

CPSC213 Environment Setup

Hello everyone and welcome to CPSC213! Since you're enrolled in a UBC CS course, you now have access to the school's Linux servers. We recommend using those to do your work rather than your local machine since there may be a difference in dependencies. This post will be a quick guide for getting you set up on that.

There aren't any assignments due for a while, so there's no need to do this immediately: if you don't feel confident, you can wait until office hours start and get help from a TA there.

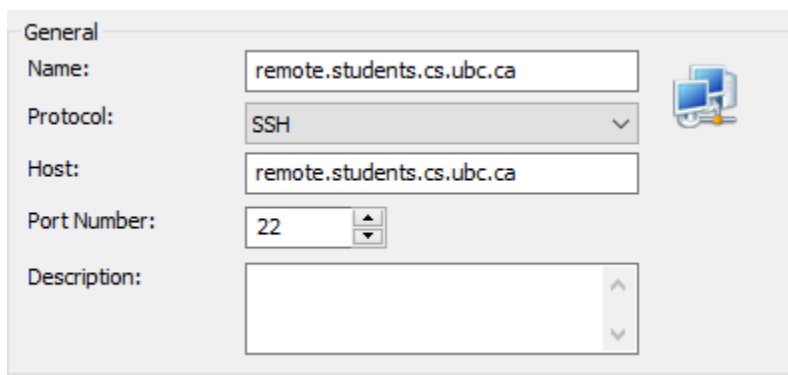
Accessing the Remote Servers

The remote servers are basically computers you can access over a network connection - you have your own allotment of memory there and can run some commands. You'll be accessing them through an SSH (secure-shell) connection - the goal is to open a terminal on the remote servers so you can input commands to compile your code, etc.

[MacOS Users]: You can open a remote shell using the Mac terminal directly. Use the command `ssh [yourCWLhere]@remote.students.cs.ubc.ca` to do so. You can use CyberDuck to move files over ssh as well - the process should be similar to the steps below for Windows users.

[Windows Users]:

- Go to this site <https://my.cs.ubc.ca/docs/free-terminal-emulation-software-xmanager>, download the XManager installation exe and install it.
- Launch either XShell or XFTP (this set of steps will have to be done for both anyways). If you're asked for a license key, you'll find it on the site you downloaded the installation exe from.
- Add a new session and fill out the form:



The screenshot shows the 'General' tab of a configuration window. It contains the following fields:

- Name:** remote.students.cs.ubc.ca
- Protocol:** SSH (selected from a dropdown menu)
- Host:** remote.students.cs.ubc.ca
- Port Number:** 22 (selected from a spinner box)
- Description:** (empty text box)

There is a small icon of a computer monitor with a plug symbol to the right of the Protocol dropdown.

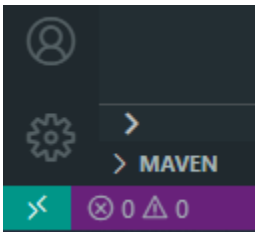
- You will be asked to login when you try to connect: use your CWL credentials.
 - That should be it! You should now have a shell or file explorer window that is directly connected to UBC servers. Any commands you execute (try `echo Hello World!`) will be done on the remote servers and you'll be able to transfer files to and from those servers.
 - Repeat the above steps for the other app and you'll be all set up!
-

Coding Remotely

Setting up remote development with your text editor makes for a better experience: you won't have to shuffle as many files around. While there are several text editors that support remote development, this guide will only go over setting it up for Visual Studio Code.

To start, open the extensions menu (Ctrl+Shift+X) and search for SSH. You should find one by Microsoft named "Remote - SSH".

- Downloading the extension will add a small button to the bottom left of your window



- Click on it and select either of the options to connect to Host.
- Select the Add SSH host option (you might also be prompted to do so directly) and input the host (network) you want to connect to: `ssh [yourCWLhere]@remote.students.cs.ubc.ca`
- VS Code will ask for a location to save the SSH connection info. Just hit Enter to save it in the default location.
- When prompted for the OS of the server you're SSH-ing into, select Linux.

That should be it! VS Code should now be connected to the remote servers, meaning that you can write and edit code stored there. Ctrl+` will open a remote terminal instance as well. As a test, try transferring a text file to the remote servers via XFTP/CyberDuck and see if you can edit it in VS Code!

Setting up SSH Keys (Optional)

You can avoid inputting your password on every connection by generating an SSH key and saving it to the remote servers. Keep in mind this process is device-specific - you'll need to do it again for any extra devices you want to connect to the remote servers with.

1. On your local device, open a PowerShell terminal (Windows). Mac users can just open a regular terminal.
2. Generate a sshkey by using the command `ssh-keygen`. If you've generated a ssh key before in the past, make sure you name the key something else (include the entire pathname) so it doesn't overwrite your previous one. Otherwise, just keep hitting Enter to accept the defaults. Feel free to include a password if you want to.
3. Assuming you went through all the defaults, paste the following command in your local terminal:

```
cat ~/.ssh/id_rsa.pub | ssh [yourCWLID]@remote.students.cs.ubc.ca 'cat  
>> ~/.ssh/authorized_keys'
```

If you've named the key differently, just make the appropriate changes to the command above. All it does is copy the contents of `id_rsa.pub` into a file called `authorized_keys` on the remote servers.

That should be all you need to get setup with SSH keys! You'll still need to use your SSH-key password (if you set one) when opening a remote shell, but editing code on VS Code shouldn't require your CWL password now.

Windows Subsystem for Linux (*OPTIONAL!*)

This is for Windows users only: you can optionally choose to install WSL - this lets you use Linux commands while still having access to your Windows files and data! Here's the official documentation: <https://docs.microsoft.com/en-us/windows/wsl/setup/environment>

Linux is generally preferred over Windows for development work, so I recommend getting used to it now, but it is by no means necessary.

If you have any questions, feel free to ask in a follow-up discussion! I'll try my best to help, although going to in-person office hours will probably be more helpful. Good luck!

Quick Explanation of .c /.h files

Hey, it seems that a lot of people have been having trouble with debugging C compiler errors, and I believe a lot of that is a result of not knowing how .h and .c files really work. So, this should hopefully be a good rundown of what's going on. You don't need to know all of this for the class but I think it's still helpful to know.

First thing's first, the C compiler is very primitive. It will start from the top of the file and read to the bottom of the file. It doesn't know a function exists until it sees a declaration for that function. This leads to some interesting effects:

```
void func_a() {  
  
}
```

```
void func_b() {  
    func_a();  
}
```

This is legal. This is because by the time the compiler gets to func_b, it has already seen func_a, and therefore knows func_a exists. However, *this*, is illegal:

```
void func_a() {  
    func_b();  
}
```

```
void func_b() {  
  
}
```

This is because by the time the compiler gets to func_a, it hasn't seen func_b before, and thinks you're calling an undefined function. And so, it will raise a compiler error. However, if you for some reason need func_a to be above func_b in the file, you can bypass this by doing the following:

```
// This is called a forward declaration. This promises the C compiler that  
func_b exists, and that the function body is stored somewhere  
void func_b();
```

```
void func_a() {  
    func_b();  
}
```

```
void func_b() {  
  
}
```

C programmers correctly determined that this was a very silly limitation, and created the concept of a header file. First, a quick digression. Let's say you have a program called `foo.c`. And let's say you include `bar.h` in that file, like so: `#include <bar.h>`. When you do a `#include`, you are telling the compiler to copy paste the contents of the file `bar.h` in place of the `#include`. It doesn't copy the `bar.c` file's contents, it *only* copies the `bar.h` file's contents.

So, what C programmers began doing is creating a header file that contains forward declarations for all the functions they want to use, and they include the header file in all the files that need to use the function. This neatly allows you to not have to worry about where functions are relative to each other in the file. (Everything I've said so far applies not only for functions, but structs, global variables, and basically everything in C).

Now, let's talk about the compiler. The compiler (`gcc`)'s job is very simple. Take every function / struct / etc. in the C file, and generate assembly code for it. The compiler puts the generated code inside of an object file, one for each C file. These are the `.o` files you see when you compile your program. The compiler compiles each file in isolation. Each C file is compiled on its own to an object file. Once all the object files are generated, a separate program, the linker (`ld`), will combine all the object files into one executable.

So, why am I telling you all this? I think the two most important takeaways from this read are:

1. The compiler reads things from the top of the file to the bottom of the file. Thus, you need to define things before you use them. If you find that you can't rearrange things in this order, you'll need to add a *forward declaration*.
2. If you ever see an error like (for example) `"/bin/ld: Undefined reference to my_function"`, that's the linker complaining that it can't find assembly code for `"my_function"` in the object files. This typically means that you didn't include the C file with the body of `my_function` in your makefile, or that you never gave the function a body in any C file.

Hope this helps, and happy debugging!

How to use GDB

GDB Debugging

This is an add-on manual in addition to PL on using GDB. A lot of the following can found by just a simple Google. A brief manual can be viewed by just simply typing the following into the terminal.

```
man gdb
```

In gdb, you can type `help` to view instructions on how to use it or a specific function.

```
# e.g. view all commands on running
help running
```

When using GDB or any other debugger, it is curial to retain debug symbols at compile time and do not let the compiler perform any optimization. To do this, add the following 2 flags.

```
gcc -O0 -g <some c files> -o <executable>
```

Basic Commands

Stepping and Breakpoints

There are couple of ways to break/run your program:

- `breakpoints`: stop at the point you specified
- `step in`: step into a function call
- `step out`: continue current function and stop at the function's return address
- `step over`: Skip over. Treat the function that you are about the step in as one statement. Equivalent to instruct step-in and step-out together.
- `continue`: continue and stop at the next breakpoint

Simple Breakpoints can be set by:

```
b [file:][function/line]
```

```
# e.g.
# file example.c content:
# void buggy() {}
b exaple.c:buggy
```

Conditional breakpoints are useful if you have a huge loop (you can attach/edit/delete the condition):

```
b [file:][function/line] if [exp]
# [exp] can be any valid C exp that evaulates to an integer/bool
# e.g. strcmp(s, "abc") == 0
```

```
# e.g. to break at line 3 only if i == 7
# file example.c content:
# void buggy() {
#     int i = 100;
#     while (i) {
#         i--;
#     }
# }
```

```
b exaple.c:3 if i == 7
```

Step-[in/out] and continue:

```
# step in:
step      # abbreviated: s
# step over:
next      # abbreviated: n
# step out:
finish    # abbreviated: fin
# continue:
continue  # abbreviated: c
```

[For more](#) related to stepping. It is highly recommended that you have your source code opened in another window. Or if you need,

```
# prints out the context of the source code
list      # abbreviated: l
```

Print/Watch a variable

You can monitor a variable by printing it.

```
print -flgs [expr]
```

Type `print -` and hit `TAB` key gives some hints on what flags could be used. They can be used to print numbers in hex for example. Additionally, `printf` is also available. You can also instruct GDB to watch a variable for change. Search `GDB print` for more only basic and most used commands are shown below.

```
set print pretty on
# pretty printing
```

```
print [expr]
# [expr] can be any valid C expression
p [expr]
# shorthand
```

```
# print a var
# int a = 90;
print a
```

```
# print a string
# char* str = "abcd";
print str
```

```
# print a pointer
# char b = 'b';
# char* c = &b;
print c
print *c
```

```
# print array
# int* a = {1, 2, 3};
print *a@3
# @ denotes the length. Again, no bounds checking
```

```
# print struct
# struct Point {
#     int x;
```

```
#      int y;
#};
# struct Point p = {10, 9};
print p
print p.x
```

Seg-Fault

GDB

Unfortunately, GDB is not guaranteed to reproduce what you see in the terminal. You might see your program runs fine in GDB but seg-faults when in terminal.

Example C file:

```
#include<stdio.h>

int* bug(int i) {
    return NULL;
}

int bar() {
    int a = 10;
    int c = *bug(a);
    return c;
}

void foo() {
    int b = 10 + bar();
    printf("%d", b);
}

int main() {
    foo();
    return 0;
}
```

In GDB, hit run, you should see:

```
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
```

```
Program received signal SIGSEGV, Segmentation fault.
```

```
bar () at test.c:9
9          int c = *bug(a);
```

Type in `backtrace` or `bt` to trace the stack:

```
#0 bar () at test.c:9
#1 0x0000555555555198 in foo () at test.c:14
#2 0x00005555555551cc in main () at test.c:19
```

Again, type `bt` - followed with `TAB` shows you some options. Most useful one:

```
bt -full
# prints the stack with local variables
```

Sanitizer

Sanitizer is a more robust way to detect all kinds of memory problems. It injects code at compile time to check any memory issue including

- used after free
- memory leak
- uninitialized values
- undefined

To enable it, add the following flags to compiler and run your executable.

```
clang -fsanitize=address -O0 -g <.c files> -o <executable>
```

Instead of clang, GCC will also work here. Clang can't detect memory leaks on macOS platform.

Search `clang sanitizer` for more.

Valgrind

Just add following flags when using valgrind to emit more useful information.

```
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes
```

Search `valgrind trace` for more.

SM213 Assembly Syntax Highlighter for VS Code

Hellooo

If you're using VS Code to write assembly for assignments, you might notice that it looks a little "boring" and "plain" – where are the colours??

If you want to bring a little more colour back into your life, you can try installing [this extension](#)! It aims to provide syntax highlighting for the assembly language we use in this course. The effects of this extension are purely aesthetic, but could make staring at your assembly code a little more enjoyable.

It's also [open source](#) so feel free to make/suggest changes!