Faculty of Computer Science, Dalhousie University *21-Sep-2018*

CSCI 2132 — Software Development

Lecture 8: Introduction to C

Location: Chemistry 125 Instructor: Vlado Keselj
Time: 12:35 – 13:25

Previous Lecture

- Filename substitution (wildcards)
- Regular expressions
  - basic regular expressions
  - grep, filters

Some Interesting grep Options

- These are some interesting grep options that can be used:
  `-n`: Output lines preceded by line numbers
  `-i`: Ignores case
  `-v`: Output lines that don't match
  `-w`: Restricts matching to whole words only

Grep Variations

- `grep` : the standard grep
- `grep -F` (or `fgrep`) : searching for fixed strings
- `grep -E` (or `egrep`) : support for extended regular expressions

Extended Regular Expressions (ERE)

- Include matacharacters: `? + | ( ) {`
- These metacharacters can still be used with a backslash; e.g., `\?`
- Back-referencing; e.g., `(...)\1`
- Further extension: PCRE — Perl-Compatible Regular Expressions

Examples of Extended Regular Expressions

- `[0-9]?[0-9][a-z]+`
- `(Mon|Wed|Fri)+`
- `(.)(.).*\2\1`
- `([0-9]{3},){2,5}[0-9]{3}`

Extended regular expressions include additional metacharacters.

The extended regular expressions include the use of parentheses, and also the so-called back-referencing expressions '\\n', such as `\1`, `\2`, `\3`, and so on, which match substrings captured with parentheses. As an example, the extended regular expression: `([ab])cd\1` would match strings: `acda` and `bcdb`, since in the first case 'a' is captured in the first set of parentheses, and 'b' is captured in the second case. As another example, we can use these kinds of expressions to find words that have the first letter the same as the last one, and the second letter the same as the second last letter. If we use a dictionary file, in which each word is placed in one line, we could use the following regular expression: `^(.)(.).*\2\1$`

The back-referencing (backslash-n or `\`n) expressions, like `\1` and `\2` can be used several times in the same expression; for example, the expression:
`^(.)(.).*\2\2\1\1$`
would match the word '`abcdebbaa`'.

Generally, the syntax of BRE/ERE may very between systems, but many features are common and we attempt to emphasize here these common parts. One way to learn more about regular expressions is to use the man command '`man re_syntax`'. However, this man page is a part of a specific language called Tcl.

8 Introduction to C

C Programming Language

- C is originally invented as a language for writing an operating system and other system software by Denis Ritchie
- C optimizes for machine efficiency at the expense of increased implementation and debugging time
- A central difficulty in C programming: programmers must do their own memory management
- C assumes that you know what you are doing

Writing a Simple Program

- `hello.c` — the first C program from K&R
  ```
  #include <stdio.h>

  int main() {
      printf("hello, world\n");
      return 0;
  }
  ```
- We can type this program using *emacs*

Compiling and Running a Simple C Program

- `gcc hello.c` — to compile the program
- `ls -l` — to verify output in `a.out`
- `./a.out` — to run the output
- You can explore Emacs and other tools about how to do this faster

Compiling using Emacs

- We can compile the program from emacs using:
  `M-x compile`
  edit the compile command to be: `gcc hello.c`
- To recompile we can use the command: `M-x recompile`
- If there are errors, we can find them using: `` C-x ` ``
- The same key can be used to find the next error, etc.

Aside: Hotkeys in Emacs

- Initial compilation in emacs: `M-x compile` $\boxed{\text{Enter}}$
- Later, we can use emacs function `recompile`
- It would be convenient to have hotkey that would run recompile
- We also want the hotkey to be defined each time we load emacs
- We do it by modifying `.emacs` file in the home directory; by adding the following line (or multiple lines):

```
(global-set-key [f5]
  (lambda () (interactive) (save-buffer)
             (recompile)))
```

From Source Code to Executable

- Three steps:
  - **Preprocessing** (by a preprocessor): modifies the program by following preprocessor directives
  - **Compiling** (by a compiler): translates modified code into object code (machine instructions)
  - **Linking** (by a linker): combines object code and additional code and produces an executable program
- `gcc` automatically executes these three steps
- Other approach to running programs: interpretation (e.g., shell scripts, Perl, Python)

In the preprocessing phase, the comments are removed from the program, and the preprocessor directives are processed, such as `#include` and `#define`. These are basically textual transformations of the program, and the preprocessor does not include much semantics of the actual program tokens.

In the compiling phase, the source C program code is translated into the machine code, also known as object code. This is the most complex phase.

In the linking phase, the machine code produced in the compiling phase is linked together to form the executable program. The symbols, such as function names, are linked to their actual memory addresses; the library functions code is also added to the final executable, and the library functions are linked with the user code.

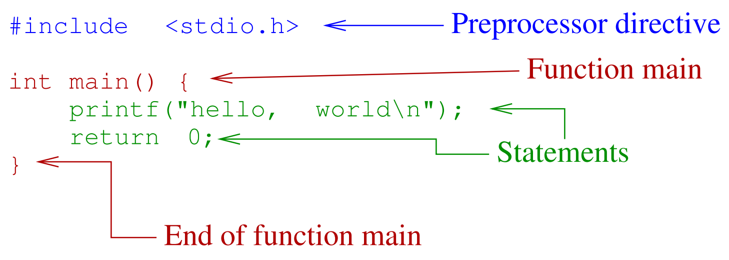
General Form of a Simple Program

```
directives
int main() {
  statements
}
```

or

```
directives
int main(void) {
  statements
}
```

Hello-world Example



```
#include  <stdio.h>

int main() {
    printf("hello,  world\n");
    return  0;
}
```

The following is an explanation of some of these lines:

`#include <stdio.h>` — includes information about C's standard I/O library.

`int main() {` — is a start of the "main" function.

`printf` — is a function from the standard I/O library to produce formatted output.

`return 0;` — return 0 as a return code from the function, which will be the exit code of the program.

Directives

The directives are commands intended for the preprocessor. They are specified by lines that start with the character # and they are one line long. There is no semicolon at the end of a directive. For example, the following line is a directive:

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

The line effectively means that content of the file `stdio.h` is to be inserted in this place during preprocessing. The file `stdio.h` is called a header file, for reasons that we will see later, and it is a system-wide file in a location known to the compiler. For example, it is in the following location at a server: `/usr/include/stdio.h`

Functions

- Building blocks from which C programs are constructed
- A function is a group of statements given a name
- **Library functions:** functions provided as a part of the C implementation; e.g., `printf`
- **Main function:** the function that is called automatically when the program is executed
- `int main()` or `int main(void)` means that main returns an integer value, and does not take any parameters
- Nested functions not allowed by standard, but gcc allows them

Statement

- A command to be executed when the program runs
- Must end with a semicolon
- Examples:
  ```
  printf("hello, world\n");
  return 0;
  ```

Printing Strings

- `printf` can print to the standard output a string literal—a series of characters enclosed between `"` and `"`
- Newline character: `\n`
- Examples:

```
printf("hello, ";
printf("world\n");
printf("hello, \nworld\n");
```

– Similar to Java, string literals can include other escape sequences: `\t`, `\r`, `\\`, `\a`, `\b`, `\f`, `\v`, `\'`, `\"`, `\ooo`, `\xHH`, and `\?`.

Comments

– `/*` comments (one or more lines) `*/`
– Example:

```
/* Name: hello.c
   Purpose: prints hello, world
   Authors: K&R
 */
```

– C99 standard: `//` comments (to the end of line)

Variables

– Types
  – Each variable must have a type
– Examples
  – `int` — integers
  – `float` — floating-point numbers
  – `double` — floating-point with double precision
  – `char` — characters
– We will see later how to build more complex types

Declarations

– Variables must be declared before use
– Syntax: *type name*`;`
– Examples:

```
int height;
float profit;
```

– In C89 or earlier, declarations must precede statements in any block of code
– No such restrictions in C99

Operators

– A rich and powerful set of operators was one of the strong novelties of C
– Some operators (in increasing precedence):
  – parentheses `()`
  – unary + and −, `++`, `--`
  – binary `*`, `/`, `%`
  – binary + and −
  – comparison: <, <= >, >=
  – equality: == and `!=`
  – assignment: =, +=, −=, `*=`, `/=`, `%=`,

Printing Variables

- Printing an integer:
  `printf("Height: %d\h", height);`
- Printing a floating-point number:
  - printing with a default value of 6 decimal digits:
    `printf("Profit: %f\n", profit);`
  - printing 2 digits after the decimal point:
    `printf("Profit: %.2f\n", profit);`

Initialization

- Variables may have a random value if declared and not initialized
- Declare and initialize in one step:

```
int height = 8;
double profit = 1030.56;
float profit = 1030.56f;
char c = 'A';
char b = '\n';
```

A note about float and double

- Generally, `double` is preferred since usually precision is important with floating point numbers
- Textbook seems to use `float` more
- Traditionally, all calculation in C was done in double, but newer standards changed this a bit
- `float` may be a better choice to save memory if storing a *lot* of numbers

printf and float-or-double question

- Formally, a float variable is printed using `%f`:
  `printf("x = %f\n", x);`
- and a double variable using `%lf`:
  `printf("x = %lf\n", x);`
- but it does not matter with `printf`—we can use `%f` in both cases
- However, the difference is significant with `scanf` (formatted input reading)

Reading Input: scanf

- Reading an `int` value:
  `scanf("%d", &height);`
- Reading a `float` value:
  `scanf("%f", &profit);`
- Reading a `double` value:
  `scanf("%lf", &precise_profit);`
- Reading an `char` value:
  `scanf("%c", &ch);`

Defining Names for Constants

- Macro definition (preprocessor directive):
  `#define PI 3.14159f`

- or simply
  `#define PI 3.14159`
- Preprocessor will replace each occurrence of token `PI` with the number
- A macro definition:
  - does not define a variable
  - is oblivious about the content of the replacement
- Macro replacement can be any sequence of tokens

Example: Expression as a Macro

- The value of a macro can be an expression:
  `#define RECIPROCAL_OF_PI (1.0/3.14159)`
- Important to remember to put parentheses `()` around if using an expression
- Example: `double pi = 1.0 / RECIPROCAL_OF_PI;`
- What would happen if we did not have parentheses?
- Convention: uppercase letters are used for constants being defined as macros

Identifiers

- Names for variables, functions, macros, etc.
- May contain letters, digits, and underscores
- Must begin with a letter or underscore
- It is good idea to avoid using underscore as the starting character for now

Example

- Suppose that we write a program for a cashier working in a retail store
- When a customer pays certain amount for a product of certain price, before HST, we want to calculate the balance to be returned to the customer.
- Design:
  - Read price, payment, calculate, print the result
  - HST can be defined as a macro constant, also called symbolic constant

```
#include <stdio.h>

#define HST 0.15

int main() {
    double price, payment, balance;

    printf("Enter price: ");
    scanf("%lf", &price);

    printf("Enter payment: ");
    scanf("%lf", &payment);

    balance = payment - price * (1.0 + HST);
    printf("Balance to be returned to customer: %.2f\n", balance);

    return 0;
}
```