THE UNIVERSITY OF SYDNEY

INFO1910 S1 2023 Week 2 Tutorial

Introduction to C

More Linux

Having finished our foray into 1980's style computing, boot your computer to Linux and log in using the window manager. Open up a terminal, start a web browser (generally the firefox, chromium or chrome commands work here) and download the tutorial sheet from Ed. Unlike last week we now have a rendered graphical environment and can take advantage of modern conveniences such as pdf formatting.

In your terminal (please do not use a graphical file manager), navigate to your downloads folder and open up the tutorial sheet using **evince**. As this process will be bound to that particular terminal, you may wish to consider backgrounding it using an & so that you can continue to use the terminal.

Some other useful linux commands for opening up or manipulating particular files include **feh** which displays images, and the **zip** and **tar** commands.

Question 1: Customising Vim

As you are going to be using either vim or emacs quite a bit this semester to write code, it's useful to know how to modify their functionality if required.

Open up vim and enter the command :version. This should display the packages and features of your current version of vim along with where the vimrc and other vim dotfiles are stored.

Have a look through the vim dotfiles, in particular .viminfo.

Here are some useful lines that you might want to add to `.vimrc`

```
syntax on
set number

set colorcolumn=80

filetype plugin indent on
set tabstop=4
set expandtab
set shiftwidth=4

colorscheme default
```

- What is the purpose of the .viminfo file?
- Create or edit your /.vimrc file and add the lines 'syntax on' and 'colorscheme gardener'.
- Try out the different colour schemes on the bash scripts you wrote in Tutorial 1. You can swap schemes within vim using the :colorscheme <scheme> command, adding the scheme to your .vimrc file will make it your default scheme.

Emacs configuration files are in and of themselves full programs written in lisp that manage how the emacs server starts and renders itself.

You may notice at this point that a large number of dotfiles that you have seen end in the letters rc. These are an initialism for 'run commands' and indicate that the contents of this file are called when the associated program is run. vimrc is then what is run when vim is started, while bashrc is run when bash (the command line) is started.

If you want something to run every time you open a terminal, it would then be added to `.bashrc`.

Hello World

In addition to bash scripts, you may have noticed a number of compiled files as you've been rummaging around. These can be identified as unreadable messes of poorly rendered symbols. You can see one by running `cat` on just about any file in */bin*, for example `cat /bin/grep`.

This mess of symbols is a series of assembly instructions that indicate a number of operations that a processor is to perform. Obviously writing this directly is a painful and time consuming exercise, so programming languages have been developed that 'compile' down to this assembly.

One such language is C, which co-incidentally you will be learning this semester.

As is often the starting point for learning a new language, the syntax for hello world in C is as follows:

```c
#include <stdio.h>

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

The #include preprocessor command links the stdio.h library, or the standard input and output library to the file to be compiled in with the code. The type that the function returns is specified by 'int' which is short for integer, and we can see that the function returns the integer 0.

We can also see that the function is named 'main'. This is a special function name in C such that when the code is run, the 'main' function will be the first one to be called. The `printf` function is one of the print statements in C and it sends a text stream to standard out, allowing it to interface with the rest of Linux.

While it is nice to have this sitting around in a file, in order to run this code we need to compile it. There are two major (one might even say competing) C compilers: gcc and clang. The name of the compiled file can be specified using the -o flag as follows.

```
gcc -o hello_world hello_world.c
```

or

```
clang -o hello_world hello_world.c
```

Then you can simply execute the compiled code.

```
./hello_world
```

Question 2: Hello World

Try writing, compiling and running hello world in c.

- Modify the file to instead print your name
- What does the compiled file look like?
- Use the `strings` command on the compiled file, what human readable strings can you extract? (if you know what you're expecting here already, use grep to find it more easily)

Compiler Flags

The C compiler can take a number of flags as arguments that modify how the compilation step works and how it should prioritise dealing with potential errors. By default the compiler tries as hard as it can to simply produce a compiled file that runs. Unfortunately this means that it does its best to ignore any and all potential problems along the way.

Using compiler flags where appropriate can turn on error and warning messages to either stop the compiler when it encounters a potential problem, or to

Some common flags that you should probably consider using, and some that you certainly should use or should know when to use:

- -std=c99 While there exist newer version of the C language, C99 is relatively universally used now and allows for modern conveniences like declaring variables inside for loop condition statements and copy operations for some object types.
- -ansi Compiles using the C89 standard, this is more restrictive than the C99 but is more cross platform compatible. Obviously don't use -std=c99 if you are using this
- -pedantic Compiles to the strict C89 implementation of the C89 standard rather than just the current implementation of the C89 standard. This is even more cross compatible, but has limitations that were last seen in the 80's and 90's such as maximum string lengths of around 500 characters.
- -Werror Turns warnings into errors

- -Wall Turns on all warnings
- -Wunreachable-code Throws a warning if the compiler finds code that will never be executed.
- -lm Links the kernel maths libraries for code that uses matrix multiplication among other things
- -0 Slows down compilation speed to allow for better error checking, compiling will take longer by association.
- -g Adds debugging information to the compiled file

Try compiling your hello world code with some of the above flags. See what happens if you put a second print statement after the return statement and use the -Wunreachable-code flag. Do any of the flags differ between gcc and clang?

If you want to view a human readable representation of the assembly code that your C is being compile to, use the -S flag when you compile and have a look in the associated .s file. In order to see more information, the **objdump** command can be called on a compiled binary file to view headers and other details associated with the file.

Make Files

We can write a near neighbour to a bash script to handle the compilation of C code from one, or many files using a Makefile. These files exist alongside C and C++ code and manage the compilation of more complex projects without needing to re-type the instructions to the compiler each time. By default the **make** command will look for a file named *makefile* and use it to compile the code as directed.

A typical (very simple) make file will generally follow the following syntax:

```
# Set variables
CC = gcc
CFLAGS = -Wall -Werror
TARGET = hello_world

# If the rule all is given then make the target
all: $(TARGET)

# If a particular target is given then make it
$(TARGET): $(TARGET).c
        $(CC) $(CFLAGS) -o $(TARGET) $(TARGET).c

# If the rule clean is given then clean up the compiled files
clean:
        rm $(TARGET)
```

It is more than a good idea to keep each separate instance of C code in a separate folder with its own make file, rather than trying to remember how to compile each one separately.

Make files contain Rules, Requirements and Directives, along with regular bash variables.

```
RULE: REQUIREMENTS
    INSTRUCTIONS
```

When the command `make <rule>` is entered, the requirements for that rule are checked, these requirements take the form of a series of files that must exist for the instructions to be performed (if the file hasn't been compiled yet, then it can't be used). If the requirements are met, then the directives are performed, if they are not met, then all the other rules are expanded to see if they provide the required files, if they do then their requirements are checked and so on.

The top of this is the `make all` or in the shortened form just the `make` command, which invokes the *all* rule. In the above case that has the requirements of the target, and will invoke the `make target` rule.

We can also see that `make clean` has no requirements, if the file doesn't exist to remove it then there's no cause for concern.

Question 3: Make script

Write a make file for your hello world script, you will be asked to continue to make make files for every C program you write this semester.

If you have questions about, or are interested in make files, the GNU project has the handbook on their website.

Variables, Literals, Types and Operators

Variables are fixed size blocks of memory with a label. A type is an interpretation of that block of memory which provides a context to perform operations. The implementation of these operations is tied to the binary encoding of the type of that variable.

A literal is a non-variable value, and hence cannot be modified, but can be used in expressions.

```c
'a' // Character literal
97 // Integer literal
0.97 // Double literal
"Hello World" // String literal
```

An expression is a syntactically legal sequence of operators and operands. Operators may be unary, binary or ternary and must act on the correct number of operands to be syntactically valid. The operands of an expression may be another expression, a variable or a literal.

```c
!0 // Unary operator
1 + 2 // Binary operator acts on two operands
(1 + 2) + (1 + 2) // Expressions may also be operands
```

We have a range of primitive types in C, a few are shown below:

```c
char a = 'a';
int b = 97;
float c = 0.97;
double d = 0.00097;
```

Here the = symbol is an operator, while `a, b, c` and `d` are variables, with types specified by the `char, int, float` and `double` keywords.

For today we will only be dealing with non-pointer types, which means you won't be able to assign "strings", we will also not have a boolean type, as the smallest allocatable unit of memory is one byte.

Operators act on variables, can either be unary, binary or ternary. Not all operators are defined over all types, or have consistent behaviour over all types. This leads to a very general overview of programming, as a collection of types and operators that map between them. In classical computing, we can even demonstrate that a small subset of operators can perform the tasks of all other operators, these are termed universal operators.

We can change one type to another using either an implicit or an explicit type conversion. Implicit type conversions require a bit of care when tracking,

```c
char a = 97; // Here 97 is implicitly cast to char
double b = 16; // Another implicit cast, this time to double
printf("%d", (int)a); // Here the variable a is explicitly cast
```

Question 4: Casting

- Cast a character to an integer and print the integer, what dictates the value of the integer?
- Cast a float to an integer, what dictates the value of the integer?
- Cast a character to a float, what dictates the value of a float?

Question 5: Operators

Operators are syntactic sugar for functions. They operate within and between the domains of types.

Name each of these unary operators and describe their action, what types are these operators defined over, and for what types do these operators exhibit inconsistent behaviour?

- `!`
- `~`
- `++`
- `--`
- `(<type>)`

- `sizeof()`

Next these binary operators

- `=`
- `==`
- `+`
- `-`
- `*`
- `/`
- `%`
- `&`
- `&&`
- `|`
- `||`
- `^`
- `<<`
- `>>`

And finally, some compound operators, there are more of these:

- `+=`
- `-=`
- `%=`
- `&=`
- `|=`
- `<<=`
- `>>=`

Just for fun, here's an extra ternary operator: `A ? B : C`

Question 6: Two's Complement

A bitmask is a value that can be used to filter certain bits out of a byte. For example if we only want the value of the 5th bit we can use the bitmask:

```
x & (1 << 5)
```

- How does the bitmask above work?
- Construct a bitmask for the first four bits.
- Using the NOT operator map the masked bit to the lowest order position

Using bitmasks, print the binary value of an integer storing the value '5'. Try it for '6'.

- Now try -5 and -6
- Predict what the value for -1 will be, why is this the case?

More Types and literals

We have a few more keywords up our sleeve when it comes to types in C. In particular we have unsigned integer types. While signed types use two's complement representation, unsigned types are positive integers (including zero) in base two.

```
unsigned char // Unsigned character
signed char  // Signed character
unsigned int // Unsigned integer
signed int // Signed integer
```

We also have size modifiers.

```
char  // 8 bit integer
short int  // >=16 bit integer
int // >=16 bit integer (typically 32)
long int // >=32 bit integer
long long int // >=64 bit integer
long double  // >=96 bit double
```

To go with these types we also have some more literals.

```
unsigned int x = 10u; // literal unsigned 10
float y = 0.7f; // literal float
long int x = 93l; // literal long integer
```

Type Promotions and Demotions

All binary operators in C act on two expressions of the same type and returns an operand of that type. Addition is defined over two integers of the same size and signedness. As a result of this when operating on two variables of different types we need to implicitly promote one of the types to the other.

$$A_t \circ B_t \to C_t$$

This causes issues when our operands are of different types. To resolve this we have three simple rules that are applied in order on any operator acting over operands of different types.

1. Non-floating point types are promoted to floating point types.
2. Smaller types (in terms of the number of bytes) are promoted to larger types.
3. Signed types are promoted to unsigned types.

The assignment operator is the exception to these rules. The assignment operator requires the *demotion* of the left hand operand to the type of the right hand operand. Another exception are the bitshift operators. For these operators the right hand operand must be an integer type, and hence the first rule above is skipped.

Question 7: Confused Types

What is the output of the following code and why?

```c
int x = -1;
unsigned int y = 1;
printf("%d\n", x < y);
```

What about this snippet?

```c
float x = 1 / 2;
printf("%f\n", x);
```

And lastly...

```c
char c = 'a';
printf("%c\n", (c << 8) >> 8);
```

Basic Functions

A function is a callable section of code. Functions are defined by their arguments and their return type, they are also assigned a 'label', or a name. A function must be declared before it can be used, otherwise the compiler will not recognise the function name as a valid label. Below you can see an example function, it takes a single integer argument and returns an integer argument, the function itself simply returns the argument multiplied by ten.

```c
int mult(int y, int z)
{
    return y * z;
}

int main()
{
    int x = 5;
    x = mult(x);
    printf("%d\n", x);
    return 0;
}
```

Arguments to functions are copied; the version that the function handles is not the same as the argument that is passed to it. As a result of this, if we want to modify variables using the function, we need to return them. In C a function can only ever return a single variable.

It's somewhat messy to place all the function declarations before main. We can separate the notion of a function declaration (that creates the 'label' for the function in the symbol table), and the definition of the function that defines what the function does.

We will also briefly note the **void** type. This can be thought of as a 'None' type and cannot be used for variable declarations, but may be used for the return type of a function that does not return anything.

```c
////////////////////////////////////////////////////
// Function Declarations
////////////////////////////////////////////////////
/*
 * mult
 * Multiplies two integer arguments together
 * :: int y :: One of the integers to be multiplied
 * :: int z :: The other integer to be multiplied
 * Returns an integer value containing the multiplied arguments
 */
int mult(int y, int z);

////////////////////////////////////////////////////
// Function Definitions
```

```c
////////////////////////////////////////////////////

int main()
{
    int x = 5;
    x = mult(x, 10); // Multiply x by 10
    printf("%d\n", x);
    return 0;
}

/*
 * mult
 * Multiplies two integer arguments together
 * :: int y :: One of the integers to be multiplied
 * :: int z :: The other integer to be multiplied
 * Returns an integer value containing the multiplied arguments
 */
int mult(int y, int z)
{
    return y * z;
}
```

It's a good idea to be quite verbose about what your functions are doing, comments describing exactly what a function does (but not how it does it), along with what the arguments are and what a function returns are very useful when you've forgotten what's going on. Developing a standard approach to this and making it a habit will dramatically improve your code readability.

Question 8: Hamming Weight

The hamming weight of a variable is the number of '1' bits in the variable. Write a function that calculates the Hamming weight of an integer.

Question 9: Memoryless Swap

Given the following code template, swap the values stored at `x` and `y` first with, then without a helper variable.

```c
#include <stdio.h>

int main()
{
    int x = 5;
    int y = 6;
```

```
    // Your code begins here

    // Your code ends here
    printf("%d %d\n", x, y);
    return 0;
}
```

Question 10: Programmable Circuits

NAND and NOR are universal operations, that is all other operations can be constructed using only these gates. Using only one of these operations acting on bits and some bit shifting, implement a byte-wise AND function.

What does this imply about the difference between a digital computer and an analogue computer?

```
#include <stdio.h>
#define BYTE char

////////////////////////////////////////////////////
// Function Declarations
////////////////////////////////////////////////////

/*
 * BYTE_AND
 * Performs a byte-wise AND on the two inputs
 * :: BYTE a :: The first byte of input
 * :: BYTE b :: The second byte of input
 * Returns the bytewise AND of a and b as a byte
 */
BYTE AND(BYTE a, BYTE b);

/*
 * BYTE_NAND
 * Performs a bitwise NAND on the target bits of the two inputs
 * :: BYTE a :: The first byte of input
 * :: BYTE b :: The second byte of input
 * :: BYTE bit_a :: The target bit of byte a
 * :: BYTE bit_b :: The target bit of byte b
 * Returns the NAND of the target bits of a and b
 */
BYTE NAND(BYTE a, BYTE b, BYTE bit_a, BYTE bit_b);

/*
 * BYTE_NOR
 * Performs a bitwise NOR on the target bits of the two inputs
 * :: BYTE a :: The first byte of input
```

```
 * :: BYTE b :: The second byte of input
 * :: BYTE bit_a :: The target bit of byte a
 * :: BYTE bit_b :: The target bit of byte b
 * Returns the NOR of the target bits of a and b
 */
BYTE NOR(BYTE a, BYTE b, BYTE bit_a, BYTE bit_b);

////////////////////////////////////////////////
// Function Definitions
////////////////////////////////////////////////

int main()
{

    printf("%d && %d = %d\n", a, b, AND(a, b))
    return 0;
}

/*
 * BYTE_AND
 * Performs a byte-wise AND on the two inputs
 * :: BYTE a :: The first byte of input
 * :: BYTE b :: The second byte of input
 * Returns the bytewise AND of a and b as a byte
 */
BYTE BYTE_AND(BYTE a, BYTE b)
{
    // This function should only contain shifts and either
    // BIT_NAND or BIT_NOR

    return 0;
}

/*
 * BYTE_NAND
 * Performs a bitwise NAND on the target bits of the two inputs
 * :: BYTE a :: The first byte of input
 * :: BYTE b :: The second byte of input
 * :: BYTE bit_a :: The target bit of byte a
 * :: BYTE bit_b :: The target bit of byte b
 * Returns the NAND of the target bits of a and b
 */
BYTE BIT_NAND(BYTE a, BYTE b, BYTE bit_a, BYTE bit_b)
{
    // Write your NAND function here
    return 0;
}
```

```c
/*
 * BYTE_NOR
 * Performs a bitwise NOR on the target bits of the two inputs
 * :: BYTE a :: The first byte of input
 * :: BYTE b :: The second byte of input
 * :: BYTE bit_a :: The target bit of byte a
 * :: BYTE bit_b :: The target bit of byte b
 * Returns the NOR of the target bits of a and b
 */
BYTE BIT_NOR(BYTE a, BYTE, b, BYTE bit_a, BYTE bit_b)
{
    // Write your NOR function here
    return 0;
}
```

Extend this to perform addition using just your NAND or NOR gate, you might want to implement some intermediary gates as helpers functions first.