Introduction to C

Stat 580: Statistical Computing

- Theme: Black - White
- Printable version

References

- "The C programming language" by Brain W. Kernighan and Dennis M. Ritchie.
- Part of this slide set is based on *Essential C* by Nick Parlante:

Stanford CS Education Library This is document #101, Essential C, in the Stanford CS Education Library. This and other educational materials are available for free at http://cslibrary.stanford.edu/. This article is free to be used, reproduced, excerpted, retransmitted, or sold so long as this notice is clearly reproduced at its beginning.

(for copyright reason, this notice is reproduced here.)

I know R. Why learn C?

- R (high-level, interpreted language) can be *slow*, due to, e.g., its extreme dynamism. (We will cover this later.)
- C (mid-level language) is fast, powerful and widely used. (See low-level languages.)
- C is easy to interface with R.
- C++ (which inherits most of C's syntax) provides easy and powerful interfacing with R, with the help of various R packages (e.g., RCpp, RCppArmadillo).

Introduction to C

- C is a general-purpose programming language.
- It is closely associated with UNIX system but not tied to any one operating system or machine.
  - So taking Stat580 is not an excuse to buy a supercomputer.
- B (developed by Ken Thompson in 1969-1970) -> C (developed by Dennis Ritchie during 1971-1973). See The Development of the C Language.

Some elements of C programs

- *variables* and *constants*
  - basic data types: characters, integers and floating-point numbers
  - complex data types: e.g., pointers, arrays, structures
- *operators* (e.g., `=`, `+`)
- *control-flow constructions*
- *functions*

First C program: "hello, world"

"hello, world"

Goal: print the words "hello, world"

This *involves*:

1. write the source code
2. compile it
3. load and run it
4. locate your output

Write the source code

```c
#include <stdio.h>

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

- (Use any text editor you like! Two classical choices: Vi (Vim) and Emacs.)
- Some general rules:
  - case-sensitive: `Printf` is different from `printf`
  - free-form line structure: you have to end the statement by `;`.
    - statement can span a few lines
    - multiple statements can be on the same line
    - space is ignored

Write the source code

```c
#include <stdio.h>

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

- `#include <stdio.h>`: loads in `stdio.h` which is called a header file
  - appears at the beginning of the source code
  - to use the standard functions, we usually have to call the corresponding header file
  - `stdio.h` is the header file of C standard input/output library
  - `<file.h>` indicates that the header file `file.h` in `/usr/include`
  - `"file.h"` indicates that the header file `file.h` in the current directory

Write the source code

```c
#include <stdio.h>

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

- C program begins executing at a function `main()`
  - `int` indicates that the `main()` function returns an integer, which matches with the `return` statement.
  - code within `{` and `}` are the code that we want to execute.
  - `main()` usually calls other functions (some that you wrote and others from libraries) to help perform its job .
    - In this case, it calls `printf()` from the standard input/output library.

Write the source code

```c
#include <stdio.h>

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

- One method of communicating data between functions is to provide a list of values, called arguments, to the function it calls.
  - `printf()` is a function, we supply the argument `"hello, world\n"`

Compile it

Save the text as "hello.c" and run the following. (We will use gcc compiler, which has already been installed on the department linux servers.)

Basic way

```
gcc hello.c
```

- This will generate an executable file `a.out`.

With name

```
gcc hello.c -o hello.out
```

- This will generate an executable file `hello.out`.

Compile it

-Wall flag: generate warning messages

```
gcc -Wall hello.c
```

Example

```
#include <stdio.h>

int main()
{
  printf("hello, world\n");
  int i=1;
  return 0;
}
```

-ansi and -pedantic flags: for adhering to the ANSI standard

```
gcc -ansi -pedantic hello.c
```

Compilation

- Preprocessor
  - scans through the source files, removes comments
  - interprets special preprocessor directives (#)
- Compiler
  - processes the source code to make assembly code, a low-level CPU-specific language
- Assembler
  - makes an object file of machine-ready instructions
- Linker
  - links libraries or multiple source files involved (if any) together to produce the executable

Remaining

Load and run it

On UNIX machines:

```
./a.out
```

Locate your output

```
hello, world
```

Comments

- Comments in C are enclosed by slash/star pairs: `/* .. comments .. */` which may cross multiple lines.
- C++ introduced a form of comment started by two slashes and extending to the end of the line: `// comment until the line end`
- The `//` comment form is so handy that many C compilers now also support it, although it is not technically part of the C language.

```
printf("I am comment\n") // I am comment
/* I am
  comment */
```

- Comments are an important part of well written code:
  - describes what the code accomplishes
  - narrates what is tricky or non-obvious about a section of code

`printf()` function

- useful function for understanding and debugging C programs
- general form: `printf(<string>, list of arguments)`
  - `printf("hello world\n")` has a string `"hello world\n"` and no arguments
- In general, `<string>` consists of three elements:
  - text to be displayed
  - format specifiers (to be replaced by the arguments in the display)
  - special characters
- `"hello world\n"` contains no format specifier, but
  - text to be displayed: `hello world`
  - special characters: `\n` (newline). See other special characters.

Example

```c
#include <stdio.h>

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

Second C program: temperature conversion

Temperature conversion

Goal: print a correspondence table of Fahrenheit temperatures and their Celsius equivalents using

$$C = (5/9)(F - 32)$$

```c
#include <stdio.h>

int main() {
  int f, c, lower, upper, step;

  lower = 0;    /* lower limit of the table */
  upper = 200;  /* upper limit */
  step = 25;    /* step size */

  f = lower;
  printf("F\tC\n");   /* table header */
  while (f <= upper) {
    c = 5 * (f - 32) / 9;   /* integer arithmetic */
    printf("%d\t%d\n", f, c);
    f = f + step;
  }
  return 0;
}
```

Variables

- Different from R, variables must be declared before use.
- A simple declaration statement looks like this:

```
int x;
```

  - It consists of variable type `int` and variable name `x`.
  - `int` indicates that `x` is an integer.
  - Other types will be introduced later.

In the example,

```
  int f, c, lower, upper, step;
```

- Several variables are declared in one statement.
  - `f`, `c`, `lower`, `upper` and `step` are all declared to be of type `int`.

Understanding the program

```c
#include <stdio.h>

int main() {
  int f, c, lower, upper, step;

  lower = 0;    /* lower limit of the table */
  upper = 200;  /* upper limit */
  step = 25;    /* step size */

  f = lower;
  printf("F\tC\n");   /* table header */
  while (f <= upper) {
    c = 5 * (f - 32) / 9;   /* integer arithmetic */
    printf("%d\t%d\n", f, c);
    f = f + step;
  }
  return 0;
}
```

`printf()` - a closer look

- general form: `printf(<string>, list of arguments)` where `<string>` consists of three elements:
  - text to be displayed
  - format specifiers (to be replaced by the arguments in the display)
  - special characters

`printf()` - a closer look

```
    printf("%d\t%d\n", f, c);
```

- `%d` is a format specifier
  - `%d` specifies a decimal integer (as opposed to e.g. binary integer).
  - In the display, format specifiers are replaced by the arguments in the same order.
    - The first `%d` is replaced by the value of `f`.
    - The second `%d` is replaced by the value of `c`.

Format specifiers

Format specifier	Description
`%d`	decimal integer
`%5d`	decimal integer, at least 5 characters wide
`%f`	floating pointer number
`%5f`	floating pointer number, at least 5 characters wide
`%.2f`	floating pointer number, 2 characters after decimal point
`%5.2f`	floating pointer number, at least 5 character wide and 2 characters after decimal point

See format specifiers.

`while` loop

`while` loop is one of the control structures, which control the flow of the program.

```
while (<expression>) {
  <statement>
}
```

- While the `<expression>` is true, the loop continues.
- `<expression>` is evaluated before every loop.

```
  while (f <= upper) {
    c = 5 * (f - 32) / 9;    /* integer arithmetic */
    printf("%d\t%d\n", f, c);
    f = f + step;
  }
```

Integer arithmetic

- output from the temperature conversion program:

```
F    C
0    -17
25    -3
50    10
75    23
100    37
125    51
150    65
175    79
200    93
```

- e.g., $5(0 - 32)/9 = -17.77778$ and $5(200 - 32)/9 = 93.33333$.
- The conversion takes away the decimal digits.
- The reason is that the variables in `5 * (f - 32) / 9` are all of integer type
  - which leads to integer arithmetic: the decimal digits are removed
- What is the result of `3/5`?

Using floating point type

```c
#include <stdio.h>

int main() {
  double f, c;
  int lower, upper, step;

  lower = 0;    /* lower limit of the table */
  upper = 200;  /* upper limit */
  step = 25;    /* step size */

  f = lower;
  printf("F\tC\n");   /* table header */
  while (f <= upper) {
    c = 5.0 * (f - 32.0) / 9.0;
    printf("%.0f\t%.1f\n", f, c); /* rounding */
    f = f + step;
  }
  return 0;
}
```

Symbolic constants

- `lower`, `upper` and `step` are tuning parameters.
- We want to provide a systematic way for one to change them without digging into the program.
- We can use `#define` preprocessor directive.

```
#define lower 0
#define upper 200
#define step 25
```

- The preprocessor will replace the symbols (`lower`, `upper`, `step`) by their values (`0`, `200`, `25`) before compilation.

Symbolic constants

This program

```c
#include <stdio.h>
#define lower 0
#define upper 200
#define step 25

int main() {
  double f, c;

  f = lower;
  printf("F\tC\n");   /* table header */
  while (f <= upper) {
    c = 5.0 * (f - 32.0) / 9.0;
    printf("%.0f\t%.1f\n", f, c); /* rounding */
    f = f + step;
  }
  return 0;
}
```

Symbolic constants

is equivalent to this one:

```c
#include <stdio.h>




int main() {
  double f, c;

  f = 0;
  printf("F\tC\n");   /* table header */
  while (f <= 200) {
    c = 5.0 * (f - 32.0) / 9.0;
    printf("%.0f\t%.1f\n", f, c); /* rounding */
    f = f + 25;
  }
  return 0;
}
```

Character input and output

Standard I/O library

- In standard library, the input or output is dealt with as streams of characters.
- A text stream is a sequence of characters divided into lines.
- Each line consists of zero or more characters followed by a newline character.
- We will focus on:
  - `getchar()`: reads the next input character
  - `putchar(c)`: prints the character stored in `c`

File copying

```c
#include<stdio.h>

int main(){
  int c;

  c = getchar();
  while (c != EOF){
    putchar(c);
    c = getchar();
  }
  return 0;
}
```

- Variable `c` is declared to be an `int`.
  - We usually declare a character as `char`.
  - However, `getchar()` distinguishes the end of the input from valid data by returning `EOF` (end of line) if it hits the end.
  - `EOF` does not belong to `char` and thus we need a bigger type which is `int` (this will be made clear in the next slide set).

Character counting

```c
#include <stdio.h>

/* count characters in input */
int main() {
  int count;

  count = 0;
  while (getchar() != EOF) {
    count =  count + 1; /* count++ */
  }
  printf("\nNumber of characters: %d.\n", count);
  return 0;
}
```

Guess how many words if we type in the following without hitting "Enter" in the last line.

```
123

45
```

How can you count the number of lines?