THE UNIVERSITY OF SYDNEY

# COMP2129 Week 6 Tutorial

**GDB, fork, exec, unions and bitfields**

## Coding Style

Coding style comes under the larger umbrella of code readability and maintainability. Correct code may be poorly written, meaning that it does what it's supposed to, but is hard to read and debug in the future. There is no one best way to write code, but there are lots of examples of what *not* to do when writing code. If in doubt, show your code to other people: if they have trouble following it, there's probably something you can do to make it easier to read.

Indentation should be consistent and logical. Any code that comes after a function definition, `if`, `for`, `while`, `switch` and so on should be idented, so that other people (and you) can see the logical flow of your code.

There should be no "magic numbers", i.e. unnamed constants in your code. This is especially important if people are trying to read your code: a name like `two_pi` makes a lot more sense than the constant `6.28319` when used in your code, especially when used in calculations: compare `diameter * two_pi` with `diameter * 6.28319`. You can create named constants using the preprocessor or constant variables:

```
1 // Using the preprocessor
2 #define two_pi (6.28319)
3
4 // Using constant variables
5 const double two_pi = 6.28319;
```

You should avoid repeated code. If you find yourself writing out very similar code to something else in your program, or copy-pasting code from other parts of your program, you should generally refactor this code into a function which can be called. This allows any bugfixes or improvements to that code to be done in one place, rather than spread across your program.

Choose variable names well. Loop variables are usually `i`, `j`, and `k` – using anything but these for loops should be well-justified. The longer a variable spends in your program, the more time you should take trying to choose a good name for it. Well-named functions and variables make your code self-documenting: you won't have to write as many comments. Keep in mind, however, that C is a terse language. It's much more in the style of C to name a temporary variable `tmp`, rather than `tmpIntVar` or anything else.

Any other questions you have can be directed at a tutor, who will be able to glance at your code and point out any obvious styling mistakes. The Week 7 EdX lectures have a section on Programming Style you can refer to.

# The GNU Debugger

Debugging programs is much easier with the help of debuggers. You might be familiar with the one built into Eclipse (for debugging Java), or perhaps the ones built into web browsers, for debugging Javascript. The GNU Debugger (GDB) is a tool which was originally designed for debugging C (but has been extended to many more languages). GDB does lots of really neat stuff under-the-hood to attach itself to running instances of programs, to be able to pause them and peek inside. It can be invaluable for debugging issues in large programs, when simply inserting print statements just aren't cutting it anymore.

# Exercise 1: Memory layout and stack frames

In this exercise, we will use GDB to investigate the *call stack* of a recursive function. The function, of course, is the factorial function: $n! = 1 \times 2 \times 3 \times \cdots \times n$. It has a neat recursive definition, by defining a base case $0! = 1$, and a recursive case $n! = n \times (n-1)!$ for numbers $n \geq 1$.

Write the following C-file:

```c
#include <stdio.h>

int factorial(const int n) {
  int result = 1;
  if (n > 0)
    result = n * factorial(n - 1);
  return result;
}

int main(void) {
  printf("%d\n", factorial(4));
  return 0;
}
```

Compile the program with debugging symbols (option **`-g`**), then execute the program in **`gdb`**:

```
$ gcc -g -o factorial factorial.c
$ gdb ./factorial
```

Now you should be in the **`gdb`** shell. Set a breakpoint within the recursive function:

```
(gdb) break factorial
(gdb) run
(gdb) cont
(gdb) cont
(gdb) cont
```

Inspect the stack frames with **`bt`** (backtrace) and investigate the local variables in each frame using the instruction **`frame <number>`** in conjunction with **`print n`** and **`print result`**.

Try removing the base case from your recursive function, and launching it with and without GDB.

# Exercise 2: Fork

UNIX systems provide the **`fork()`** system call, which allows existing processes to create new ones. The new process created by **`fork`** is called the *child process*, which means the original one is commonly referred to as the *parent process*. **`fork`** is called once, but returns twice: the parent is returned the *process id* (**`pid`**) of the child, and the child is returned 0. The return value of fork is the only immediate way of distinguishing between parent and child.

Both the child and parent continue executing with the instruction immediately following the call to **`fork`**. The child is a copy of the parent, meaning the child gets a copy of the parent's stack, heap, and data space. (This is not wasteful: if you are interested on seeing how this actually works, look up "copy-on-write").

In this exercise, we will investigate how to use **`fork()`**. The following code is available on eLearning and Piazza, called **`forkdemo.c`**. Be sure to check out the manpage: **`man 2 fork`**.

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {
  int n = 6;
  pid_t pid;

  printf("Before fork\n");
  pid = fork();
  if (pid < 0) {
    perror("Fork failed");
    return 1;
  }

  if (pid == 0) { // Child
    n++;
    //sleep(1);
  } else {        // Parent
    n *= 2;
    //wait(NULL);
  }

  printf("Fork returned %d, n is %d\n", pid, n);
  return 0;
}
```

- Compile and run the program a few times. Do the lines always get printed out in the same order?
- Uncomment the **`sleep`** function so that the child sleeps (does nothing) for a second. Now try running the program.
- Also uncomment the **`wait`** function so that the parent waits for the child to exit before proceeding past the **`wait`** function. Now try running the program.

# Exercise 3: Processes on UNIX

If you're trying to debug a program that uses **fork**, you'll eventually want to look at running processes. UNIX systems provide the **ps** command for this. By default (and it varies slightly depending on what system you're on), **ps** will only show you processes launched by the current terminal or shell.

- Modify your previous **forkdemo.c** code so that the child exits a long time after the parent.
- Run this program: you should be returned to the shell almost immediately, without seeing the child's output.
- Run **ps** to locate the child process, and its process ID.
- Use **kill <process id>** to kill the child process before it prints anything to the screen.

# Exercise 4: Exec

The **exec** family of functions allow a process to start an executable file. However, when a process calls **exec**, it is completely replaced by the new process. The process ID of the process does not change. Usually a call to **fork** is followed by a call to **exec**, and pipes are used to communicate between the parent process and the new process. Here we'll just use **exec**, and introduce pipes later on.

Use **exec** to invoke the sorting program **/usr/bin/sort** to sort the lines in the previous program, **forkdemo.c**. The function used is called **execl**, which you can look up the manpage for: **man 3 exec**. **execl** is given the path to a program to execute, then the contents of **argv[]**, terminated by a null pointer.

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

int main(void) {
  printf("About to launch /usr/bin/sort\n");

  if (execl("/usr/bin/sort", "sort", "forkdemo.c", (char *)NULL) < 0) {
    perror("Exec failed");
    return 1;
  }

  printf("Sort finished.\n");
  return 0;
}
```

- Compile and run the above code. Does it do what you expect? Compare your output to running **sort forkdemo.c** in the shell.
- Modify the code to launch another executable: you can find an executable's path by using the **which** command in the terminal, for example **which sort**. Try to pass in multiple arguments on **argv**.

# Multidimensional Arrays

C has support for multidimensional arrays declared at compile time. For example, a $3 \times 3$ matrix can be declared as `int matrix[3][3]`. It also allows you to access the rows and columns easily, for example `int matrix[i][j]` will access the $i$th row and the $j$th column. Since the compiler knows the sizes at compile time, it can do the proper indexing for you. The underlying data structure is just a regular array (containing `int`s in this case).

Arrays declared in this way are stored in row-major order, meaning that if you start at the element `matrix[0][0]` in memory and start reading, you will read the first row, then the second, and so on.

If you want to have a grid which is sized at run-time (for example, if you are processing images), then you'll have to do the indexing yourself. A common pattern is to still use row-major order, but now take over your own indexing. The following is an example of allocating space for, and accessing, a $4 \times 3$ grid (which in principle, could be any size).

```
1  int width = 4, height = 3;
2  int *grid = malloc(width * height * sizeof(int));
3
4  // Set the third row, second column to 1
5  grid[width * 2 + 1] = 1;
```

Which one to use depends on the use case. For example, 3D graphics typically uses a lot of $3 \times 3$ and $4 \times 4$ matrices, so for these it makes sense to have them statically declared at compile time. However, it also uses many variably-sized grids (like for representing textures, the computer screen, and so on), for which it makes more sense to have dynamically sized grids.

# Exercise 5: Matrix-vector multiplication

Write a function which multiplies $3 \times 3$ matrices with vectors of length 3. Your function should take three arguments: `matrix`, the matrix $M$, `ivector`, the vector $\mathbf{v}$, and `ovector`, where you should write the product $M\mathbf{v}$. The prefixes `i` and `o` are used to make more clear which is the input and which is the output vector.

```
1  // Write the product (matrix)(ivector) to ovector
2  void matrix_vector_mult(int matrix[3][3], int ivector[3], int ovector[3]);
```

Here is an example of a matrix, a vector, and their product:

$$M = \begin{bmatrix} 1 & 0 & 1 \\ 0 & -1 & -2 \\ -1 & 1 & 0 \end{bmatrix} \quad \mathbf{v} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \quad M\mathbf{v} = \begin{bmatrix} 2 \\ -3 \\ 0 \end{bmatrix}$$

There is some code available on eLearning and Piazza, `matrix.c`, which contains routines for pretty-printing matrices and vectors. Looking at their code might help you with matrix multiplication.

Note: we've used `int`s here to make everything print out nicely. If you want to use matrices containing real numbers in a program, it's better to have them of type `double`.

# Exercise 6: Two's complement

When numbers are represented in two's complement, their most significant bit plays a special role: it is negated. For example, the number $1000\,0011_2$ is interpreted as $-128 + 2 + 1$, rather than what it usually is $(128 + 2 + 1)$. This exercise constructs a `union` to take apart these numbers, to view the most significant bit separately to the rest of the number. The following table shows some examples of numbers (the **`msb`** and **`rest`** columns correspond to the union below):

| Binary | Interpretation | Result | **`msb`** | **`rest`** |
|---|---|---|---|---|
| $1000\,0011_2$ | $-128 + 2 + 1$ | $-125$ | 1 | 3 |
| $1111\,1111_2$ | $-128 + 64 + 32 + 16 + 8 + 4 + 2 + 1$ | $-1$ | 1 | 127 |
| $0000\,0011_2$ | $2 + 1$ | 3 | 0 | 3 |
| $1010\,0000_2$ | $-128 + 32$ | $-96$ | 1 | 32 |

The union below stores an `int8_t` (a signed 8-bit number) in the same place as a struct made of two bitfields of length 1 and 7. This allows us to view the union as a regular number, using the **`.val`** member, or as broken into its most significant bit **`.parts.msb`** and the rest of the number **`.parts.rest`**.

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

union int8bits {
  int8_t val;
  struct {
    uint8_t rest : 7;
    uint8_t msb  : 1;
  } parts;
};

int main() {
  union int8bits ibits;
  for (ibits.val = -3; ibits.val <= 3; ibits.val++) {
    printf("Value: %4d, First: %d, Rest: %3d\n",
        ibits.val, ibits.parts.msb, ibits.parts.rest);
  }

  return 0;
}
```

- The code above is available on eLearning and Piazza as **`twoscomp.c`**. Download, compile, and run it, and make sure you can make sense of its output.
- Write a function which takes a `int8_t`, and uses the union to turn on the most significant bit. Confirm that this does nothing to negative numbers, but subtracts 128 from nonnegative numbers.
- Print out the **`msb`** and **`rest`** of the number 127, before and after adding 1 to it. Why do you think computers use two's complement representation?