C++ Fundamentals
Setup Guide

During the course we will be writing and debugging a series of C++ programs. To do that we need an editor and a debugger. On Unix there are many editors to choose from, vi, vim, emacs, pico etc and from the early days of Unix there have been a number of debuggers available. In the mid 80s gdb was modelled after dbx (an earlier debugger) and is now the principle debugger used in Unix systems.

Integrated Development Environments or IDEs combine editors and debuggers into a single package. IDEs often provide additional facilities, but the editor and debugger remain the principle components. Since 2000, the Eclipse IDE has been very popular on Unix systems. Eclipse employs a series of plug-ins to provide support for various languages (C, C++, Perl, Pyton, Rust etc). The C/C++ plugin-in is called CDT, so Eclipse/CDT was the IDE of choice for a number of years.

However with the introduction of the STL into C++ it became necessary to provide "pretty printers" to display the contents of the STL containers. The pretty printers are written in Python and can be used with a Python aware version of gdb. Unfortunately, CDT has been bedeviled with problems using the pretty printers. In short, the pretty printers work in gdb, but often fail in CDT. This has led a number of developers to try other IDEs.

JetBrains have produced an excellent IDE called CLion, but it is not free. In the last couple of years Microsoft have released "Visual Studio Code" or vscode, based on the similarly named product "Microsoft Visual Studio" which is sold on Windows.

"Microsoft Visual Studio" on Windows has a community edition which is free, but it only works on Windows. However, vscode works on Windows, MacOS and Linux and is free. I suspect, vscode will become the dominent IDE on Unix in the near future. The editor part of the IDE is basically the same as its Windows counterpart, but on Linux, vscode uses gdb or lldb as a debugger. The pretty printers must be installed in gdb, but then vscode works perfectly "out of the box".

Recently, another IDE based on Typescript and Python has been released by Chad Smith (of Facebook) called "gdbgui". This provides an excellent front end to gdb, but its editor is read only. Nevertheless, gdbgui is an excellent product and easily installed with Python. Well worth a try.

Other front ends to gdb are availble, see below.

1. Downloading Examples

All the examples for the course are stored on github. You can clone the repository with:

```
git clone https://github.com/seddon-software/cpp.git
```

2. Using a text editor and gdb-tui

This is probably the simplest setup and one I sometimes use on the course. You can use your favourite text editor (mine is **vim**) and **gdb-tui** for debugging. The following module loads are required:

```
module load gcc/9.2.0
module load python/3.7
```

You can debug with **gdb** alone, but a windows based gdb is much easier to use. There are two popular ways to achieve this: **gdb-tui** and **gdb-gui**. In both cases you need to do a little setup, because gdb doesn't support debugging STL structures out of the box. Instead you nee to setup pretty-printers using the gdb's Python plugin. You can find instructions on how to do this on the web, but Diamond have made it a little easier by developing a Python script which is added to the start of the gdb initialization script.

We will have a short overview of how to compile C++ programs on the course itself, but to debug code you will need to do the following:

a) write a gdb initialization file for use with gdb-tui. The file should be in you home directory and called **.gdbinit**:

```
# add pretty-printers
add-auto-load-safe-path /dls_sw/apps/gcc/9.2.0/7/lib64/libstdc++.so.6.0.27-gdb.py

# output terminal for gdb-tui
tty /dev/pts/25          # you'll have to change this line

# gdb commands
break main
run

# hook to refresh screen after stepping
define hook-next
  refresh
end
```

Note that gdb-tui tends to scroll the screen which makes it difficult to see what is going on. To overcome this you can refresh the screen after every operation. However it is better to add a hook (as in the above) to do this automatically. Further, any printed output (from cout) will mess up the screen, so it is best to send output to another terminal. In the example above I've used:

```
/dev/pts/25
```

but you'll need to change that to your own pseudo terminal. You can find out the name of any pseudo terminal with the command:

```
tty
```

b) You can now invoke gdb-tui with the following command:

```
gdb -q -tui -x $HOME/.gdbinit --args myfile.exe
```

where **myfile.exe** is your compiled code.

3. Using Eclipse/CDT on a Diamond machine

Eclipse has been around for a long time and can be used for many programming languages. To work with C++ you need the CDT plugin. However, many people prefer to use the more modern Visual Studio Code or Clion. If you want to gice Eclipse/CDT a try at Diamond, you use module loads to setup Eclipse. The debugger uses a Python plug-in, so several module loads are required:

```
module load eclipse/201812_20181219
module load gcc/9.2.0
module load python/3.7
```

Make sure you don't have two versions of eclipse installed. To see what is loaded type:

```
module list
```

If you see "eclipse/473a(default)", ir something similar, you will need to unload it:

```
module unload eclipse/473a
```

Note that all module loads only apply to the current terminal window. If you create a new window you will need to type these commands in again (unless you add the commands to your .bashrc file).

You can now run Eclipse with:

```
eclipse&
```

The current version of **CDT** doesn't work properly with pretty-printers, even though **gdb** works fine.

4. Using vim for development

Modern versions of vim allow you to use plug-ins to enhance the way vim works. There are many plug-ins available. One standard plugin is used for debugging, see:

```
https://www.dannyadam.com/blog/2019/05/debugging-in-vim/
```

Using the **termdebug package** plug-in is very similar to to using **gdb-tui**. If you are interested in other plugins then check out:

```
https://vim.fandom.com/wiki/Use_Vim_like_an_IDE
```

5. Using Visual Studio Code

Visual Studio Code (not to be confused with Visual Studio IDE) is a free IDE from Microsoft and works well on all operating systems, including Linux. The studio doesn't directly support **make**, but this is easily remedied. You can read up on how to use Visual Studio code on your own machine (you need to add C++ extensions), but Diamond already has a version available with:

```
module load vscode/1.42.1
```

You can run vscode with:

```
code
```

You should then go to the Extensions icon and add the C++ extensions module.

Note that vscode requires that you set up 2 JSON scripts (which I've included in the files on github): one for building code and one for running code. However I've found you can dispense with the build script and just use **make** in the terminal that is built into Visual Studio code. Alternatively, it isn't very difficult to develop a JSON script for make.
Here is the script I'll use on the course:

tasks.json

```
{
   // See https://go.microsoft.com/fwlink/?LinkId=733558
   // for the documentation about the tasks.json format
   "version": "2.0.0",
   "tasks": [
      {
         "label": "make",
         "type": "shell",
         "options": {
            "cwd": "${fileDirname}"
         },
         "command": "make",
         "args": [],
         "problemMatcher": [],
         "group": {
            "kind": "build",
            "isDefault": true
         }
      },
      {
        "label": "master make",
        "type": "shell",
        "options": {
           "cwd": "${workspaceFolder}/_Build"
        },
        "command": "make",
```

```
        "args": [],
        "problemMatcher": [],
        "group": {
            "kind": "build",
            "isDefault": true
        }
    },
    {
      "label": "master make clean",
      "type": "shell",
      "options": {
          "cwd": "${workspaceFolder}/_Build"
      },
      "command": "make",
      "args": ["clean"],
      "problemMatcher": [],
      "group": {
          "kind": "build",
          "isDefault": true
      }
    },
    {
          "label": "make clean",
          "type": "shell",
          "options": {
              "cwd": "${fileDirname}"
          },
          "command": "/usr/bin/make",
          "args": [
              "clean"
          ],
          "problemMatcher": []
      }
  ]
}
```

The default launch (debug) script needs to customised to work with my example files. The script has to be called “launch.json”:

launch.json

```
{
   "version": "2.0.0",
   "configurations": [
      {
         "name": "(gdb) Launch",
         "type": "cppdbg",
         "request": "launch",
         "program": "${fileDirname}/${fileBasenameNoExtension}.exe",
         "args": [],
         "stopAtEntry": true,
         "cwd": "${workspaceFolder}",
         "environment": [],
         "externalConsole": false,
         "MIMode": "gdb",
         "setupCommands": [
            {
               "description": "Enable pretty-printing for gdb",
               "text": "-enable-pretty-printing",
               "ignoreFailures": true
            }
        ]
      }
   ]
}
```

Both json files are stored in the **.vscode** subdirectory of **_Build**.

6. Using CLion

CLion from JetBrains is a new and popular addition to the set of IDEs you can use for C++ development. Unfortunately **CLion** is not free, but you can try an evaluation copy (30 days) by downloading from the JetBrains site or using the evaluation copy installed at Diamond. To use the Diamond evaluation copy use the following module load:

```
module load clion/2019-2
```

and then

```
cd /dls_sw/prod/tools/RHEL7-x86_64/clion/2019-2
clion.sh
```

7. Using gdbgui

You can even experiment with this Python module. Install with:

```
pip install gdbgui --user
```

gdbgui is a browser-based frontend to gdb. Use the normal gdb commands, just inside a browser. Start the program with:

```
gdbgui
```

I've been very impresses with this program. Well worth a try.