UNIX and C Programming (COMP1000)

Lecture 1: Basics and Revision

Updated: 30th July, 2019

Department of Computing
Curtin University

Outline

What is C?

C is:

- ▶ A programming language – one of thousands.
  - ▶ Not the best, but perhaps the most *important*.
- ▶ An old language –
  - ▶ Simple and primitive, powerful and pervasive.
- ▶ A *compiled* language –
  - ▶ You write "source code".
  - ▶ Then, you convert it to "machine code" using a compiler.
  - ▶ The machine code is the final program.
- ▶ A *statically-typed* language –
  - ▶ Strict data types: integers, real numbers, chars, etc.
  - ▶ These are known at compile time ("static").
- ▶ A *procedural* language –
  - ▶ Not object orientated – no classes or objects.

The pervasiveness of C

▶ As of July 2015, the Ohloh website tracked about 75 thousand C projects with 5.9 *billion* lines of code [1].

▶ Well-known software written in C includes:
  ▶ Much of a typical Linux distribution, particularly the kernel;
  ▶ Android — the mobile phone/tablet OS (based on Linux);
  ▶ Apache — the web server software that runs the majority of the world's websites.

▶ Much more software is written in languages based on C:
  ▶ Microsoft Windows (C++).
  ▶ Mac OS (Objective C).
  ▶ Mozilla Firefox (C++).
  ▶ Google Chrome (C++).
  ▶ Internet Explorer (C++).

---

[1]https://www.ohloh.net/languages/c

Why use C?

Because:

- ▶ It's so widely used already (new graduates often have to maintain old code!).
- ▶ It's very efficient (if you know what you're doing).
- ▶ It's portable (to an extent).

A very, very brief history

- ▶ "C" itself is descended from other languages (most directly from "B").
- ▶ For a long time, C was not well standardised — unlike Java, there was no single organisation in control.
- ▶ Incompatibilities arose between different implementations.
- ▶ However, C standards do now exist — C89/C90, C99 and C11.
- ▶ C89 (ANSI) and C90 (ISO) are the same thing.
- ▶ C99 and C11 are minor updates.

C99 and C11

- ▶ The new versions — actually not very important.
- ▶ You need to know the *old* C (i.e. C89/C90).
    - ▶ The old C is the lowest common denominator; used everywhere.
    - ▶ If you want a "new" language, why use C at all? Use Java, C#, Python, etc. instead.
- ▶ Unfortunately, some C books (e.g. Kochan) don't say what's C99-specific and what isn't.
- ▶ The lectures will briefly mention C99 features where relevant.

C++

- ▶ A major extension to C, implementing object orientation (and other interesting features).
- ▶ Very widely used, alongside C.
- ▶ Some features of C++ have been retroactively added to C.
- ▶ Like C99, **C++ will not be assessed in this unit!**
- ▶ However, the last lecture will discuss C++, and it is well worth learning.

What is there to know about C?

- ▶ Pointers, pointers, pointers and pointers.
- ▶ Engineering Programming (COMP1004) mostly skips around pointers.
- ▶ Pointers are the heart and soul of C — most of C's more useful features are built on pointers.
- ▶ That said, you *also* need to know about:
    - ▶ The structure of a C program.
    - ▶ The environment (UNIX) in which you're working.

Things you should have seen before

These are (essentially) the same in both C and Java:

- ▶ Symbols like semicolons ("`;`"), braces ("`{...}`") and parentheses ("`(...)`").

- ▶ Simple data types (e.g. `int`, `double`, `char`).

- ▶ Variables and assignments (e.g. "`int v = 42;`").

- ▶ Expressions (e.g. "`v = 33 * x + z / 3;`").

- ▶ `if`, `else`, `switch`, `break`, `default`, `for`, `while`, `do`, `return`.

- ▶ Functions/methods; e.g.

```
double calculate(double x, double y)
{
    return x * y;
}
```

Remember variables and data types?

▶ Each variable stores a single value, depending on its data type.

▶ Simple data types include:

<div style="margin-left:2em">

Signed integers: `int`, `short`, `long`.

Unsigned integers: `unsigned int`, `unsigned short`, `unsigned long` (cannot be negative).

Reals: `float`, `double`, `long double`.

Characters: `char` (letter, digit, symbol, etc.) – technically a kind of integer.

</div>

▶ Mainly use `int`, `double` and `char`.

▶ There are no classes or objects in C (unlike Java).

▶ Other special data types including pointers, arrays, `structs`, `unions` and `enums` (which we'll cover later).

▶ C has strings, but no formal "String" data type (we'll get to that too).

Variable declarations

▶ Variables must be *declared* before they can be used.
▶ A declaration consists of a datatype and a name:

```c
float number;        /* Declaration */
number = 5.0 * 2.5;  /* Assignment */
```

Here, number is declared (created) as a float (real number).
▶ Declarations can also *initialise* variables:

```c
float number = 5.0 * 2.5;   /* Same as above */
```

▶ Multiple variables can be declared at once, if they have the same data type:

```c
int alpha = 3, beta, gamma, delta = 6;
```

Here, alpha, beta, gamma and delta are all ints. Two of them have been initialised.

Variable names

- ▶ Should reflect the meaning of the variable!
- ▶ Should be concise, but not too concise.
- ▶ Can contain letters (a-z and A-Z), digits (0-9) and underscores (_).
- ▶ Cannot start with a digit.
- ▶ Case-sensitive (size, SIZE and Size are distinct names).
- ▶ Cannot be a reserved word (auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile or while).

A small difference from Java

How do you repeat something 100 times?

- ▶ In Java and C99 you can do this:

```
for(int i = 0; i < 100; i++) { ... }
```

- ▶ In C (in general), you cannot declare i inside the for
  statement. Instead, you must do this:

```
int i;
for(i = 0; i < 100; i++) { ... }
```

Comments

Use comments to describe code in natural language.

Block comments

Traditionally, comments are enclosed between /* and */

```
/* Speed of light
(metres per second) */
int speed = 299792458;
```

Single-line comments (only C99!)

In C99, single-line comments start with "//".

```
int speed = 299792458; // Speed of light (m/s)
```

. . . but you can't do this in C89.

Comments, meaningful names, indentation and whitespace

- ▶ It's not just machines that read your code.
- ▶ Source code must frequently be fixed, updated, improved, etc.
- ▶ This requires that humans be able to read your code as well.
- ▶ **Comments** help remind you, and explain to others, what the code does and how it works.
- ▶ **Meaningful variable names** serve a similar purpose.
- ▶ **Indentation** helps you see the code structure instantly.
- ▶ **Whitespace** helps you see words and numbers more easily.
- ▶ You *must* use all these — properly — to make your code readable!

Revision

The following code is valid in both C and Java. Take a minute to figure out what it does.

```c
int base = 3;
int exponent = 19;
int result = 1;
int i;

for(i = 0; i < exponent; i++)
{
    result = result * base;
}
```

More Revision

Do you know how functions or methods work?

```c
int addCubes(void)
{
    int i, result = 0;
    for(i = 1; i < 5; i++)
        result += cube(i);
    return result;
}

int cube(int x)
{
    return x * x * x;
}
```

This will give you a result of $1^3 + 2^3 + 3^3 + 4^3 = 100$.

Jumps — `return`, `break`, `continue` and `goto`

- ▶ These statements jump to another place [2].
- ▶ `return` and `break` are necessary in certain places.
- ▶ Many advise against `goto` under any circumstances.
- ▶ Some advise against `break` (outside of `switch`) and `continue` as well.
- ▶ Some advise against multiple `returns` in a single function.
- ▶ However, it's worth noting what they actually do.

---

[2]In fact, `if`, `switch`, `for`, `while` and `do-while` all jump as well, but they provide *structure* to it.

The `break` statement

▶ `break` immediately ends the current `switch`, `for`, `while` or `do-while` statement.

▶ Often used in a `switch` statement:

```
switch(...) {
    case 1:

        ...
        break;
    case 2:

        ...
        break;
    ...
}
```

This is normal, and necessary.

The `break` statement

▶ Can also end a loop (usually from within an `if`):

```
while(...) {
    ...
    if(...) {
        break;
    }
    ...
}
```
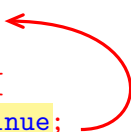
▶ This is *not* necessary — rearranging the `if` statement will achieve the same thing.

▶ `break` can lead to more compact code.

▶ `break` can also lead to unreadable code.

The `continue` statement

▶ `continue` ends the current *iteration* of a loop.
▶ Jumps back to the top of the loop, and continues with the next iteration, if any.

```
while(...) {
    ...
    if(...) {
        continue;
    }
    ...
}
```
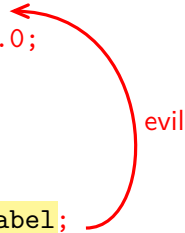
▶ `continue` is never strictly necessary — `if` statements can achieve the same thing.
▶ Like `break`, continue can lead to more compact code.
▶ Like `break`, continue can also lead to unreadable code.

The `goto` statement

- A relic from the bad old days, now very poor practice.
- `goto` simply jumps somewhere else, ignoring all the rules about flow of control.
- The target location is marked by a *label* — a name followed by a colon (":").

```
while(x > 2.0) {              /* Never, ever do this!! */
    theLabel:
    x = x - 2.0;
}

if(x < 3.0) {
    x += 5.0;
    goto theLabel;
}
```

evil

Spaghetti code

- ▶ goto can easily turn your code into spaghetti:
    - ▶ Not clear what order things happen in.
    - ▶ Very difficult to read, understand and fix — easier to throw it out and start again.
    - ▶ Your life will be miserable!
- ▶ break and continue are more constrained, and so less dangerous, but can still do the same thing.
- ▶ Newer languages don't have goto, for good reason.
- ▶ However, in C, goto is sometimes used to separate error handling code from the main algorithm, because C lacks exception handling.

What is a function?

- ▶ C programs are broken up into functions.
- ▶ Each function is a chunk if code, which may "call" (or "invoke") other functions.
- ▶ Functions are *separate* — they cannot contain one another.
- ▶ Execution always starts with the main() function.
- ▶ When function alpha() calls function beta():
    1. alpha is paused;
    2. beta receives "parameter" values (data) from alpha;
    3. beta performs its task;
    4. beta optionally "returns" a value (data) to alpha;
    5. alpha resumes where it left off.

    (Note: the words "call" and "invoke" always carry this precise meaning. You cannot "call" anything other than a function.)

C functions vs. Java methods

▶ For those having studied Java, a function is very similar to a method.

▶ Just like a method, a function has:
  ▶ any number of parameters (imports);
  ▶ an optional return value (an export);
  ▶ its own set of "local" variables; and
  ▶ a chunk of code that performs a specific task.

▶ However, functions are *stand-alone*, whereas methods occur inside classes (and classes don't exist in C).

Anatomy of a function

The following example function takes (imports) a real number and returns (exports) either the letter L or the letter H.

```c
char indicator(float measurement)
{
    char result = 'L';
    if(measurement >= 0.5)
    {
        result = 'H';
    }
    return result;
}
```

Historical interest

▶ Pre-1989 C had a different function parameter syntax:

```
int someFunction(x, y)
    int x;
    float y;
{
    ...
}
```

▶ In modern C, this would be written as:

```
int someFunction(int x, float y)
{
    ...
}
```

▶ The old form is still valid C, but rarely used.

Function calls and returns

```
void calc(void) {
    ...
    int number = addTwo(5);
    ...
}

int addTwo(int param)
{
    int result = param + 2;
    return result;
}
```
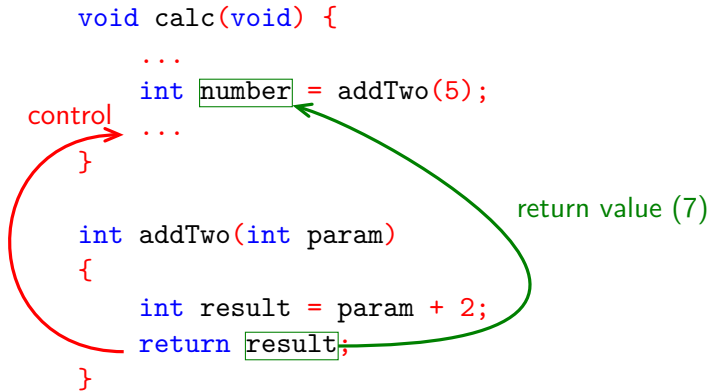
Function calls and returns

The call:

▶ Function `calc()` calls function `addTwo()`.

▶ `calc()` pauses, `addTwo()` begins.

▶ `param` becomes 5.

Function calls and returns

```
         void calc(void) {

             ...
             int number = addTwo(5);
control      ...
         }


         int addTwo(int param)
         {
                                            return value (7)
             int result = param + 2;
             return result;
         }
```

The return:

▶ addTwo() finishes, calc() resumes.
▶ number takes the value of result (7).

The `return` statement

▶ Immediately ends a function — jumps back to the caller.

▶ Returns a value to the caller (if this function is non-void).

▶ Multiple return statements can lead to spaghetti code:

```c
int badExample(void) {
    if(...) return x;
    if(...) return y;
    return z;
}
```

▶ Try for one return only, on the last line:

```c
int goodExample(void) {
    ...
    return result;
}
```

C at a glance	Revision	Jumps	Functions	Expressions	C Programs
ooooooo	ooooooooo	oooooo	ooooooo●ooo	ooo	ooooooooo

Void

▶ Can show that a function does not return a value:

```
void outputProduct(int x, int y) {
    printf("The product is %d\n", x * y);
}
```

▶ Can show that a function has no parameters:

```
void hello(void) {
    printf("Hello world.\n");
}
```

(Note: *this* void is sometimes omitted altogether, but technically that means "anything goes".)

▶ void has a third use related to pointers, which we'll discuss in a couple of weeks.

Void returns

▶ A void function, returning no value, needs no return statement.

▶ These functions hand back control at the end, automatically:

```c
void hello(void) {
    printf("Hello world.\n");
}
```

▶ However, you *can* technically do this:

```c
void hello(void) {
    printf("Hello world.\n");
    return;
}
```

▶ You can abuse void returns much like the other jump statements, but you wouldn't do that, would you?

The printf() function

- ▶ Used to output text and other values.
- ▶ A pre-existing function available for you to use.
- ▶ Works differently from the Java System.out.println() method (but used for the same purpose).
- ▶ Described later in more detail.

Examples

```
printf("Hello world.\n");
```

```
printf("Coordinates: (%d, %d)\n", x, y);
```

The scanf() function

- Used to input values (like the reverse of printf()).
- Described later in more detail.

Example

```
int value;
printf("Enter value: ");
scanf("%d", &value);
```

(The printf() here is not strictly necessary.)

Expressions

- ▶ Expressions occur virtually *everywhere*.
- ▶ They are built up from:
  - ▶ operands — variables, function calls and "literal" values;
  - ▶ operators — +, *, >=, ++, &&, typecasts, etc.; and
  - ▶ parentheses — "(...)".
- ▶ They have a resulting data type, based on the operators and operands.
- ▶ Variables, function calls and literal values are syntactically interchangeable.
  - ▶ Variables and function calls are (generally) allowed anywhere that values are needed.

Statements that require expressions

These statements all use expressions:

- ▶ `someVariable` **=** **expression** `;`
- ▶ `someFunction(` **expression1** `,` **expression2** `, ...);`
- ▶ `if(` **expression** `) { ... }`
- ▶ `switch(` **expression** `) { ... }`
- ▶ `while(` **expression** `) { ... }`
- ▶ `do { ... } while(` **expression** `);`
- ▶ `for(` **expression1** `;` **expression2** `;` **expression3** `) { ... }`
- ▶ `return` **expression** `;`

There are no limits to how complex expressions can be! (However, simpler expressions are more readable.)

Function calls are values

```
/* Here we call beta and gamma, and then alpha. */
alpha(beta(), gamma() + 3.0);
...

float beta(void) {
    return 5.0;
}

float gamma(void) {
    return 7.0 ;
}

void alpha(float x, float y ) { ... }
```

Function calls can be part of any expression, just like variables.

Function calls are values

```
/* Here we call beta and gamma, and then alpha. */
alpha(beta(), gamma() + 3.0);
...

float beta(void) {
    return 5.0;
}

float gamma(void) {
    return 7.0 ;
}

void alpha(float x, float y ) { ... }
```

Here, beta() and gamma() are called first, and return their values.

Function calls are values

```
/* Here we call beta and gamma, and then alpha. */
alpha(beta(), gamma() + 3.0);
...

float beta(void) {
    return 5.0;
}

                                    10.0 (7.0 + 3.0)

float gamma(void) {
    return 7.0 ;
}
                            5.0

void alpha(float x, float y ) { ... }
```

These values immediately become parameters to alpha().

A simple C program

```c
#include <stdio.h>

int main(void)
{
    printf("Hello world.\n");
    return 0;
}
```

Forward declarations

What's wrong with this code?

```c
void printSum(int x, int y) {
    printNumber(x + y);
}

void printNumber(int number) {
    printf("%d\n", number);
}
```

In C, the order of declaration is important (unlike Java).

Forward declarations (2)

▶ In C, a function must be declared *above* its first use.

▶ If needed, you can place the function "prototype" at the top.

Example

```
void printNumber(int number);

void printSum(int x, int y) {
    printNumber(x + y);
}

void printNumber(int number) {
    printf("%d\n", number);
}
```

Forward declarations (3)

Some terms:

- A *prototype* consists of the name, parameters and return type of a function.
- A *forward declaration* gives the prototype by itself (usually at the top of the source code).
- A function *definition* gives the prototype *and* the actual code.

(You can sometimes just re-arrange the order of the functions, but this is often inconvenient or even impossible.)

C program structure

- A C program consists of one or more files.
- Each file contains one or more functions.
- One file contains the main() function.

C at a glance
0000000

Revision
000000000

Jumps
0000000

Functions
00000000000

Expressions
000

C Programs
000000●000

C program structure

```c
#include <stdio.h>

void bananas(int n);
double mangos(void);

int main(void) {
    ...
    return 0;
}

void bananas(int n) { ... }

double mangos(void) { ... }
...
```

Compiling your C programs

- ▶ In Java you use javac.
- ▶ In C you can use gcc, though there are many alternatives.
- ▶ To convert your source code (prog.c) into an executable program (prog):

```
[user@pc]$ gcc prog.c -o prog
```

- ▶ "-o name" gives the output filename.
  - ▶ Without it, the executable will be called "a.out", which is silly.
- ▶ There are other options as well; e.g.
  - ▶ -Wall turns on "all warnings".
  - ▶ -ansi turns *off* C99 extensions (leaving C89 only).
  - ▶ -pedantic turns on additional checks for C89 compliance.
- ▶ These options help you produce better, more portable code.

```
[user@pc]$ gcc -Wall -ansi -pedantic prog.c -o prog
```

Running your C programs

▶ Unlike Java, C programs *don't* (typically) run on a "virtual machine"

▶ You don't need any other software installed, except for:

    ▶ the operating system

    ▶ any libraries your program uses (via #include)

Example

```
[user@pc]$ ./prog
```

That's all for now!

▶ Make sure you attempt the pre-lab exercises before coming to class.

▶ If you have any uncertainties, please see the lecturer or unit coordinator ASAP!