` less than or equal to
  - `>=` greater than or equal to
• C provides two equality operators:
  - `==` equal to
  - `!=` not equal to
• More complicated logical expressions can be built from simpler ones by using the logical operators:
  - `!` logical negation
  - `&` logical and

These operators produce 0 (false) or 1 (true) when used in expressions.
Logical Operators

- Both `&&` and `||` perform “short-circuit” evaluation: they first evaluate the left operand, then the right one.
- If the value of the expression can be deduced from the left operand alone, the right operand isn’t evaluated.
- Example:
  
  \[(i \neq 0) \land (j \div i > 0)\]
  
  `(i \neq 0)` is evaluated first. If `i` isn’t equal to 0, then `(j \div i > 0)` is evaluated.
- If `i` is 0, the entire expression must be false, so there’s no need to evaluate `(j \div i > 0)`. Without short-circuit evaluation, division by zero would have occurred.

Relational Operators & Lack of Boolean
Watch out!!!

- The expression
  
  \[i < j < k\]
  
  is legal, but does not test whether \(j\) lies between \(i\) and \(k\).
- Since the `<` operator is left associative, this expression is equivalent to
  
  \[(i < j) < k\]
  
  The 1 or 0 produced by `i < j` is then compared to `k`.
- The correct expression is `i < j \land j < k`. 
Loops

• The while statement has the form
  while ( expression ) statement
• General form of the do statement:
  do statement while ( expression ) ;
• General form of the for statement:
  for ( expr1 ; expr2 ; expr3 ) statement
  expr1, expr2, and expr3 are expressions.
• Example:
  for (i = 10; i > 0; i--)
    printf("T minus %d and counting\n", i);
• In C99, the first expression in a for statement can be replaced by a declaration.
• This feature allows the programmer to declare a variable for use by the loop:
  for (int i = 0; i < n; i++)
    ...

STOP!!
Acknowledgements


Some content is also included from the lecture slides provided by Prof. K. N. King. Thank You!