TRƯỜNG ĐẠI HỌC BÁCH KHOA – ĐẠI HỌC QUỐC GIA TP.HCM
TRUNG TÂM KỸ THUẬT ĐIỆN TOÁN
268 Lý Thường Kiệt, Phường 14, Quận 10, TP.Hồ Chí Minh
Điện thoại: 84-8-3864 7256 ext 5371 – Fax: 84-8-3865 8687
Website: **www.cce.hcmut.edu.vn**
E-mail: **dientoan@hcmut.edu.vn**

Lab 2
Linux Programming with C++

MSc.Hoàng Lê Hải Thanh
High Performance Computing Laboratory, HCMUT, VNU-HCMC
thanhhoang@hcmut.edu.vn

I. GOALS

- Familiar with C++ development on Linux using GNU Toolchain

II. OBJECTIVES

- Know the history and variants of the GNU C Compiler (GCC)
- Understand the fundamental of the C++ compilation process
- Be familiar with GNU Make in modern software development
- Practice debugging C++ programs using the GNU Debugger

III. PREREQUISITES

- Make sure your OS has the GCC installed by using the following command:

```
~$ g++ --version
g++ (GCC) 4.8.5 20150623 (Red Hat 4.8.5-44)
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

- If GCC is not installed in your environment, please install using this link: https://linuxize.com/post/how-to-install-gcc-compiler-on-ubuntu-18-04/

- Basic C++ programming knowledge is also required for this lab.

IV. INTRODUCTION TO GNU C COMPILER

1. GCC History

The GNU Compiler Collection, commonly known as GCC, is a set of compilers and development tools available for Linux, Windows, various BSDs, and a wide assortment of other operating systems.

As its name, GCC primarily supports C and C++ and includes other modern languages like Objective-C, Ada, Go, and Fortran. The Free Software Foundation (FSF) wrote GCC and released it as free software.

GCC is a key component of the so-called "GNU Toolchain", for developing applications and writing operating systems. The GNU Toolchain includes

- GNU Compiler Collection (GCC): a compiler suite that supports many languages, such as C/C++ and Objective-C/C++.
- GNU Make: an automation tool for compiling and building applications.

- GNU Binutils: a suite of binary utility tools, including linker and assembler.
- GNU Debugger (GDB).
- GNU Autotools: A build system including Autoconf, Autoheader, Automake, and Libtool.
- GNU Bison: a parser generator (similar to *lex* and *yacc*).

GCC is portable and runs on many operating platforms. GCC (and GNU Toolchain) is currently available on all Unixes. They are also ported to Windows (by Cygwin, MinGW, and MinGW-W64).

GCC is also a cross-compiler, for producing executables on different platforms. GNU Toolchain, including GCC, is included in all Unixes. It is the standard compiler for most Unix-like operating systems.

Microsoft Visual C++, GNU Compiler Collection (GCC), and Clang/Low-Level Virtual Machine (LLVM) are three mainstream C/C++ compilers in the industry. In this course, we will use the standard GCC as it is the default compiler on Linux.

2. How GCC Works

a. GCC Compilation Stages

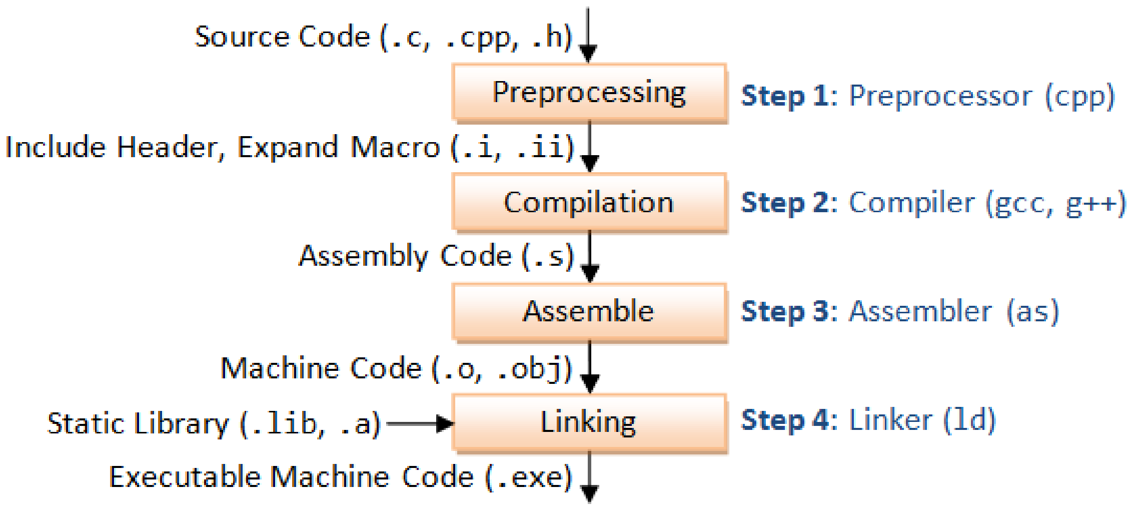
GCC is a toolchain that compiles code, links it with any library dependencies, converts that code to assembly, and then prepares executable files. It follows the standard UNIX design philosophy of using simple tools that perform individual tasks well.

First of all, what even is **an executable**? An executable is a special type of file that contains machine instructions (ones and zeros), and running this file causes the computer to perform those instructions. Compiling is the process of turning our C++ program files into an executable.

When we run GCC on a source code file, there are four steps before an executable file is made:

(1) **Preprocessing** is the first pass of any C compilation. It removes comments, expands #include files and macros, and processes conditional compilation instructions. This can be output as a *.i* file.
(2) **Compilation** is the second pass. It takes the output of the preprocessor, and the source code, and generates assembler source code. Assembly language is a low-level programming language (even lower than C) that is still human-readable but consists of mnemonic instructions that have strong correspondence to machine instructions.
(3) **Assembly** is the third stage of compilation. It takes the assembly source code and produces an object file, which contains actual machine instructions and symbols (e.g., function names) that are no longer human-readable since they are in bits.
(4) **Linking** is the final stage of compilation. It takes one or more object files or libraries as input and combines them to produce a single (usually executable) file. In doing so, it resolves references to external symbols, assigns final addresses to procedures/functions

and variables, and revises code and data to reflect new addresses (a process called relocation).


Source Code (.c, .cpp, .h)
Preprocessing
Step 1: Preprocessor (cpp)
Include Header, Expand Macro (.i, .ii)
Compilation
Step 2: Compiler (gcc, g++)
Assembly Code (.s)
Assemble
Step 3: Assembler (as)
Machine Code (.o, .obj)
Static Library (.lib, .a)
Linking
Step 4: Linker (ld)
Executable Machine Code (.exe)

Let's consider the following basic Hello World C++ program:

```
#include<iostream>

#define NAME "World"

using namespace std;

// File helloworld.cpp
// This is the main program
int main() {
	cout << string("Hello ") + NAME  << endl;
	return 0;
}
```

Normally, developers will compile and run this file directly using the following command:

```
~$ g++ -o helloworld helloworld.cpp
```

```
~$ ./helloworld
Hello World
```

Moreover, we can stop at any of four above compilation stage using corresponding options. Let's stop at the first step - **preprocessing** and discuss its output.

```
~$ g++ -E helloworld.cpp -o helloworld.i
```

```
~$ cat helloworld.i

…
  static ios_base::Init __ioinit;


}
# 2 "helloworld.cpp" 2



using namespace std;


int main() {
 cout << string("Hello ") + "World" << endl;
 return 0;
}
```

We can see that the preprocessor handles the preprocessor directives, like #include and #define. It is agnostic of the syntax of C++, which is why it must be used with care.

Next, let's check the output of the **compilation** step, which is the assembly format:

```
~$ g++ -S helloworld.cpp
```

```
~$ cat hello-world.s
       .file  "hello-world.cpp"
       .local _ZStL8__ioinit
       .comm  _ZStL8__ioinit,1,1
       .section      .rodata
.LC0:
       .string       "Hello "
.LC1:
       .string       "world"
       .text
       .globl main
       .type  main, @function
…
```

Then we will use the assembler (*as*) to convert the assembly code into machine code:

```
~$ as helloworld.s -o helloworld.o
```

```
~$ file helloworld.o
```

```
helloworld.o:  ELF  64-bit  LSB  relocatable,  x86-64,  version  1  (SYSV),  not
stripped
```

We can also see the detailed compilation process by enabling the *-v* (verbose) option:

```
~$ g++ -v -o helloworld helloworld.cpp
```

V. GNU MAKE

1. What is Makefiles?

Makefiles are used to help decide which parts of a large program need to be recompiled. In the vast majority of cases, C or C++ files are compiled.

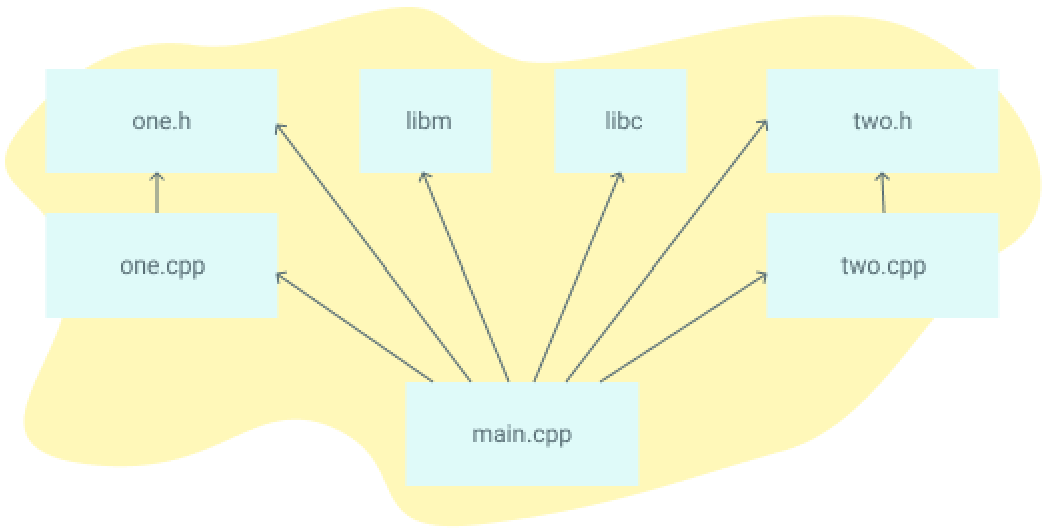
Since building large C/C++ programs usually involves multiple steps, a tool like Make is needed to ensure all source files are compiled and linked. Make also lets the developer control how ancillary files like documentation, man pages, systemd profiles, init scripts, and configuration templates are packaged and installed.

Make is not limited to languages like C/C++. Web developers can use GNU Make to do repetitive tasks like minifying CSS and JS, and system administrators can automate maintenance tasks. Additionally, end-users can use Make to compile and install software without being programmers or experts on the software they are installing.

Here's an example dependency graph that you might build with Make. If any file's dependencies change, then the file will get recompiled.

Make was initially developed by Stuart Feldman in April of 1976, but GNU released its free software version of the tool in the late 1980s. Version 3.56 is still available (via diffs) from the GNU FTP server, dating back to September 23, 1989.

GNU Make is a valuable tool for compiling software projects, especially those in the GNU/Linux landscape. Its simple Makefile syntax and intelligent processing of target files make it an excellent choice for development.



2. Makefile Syntax

A makefile is simply a text file (commonly named *Makefile*) that consists of a set of rules. A rule generally looks like this:

```
targets: prerequisites
	command
	command
	command
```

- The **targets** are file names, separated by spaces. Typically, there is only one per rule.
- The **commands** are a series of steps typically used to make the target(s). These need to start with a tab character, not spaces.
- The **prerequisites** are also filenames, separated by spaces. These files need to exist before the commands for the target are run. These are also called dependencies

3. Getting Started with Makefile

Let's start with a formal Hello World example. Use *nano* or another of your favorite text editors to compose a file named *Makefile* with the following content:

```
hello:
	echo "Hello, World"
	echo "This line will always print because the file hello does not exist."
```

Let's break it down:

- We have one target called *hello*
- This target has two commands
- This target has no prerequisites

We'll then run ***make hello***. As long as the hello file does not exist, the commands will run:

```
~$ make hello
echo "Hello, World"
Hello, World
echo "This line will always print because the file hello does not exist."
This line will always print because the file hello does not exist.
```

It's important to realize that *hello* is both a target and a file. That's because the two are directly tied together. Typically, when a target is run (aka when the commands of a target are run), the commands will create a file with the same name as the target. In this case, the *hello* target does not create the *hello* file.

Let's create a more typical Makefile - one that compiles our previous C++ file:

```
hello: helloworld.cpp
	g++ -o hello helloworld.cpp
```

When we run *make* again and again, the following set of steps happens:
- The first target *hello* is selected because the first target is the default target
- This has a prerequisite of *helloworld.cpp*
- Make decides if it should run the *hello* target. It will only run if *hello* doesn't exist, or *helloworld.cpp* is newer than the previous build.

This last step is critical and is the essence of our Make. What it's attempting to do is decide if the prerequisites of blah have changed since blah was last compiled.

That is, if *helloworld.cpp* is modified, running make should recompile the file. And conversely, if *helloworld.cpp* has not changed, then it should not be recompiled.

To make this happen, it uses the filesystem timestamps as a proxy to determine if something has changed. This is a reasonable heuristic because file timestamps typically will only change if the files are modified. But it's important to realize that this isn't always the case.

You could, for example, modify a file, and then change the modified timestamp of that file to something old. If you did, Make would incorrectly guess that the file hadn't changed and thus could be ignored.

4. Make clean

The *clean* is often used as a target that removes the output of other targets, but it is not a special word in Make. You can run make and make clean on this to create and delete some_file.

Note that clean is doing two new things here:

- It's a target that is not first (the default) and not a prerequisite. That means it should never run unless you explicitly call make clean
- It's not intended to be a filename and should be marked as *PHONY*. If you happen to have a file named clean, this target won't run, which is not what we want.

A phony target is one that is not really the name of a file. It is just a name for some commands to be executed when you make an explicit request. There are two reasons to use a phony target: to avoid a conflict with a file of the same name and to improve performance.

```
hello: helloworld.cpp
	g++ -o hello helloworld.cpp

.PHONY: clean
clean:
	rm -f hello
```

5. Makefile Variables

We can define variables to reuse them across the Makefile. However, variables can only be strings. We can reference variables using either ${} or $():

```
input := helloworld.cpp
output := helloworld

${output}: $(input)
	g++ -o ${output} ${input}

clean:
	rm -f ${output}
```

Please notice that single or double quotes have no meaning to Make. They are simply characters that are assigned to the variable. Quotes are only useful to shell/bash command,

6. Targets

You can define an *all* target. Since this is the first rule listed, it will run by default if make is called without specifying a target.

```
all: one two three

one:
	touch one
two:
	touch two
three:
	touch three

clean:
	rm -f one two three
```

When there are multiple targets for a rule, the commands will be run for each target. We will use *$@* which is an automatic variable that contains the target name.

```
all: f1.o f2.o

f1.o f2.o:
	echo $@
# Equivalent to:
# f1.o:
#	echo f1.o
# f2.o:
#	echo f2.o
```

7. Wildcards

The * wildcard pattern searches your filesystem for matching filenames. * may be used in the target, prerequisites, or in wildcard function. Please notice that:

- The * pattern may not be directly used in variable definitions
- When * matches no files, it is left as it is (unless run in the wildcard function)

```
thing_wrong := *.o # Don't do this! '*' will not get expanded
thing_right := $(wildcard *.o)

all: one two three four

# Fails, because $(thing_wrong) is the string "*.o"
one: $(thing_wrong)

# Stays as *.o if there are no files that match this pattern
two: *.o

# Works as you would expect! In this case, it does nothing.
three: $(thing_right)

# Same as rule three
four: $(wildcard *.o)
```

- The **targets** are file names, separated by spaces. Typically, there is only one per rule.
- The **commands** are a series of steps typically used to make the target(s). These need to start with a tab character, not spaces.
- The **prerequisites** are also filenames, separated by spaces. These files need to exist before the commands for the target are run. These are also called dependencies

The % pattern is really useful, but is somewhat confusing because of the variety of situations it can be used in:

- When used in "matching" mode, it matches one or more characters in a string. This match is called the stem.
- When used in "replacing" mode, it takes the stem that was matched and replaces that in a string.
- % is most often used in rule definitions and in some specific functions.

8. Automatic variables

They are variables that have values computed afresh for each rule that is executed, based on the target and prerequisites of the rule:

```
hey: one two
	# Outputs "hey", since this is the target name
	echo $@

	# Outputs all prerequisites newer than the target
	echo $?

	# Outputs all prerequisites
	echo $^

	touch hey

one:
	touch one

two:
	touch two

clean:
	rm -f hey one two
```

- The **targets** are file names, separated by spaces. Typically, there is only one per rule.
- The **commands** are a series of steps typically used to make the target(s). These need to start with a tab character, not spaces.
- The **prerequisites** are also filenames, separated by spaces. These files need to exist before the commands for the target are run. These are also called dependencies

Since building large C/C++ programs usually involves multiple steps, a tool like Make is needed to ensure all source files are compiled and linked. Make also lets the developer control how ancillary files like documentation, man pages, systemd profiles, init scripts, and configuration templates are packaged and installed.

9. Implicit Rules

Let's see how we can now build a C++ program without ever explicitly telling Make how to do the compilation, by using the implicit rules:

- CC: Program for compiling C programs; default cc
- CXX: Program for compiling C++ programs; default g++

- CFLAGS: Extra flags to give to the C compiler
- CXXFLAGS: Extra flags to give to the C++ compiler
- CPPFLAGS: Extra flags to give to the C preprocessor
- LDFLAGS: Extra flags to give to compilers when they are supposed to invoke the linker

```
CCX = g++ # Flag for implicit rules
CXXFLAGS = -g # Flag for implicit rules. Turn on debug info

# Implicit rule #1: helloworld is built via the C++ linker implicit rule
# Implicit rule #2: helloworld is built via the C++ compilation implicit rule
because helloworld.cpp exists

helloworld: helloworld.cpp

clean:
	rm -f helloworld
```

10. Pattern rules

As described in the previous example, we have used some pattern rules to write our Makefile more efficiently. Let’s take a deeper look at these rules:

```
# Define a pattern rule that compiles every .c file into a .o file
%.o : %.c
		$(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
```

```
# Define a pattern rule that has no pattern in the prerequisites.
# This just creates empty .c files when needed.
%.c:
   touch $@
```

```
all: blah

# Double-Colon Rules are rarely used, but allow multiple rules to be defined for
the same target
blah::
	echo "hello"

blah::
	echo "hello again"
```

We can also combine implicit rules, wildcards, and automatic variables to create static pattern rules:

```
objects = foo.o bar.o all.o

all: $(objects)
```

```
# These files compile via implicit rules
# Syntax - targets ...: target-pattern: prereq-patterns ...
# In the case of the first target, foo.o, the target-pattern matches foo.o and
sets the "stem" to be "foo".
# It then replaces the '%' in prereq-patterns with that stem

$(objects): %.o: %.c

all.c:
	echo "int main() { return 0; }" > all.c

%.c:
	touch $@

clean:
	rm -f *.c *.o all
```

11. Check if a variable is empty

```
nullstring =
foo = $(nullstring) # end of line; there is a space here

all:
ifeq ($(strip $(foo)),)
	echo "foo is empty after being stripped"
endif
ifeq ($(nullstring),)
	echo "nullstring doesn't even have spaces"
endif
```

12. Check if variable is defined

```
bar =
foo = $(bar)

all:
ifdef foo
	echo "foo is defined"
endif
ifndef bar
	echo "but bar is not"
endif
```

13. Makefile Cookbook

```
TARGET_EXEC := final_program

BUILD_DIR := ./build
SRC_DIRS := ./src

# Find all the C and C++ files we want to compile
# Note the single quotes around the * expressions. Make will incorrectly expand
these otherwise.
SRCS := $(shell find $(SRC_DIRS) -name '*.cpp' -or -name '*.c' -or -name '*.s')

# String substitution for every C/C++ file.
# As an example, hello.cpp turns into ./build/hello.cpp.o
OBJS := $(SRCS:%=$(BUILD_DIR)/%.o)

# String substitution (suffix version without %).
# As an example, ./build/hello.cpp.o turns into ./build/hello.cpp.d
DEPS := $(OBJS:.o=.d)

# Every folder in ./src will need to be passed to GCC so that it can find header
files
INC_DIRS := $(shell find $(SRC_DIRS) -type d)
# Add a prefix to INC_DIRS. So moduleA would become -ImoduleA. GCC understands
this -I flag
INC_FLAGS := $(addprefix -I,$(INC_DIRS))

# The -MMD and -MP flags together generate Makefiles for us!
# These files will have .d instead of .o as the output.
CPPFLAGS := $(INC_FLAGS) -MMD -MP

# The final build step.
$(BUILD_DIR)/$(TARGET_EXEC): $(OBJS)
	$(CXX) $(OBJS) -o $@ $(LDFLAGS)

# Build step for C source
$(BUILD_DIR)/%.c.o: %.c
	mkdir -p $(dir $@)
	$(CC) $(CPPFLAGS) $(CFLAGS) -c $< -o $@

# Build step for C++ source
$(BUILD_DIR)/%.cpp.o: %.cpp
	mkdir -p $(dir $@)
	$(CXX) $(CPPFLAGS) $(CXXFLAGS) -c $< -o $@


.PHONY: clean
clean:
	rm -r $(BUILD_DIR)

# Include the .d makefiles. The - at the front suppresses the errors of missing
# Makefiles. Initially, all the .d files will be missing, and we don't want those
# errors to show up.
-include $(DEPS)
```

VI. GNU DEBUGGER

1. Introduction to GDB

The normal process for developing computer programs goes something like this: write some code, compile the code, and run the program. If the program does not work as expected, then you go back to the code to look for errors (bugs) and repeat the cycle again.

Depending on the complexity of the program and the nature of the bugs, there are times when you can do with some additional help in tracking down the errors. This is what a "debugger" does. It allows you to examine a computer program while it is running. You can see the values of the different variables, you can examine the contents of memory and you can halt the program at a specified point and step through the code one line at a time.

The primary debugger on Linux is the GNU debugger (gdb). It might already be installed on your system (or a slimmed-down version called gdb-minimal), but to be sure type the following *gdb* command in a terminal:

```
~$ gdb --version
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-120.el7
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
```

2. Debugging C++ program with GDB

Let's modify our following basic Hello World C++ program:

```
#include<iostream>

using namespace std;

int main() {
      int i;
       for (i=0; i < 10; i++) {
           cout<<i<<endl;
   }
       return 0;
}
```

Then compile with the *-g* option:

```
~$ g++ -g -o helloworld helloworld.cpp
```

To start debugging the program type:

```
~$ gdb helloworld
```

At this point if you just start running the program (using the "*run*" command), then the program will execute and finish before you get a chance to do anything. To stop that, you need to create a "**breakpoint**" that will halt the program at a specified point. The easiest way to do this is to tell the debugger to stop in the function "*main()*":

```
break main
```

Now start the program:

```
run
```

The debugger will stop at the first executable line of the main function, e.g. the "for" loop. To step to the next line, type "*next*" or "*n*" for short. Keep using "*next*" to repeat the loop a couple of times.

To inspect the value of a variable use the "*print*" command. In our example program, we can examine the contents of the variable "*i*":

```
print i
```

Repeat around the loop and few more times and see how "*i*" changes:

```
next

next

next

next

print i
```

The loop will continue while "*i*" is less than 10. You can change the value of a variable using "*set var.*" Type the following in gdb to set "*i*" to 10.

```
set var i = 10

print i
```

```
next
```

You may need to do another "*next*" (depending on where the program was halted when you set "*i*" to 10), but when the "*for*" loop line is next executed, the loop will exit, because "*i*" is no longer less than 10.

The "*next*" command doesn't drill down into functions but rather the function is executed, and the debugger stops again at the next line after the function. If you want to step into a function, use the "*step*" command, or "*s*" for short.

Another way to debug your program is to set a watch on a variable. What this does is halt the program whenever the variable changes. Restart the program again by typing "*run*." Since the program is already running, the debugger will ask if you want to start it again from the beginning.

The program will stop in the main (as we didn't remove the breakpoint). Now set a watch on "*i*":

```
watch i

continue
```

To stop debugging, just use the "*quit*" command.

VII. IN-CLASS EXERCISE

1. Exercise Requirements:

In the previous class, we built a Shell Script version of a calculator. Following this idea, we will re-implement a C++ version.

Requirements:

- Students must create a Makefile to build the program with at least these 2 targets: *all* and *clean*
- The executable name is "*calc*"
- The main program is implemented in the *calc.cpp* source file, while the calculation logic is held on the other source files.
- Input and output requirements are the same as the Shell Script version

```
~$ ./calc
>> 2 + 5
7
```

```
~$ ./calc
>> ANS + 3
10
```

2. Submission

- Submission guidelines will be announced later on the Elearning website.

VIII. REFERENCES AND EXTRA MATERIALS

- The C++ compilation process.
  https://riptutorial.com/cplusplus/example/26378/the-cplusplus-compilation-process
- How to Debug a C or C++ Program on Linux Using gdb.
  https://www.maketecheasier.com/debug-program-using-gdb-linux/
- GCC and Make: Compiling, Linking and Building C/C++ Applications.
  https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html