ENSF 337: Programming Fundamentals for Software and Computer

Department of Electrical & Computer Engineering
University of Calgary

Lab 7, Fall 2021 – Week of November 1st

M. Moussavi, PhD, P.Eng

Acknowledgment: This lab contains material written by Dr. Maan Khedr, for ENSF 337- Fall 2021

In This Lab You Can Work with a Partner (Groups of three or more are NOT allowed):

Working with a partner usually should give you the opportunity to discuss some of details of the topics and learn from each other. Also, it will give you the opportunity to practice one of the popular methods of program development called, **pair-programming**. In this method, which is normally associated with "Agile Software Development" technique, two programmers work together normally on the same workstation (you may consider a zoom session for this purpose). While one partner, the driver, writes the code, the other partner, acts as observer, looks over his/her shoulder making sure the syntax and solution logic is correct. Partners should switch roles frequently in a way that both of them have equivalent opportunity to practice both roles.
If you decided to work with a partner, please submit only one lab report with both names. Submitting two lab reports with the same content will be considered as copies and plagiarism.

Objective:

Here is the summary of the some of the topics that are covered in this lab:

- C++ objects on the computer memory
- Designing a C++ class
- Understanding the details of copying objects in C++

Marking Scheme:	
Exercise A:	4 marks
Exercise B:	not marked
Exercise C:	22 marks
Exercise D:	12 marks
Exercise E:	4 marks
Total:	**42 marks**

Important Notes:

- Exercise B is not marked, but it is an important exercise that helps you to understand how some of the elements of object-oriented programs, including: constructor, default constructor, and destructor work, and when and how they will be called.

- If you are compiling and running your program from command line, make sure to use C++ compilation command `g++` instead of `gcc`.

Due Date: Thursday November 18, before 2:00 PM

Exercise A:

The objective of this exercise is to help you in understanding how C++ objects are shown on a memory diagram, and how a member function of a class uses the `'this'` pointer to access an object associated with a particular function call. For further details please refer to your lecture notes and slides.

What to Do:

Download files `cplx.cpp`, `cplx.h`, and `lab7ExA.cpp` from the D2L and draw AR diagrams for: **point one** and **point two.**

For this exercise you just need to draw the diagrams. However, if you want to compile and run it from command line, you should have all of the given files in the same directory and from that directory you should use the following command to compile and create an executable:

```
g++ -Wall cplx.cpp lab7ExA.cpp
```

Please notice that you shouldn't have the header file name(s) in this command.

Exercise B: Construction and Destruction of Objects

What to Do:

Compile the files `lab7ExB.cpp`, and `lab7String.cpp`, run the program and fill out the following table to indicate which object or which pointer (a, p, d, b, c, or g) is associated with the call to a constructor or destructor. As an example, the answer for the first row is given – which indicates that the first call to the constructor (abbreviated as `ctor`) is associated with the `Lab7String` object, `a`.
Notes:

- Also, many programmer use `dtor` as an abbreviation for destructor.
- When compiling this program, compiler may give you warning for unused variables. Although warnings are always important to be considered and to be fixed, but in this exercise, we ignored those warnings to keep the program short and simple.

Program output in order that they appear on the screen	Call is associated with object(s):
`ctor called...`	`a`
`default ctor called...`	
`default ctor called...`	
`ctor called...`	
`ctor called...`	
`ctor called...`	
`The first four calls to dtor`	
`The last two calls to dtor`	

Exercise C: Designing a C++ Class:

Read This First – What is a Helper Function?

One of the important elements of good software design is the concept of code-reuse. The idea is that if any part of the code is repeatedly being used, we should wrap it into a function, and then reuse it by calling the function as many times as needed. In the past labs in this course and the previous programming course, we have seen how we can develop global function to reuse them as needed. A similar approach can be applied within a C++ class by implementing **helper-functions**. These are the functions that are declared as private member functions and **<u>are only available to the member functions of the class</u>** -- Not available to the global functions such as main or member functions of the other classes.

If you pay close attention to the given instruction in the following "What to Do" section, you will find that there are some class member functions that need to implement a similar algorithm. They all need to change the value of data members of the class in a more or less similar fashion. Then, it can be useful if you write one or more **private helper-function**, that can be called by any of the other member functions of the class, as needed.

Read This Second – Complete Design and Implementation Class - Clock

In this exercise you are going to design and implement a C++ class called, `Clock` that represents a 24-hour clock. This class should have three private integer data members called: `hour`, `minute`, and `second`. The minimum value of these data members is zero and their maximum values should be based on the following rules:

- The values of `minute`, and `second` in the objects of class `Clock` **cannot** be less than 0 or more than `59`.
- The value of hour in the objects of class `Clock` cannot be less than 0 or more than 23.
- As an example any of the following values of hour, minute, and second is acceptable for an object of class `Clock` (format is `hours:minutes:seconds`) : `00:00:59, 00:59:59, 23:59:59, 00:00:00`. And, all of the following examples are **unacceptable**:
  - `24:00:00` (hour cannot exceed 23)
  - `00:90:00` (minute of second cannot exceed 59)
  - `23:-1:05` (none of the data members of class `Clock` can be negative)

Class `Clock` should have three constructors:
A default constructor, that sets the values of the data-members `hour`, `minute`, and `second` to zeros.
A second constructor, that receives an integer argument in seconds, and initializes the `Clock` data members with the number of `hour`, `minute`, and `second` in this argument. For example, if the argument value is `4205`, the values of data members `hour`, `minute` and `second` should be: `1`, `10`, and `5` respectively. If the given argument value is negative the constructor should simply initialize the data members all to zeros.
The third constructor receives three integer arguments and initializes the data members `hour`, `minute`, and `second` with the values of these arguments. If any of the following conditions are true this constructor should simply initialize the data members of the `Clock` object all to zeros:

- If the given values for second or minute are greater than 59 or less than zero.
- If the given value for hour is greater than 23 or less than zero.

Class `Clock` should also provide a group of access member functions (getters, and setters) that allow the users of the class to retrieve values of each data member, or to modify the entire value of time. As a convention, lets have the name of the getter functions started with the word `get`, and the setter functions started with word `set`, both followed by an underscore, and then followed by the name of data member. For example, the getter for the data member `hour` should be called `get_hour`, and he setter for the data member `hour` should be called `set_hour`. Remember that getter functions must be declared as a `const` member function to make them read-only functions.

All setter functions must check the argument of the function not to exceed the minimum and maximum limits of the data member. If the value of the argument is below or above the limit the functions are supposed to do nothing.

In addition to the above-mentioned constructors and access functions, class `Clock` should also have a group of functions for additional functionalities (lets call them implementer functions) as follows:

1. A member function called `increment` that increments the value of the clock's time by one.

Example: If the current value of time is `23:59:59`, this function will change it to: `00:00:00` (which is midnight sharp). Or, if the value of the time is `00:00:00` a call to this function increments it by one and makes it: `00:00:01` (one second past midnight – the next day) .

2. A member function called `decrement` that decrements the value of the clock's time by one.
   Example: If the current value of time is `00:00:00`, this function will change it to: `23:59:59`. Or, if the value of current time is `00:00:01`, this function will change it to: `00:00:00`.
3. A member function called `add_seconds` that REQUIRES to receive a positive integer argument in seconds and adds the value of given seconds to the value of the current time.
   For example, if the clock's time is `23:00:00`, and the given argument is `3601` seconds, the time should change to: `00:00:01`.
4. Two helper functions**.** These functions should be called to help the implementation of the other member functions, as needed. Most of the above-mentioned constructors and implementer function should be able to use these functions:
   - A **private** function called `hms_to_sec`: that returns the total value of data members in a `Clock` object, in seconds. For example if the time value of a `Clock` object is `01:10:10`, returns `4210` seconds.
   - A **private** function called `sec_to_hms`, which works in an opposite way. It receives an argument (say, `n`), in seconds, and sets the values for the `Clock` data members, `second`, `minute`, and `hour`, based on this argument. For example, if `n` is `4210` seconds, the data members values should be: `1, 10` and `10`, respectively for `hour`, `minute`, and `second`.

What to Do:

If you haven't already read the "**Read This First**" and "**Read This Second**", in the above sections, read them first. The recommended concept of helper function can help you to reduce the size of repeated code in your program.

Then, download file `lab7ExC.cpp` from D2L. This file contains the code to be used for testing your class `Clock`.

Now, take the following steps to write the definition and implementation of your class `Clock` as instructed in the above "Read This Second" section.

1. Create a header file called `lab7Clock.h` and write the definition of your class `Clock` in this file. Make sure to use the appropriate preprocessor directives (`#ifndef, #define`, and `#endif`), to prevent the compiler from duplication of the content of this header file during the compilation process.
2. Create another file called `lab7Clock.cpp` and write the implementation of the member functions of class `Clock` in this file (remember to include `"lab7Clock.h"`).
3. Compile files `lab7ExC.cpp` (that contain the given `main` functions) and `lab7Clock.cpp` to create your executable file.
4. If your program shows any compilation or runtime errors fix them until your program produces the expected output as mentioned in the given main function.
5. Now you are done!

What to Submit:

Submit your source code, `lab7Clock.h`, and `lab7Clock.cpp`, and the program's output as part of your lab report in pdf on the D2L Dropbox.

Exercise D: A Simple Class Vector and Copying Object

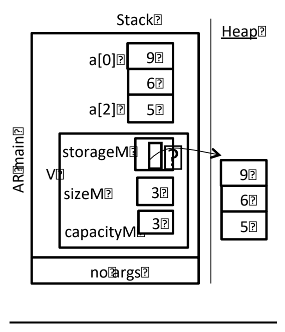
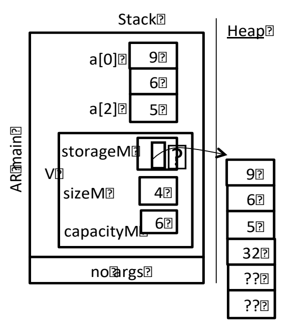
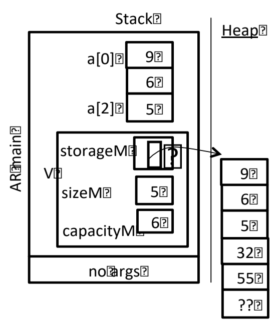
The objective of this exercise is to practice more code-level design concepts such as dynamic allocation and de-allocation of memory for class data members and to understand the concepts of copying objects.

What to Do:

Download files `simpleVector.h`, `simpleVector.cpp`, and `lab7ExD.cpp` from D2L. Open the given files and read their contents carefully to understand the details of implementation of some of the given members. If you compile and run this program, you will notice that some of the expected test results are incorrect. This is because the `copy constructor`, the `assignment operator` (which is also known as `copy assignment operator`), and one of the member functions called `push_back` is missing.

In this exercise your job is to complete the definition of these functions. In the given main function, part of the code that tests the copy assignment operator or copy constructor is commented out by being confined between `#if 0` … `#endif`. When ready to test your copy assignment operator and copy constructor, please change the `#if 0` to `#if 1`. You are also recommended to test one function at time; once one function works with no errors move to the next one. For this purpose you can move the location of the conditional compilation directives (`#if 0`) to an appropriate lower parts of the main function, as you progress. To better understand how this class is supposed to work, first read carefully the content of the given files and the comments on memory policy. Also, see the following example for a main function that creates an object of class `SimpleVector`, and the AR diagrams at points one, two, and three. The diagrams show how the object `v` is initially created, and then what will happen to the object if the function `push_back` works as it is expected.

int main(void)	Point 1	Point 2	Point 3

{
  int a [] = { 9, 6, 5 };
  SimpleVector v(a, 3);
  // point 1
  v.push_back(32);
  // point 2
  v.push_back(55);
  // point 3
  return 0;
}
			

What to Submit:

Submit your `simpleVector.cpp` and the output of the program, as part of your lab report in pdf format.

Exercise E - Code debugging and tracing:

written by: Dr. Maan Khedr

Read this first:

Code debugging and tracing is essential for resolving runtime and logical errors. When compiling a program, the compiler checks for code semantics and syntax and informs you of unrecognized instructions, wrong usage of variables and function calls, and many more. These errors that the compiler detect re referred to as compilation errors.

Unwanted behavior of a program during runtime is not something the compiler can account for, for example a program that tries to open a file in read mode but can not find the file will not cause a compilation error as the code itself is correct but there is something wrong in the logic or something went wrong during runtime. Hence it is essential to debug and trace your program output to know what went wrong

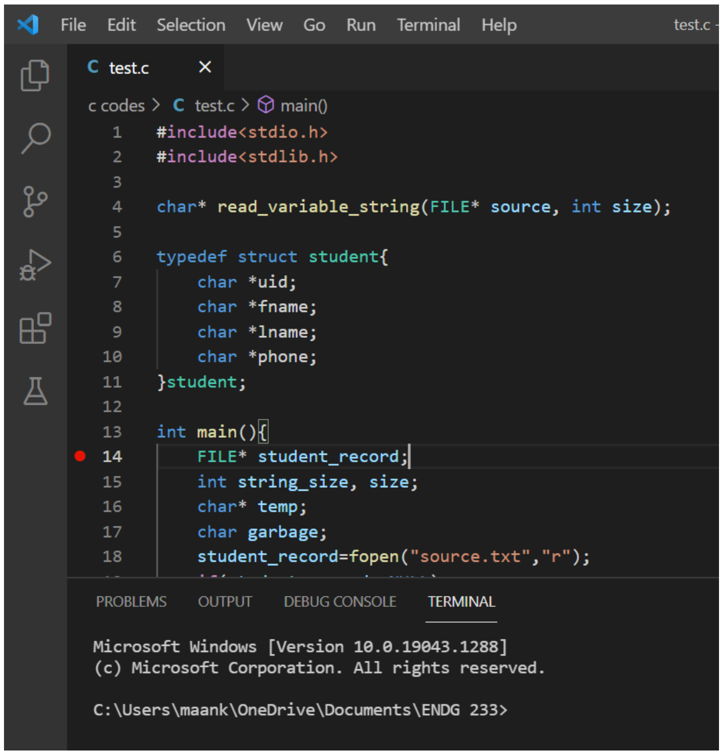
IDE-based debugging:

Integrated Development Environments provide a set a tools that are integrated together such as text-editors, preprocessors, compilers, and debuggers. This makes it easy for a developer to trace their code and keep an eye on variables and their contained values during runtime to trace the behavior of the program.

To do so, we utilize ***breakpoints***:

- Marked lines of code that the program will pause at during runtime, where the developer can then check the content of variables in memory at that instance. Furthermore, once the program pauses at a breakpoint they get full control on how to proceed from that line (single step, step into function calls, step over line, continue to completion or another breakpoint is reached).

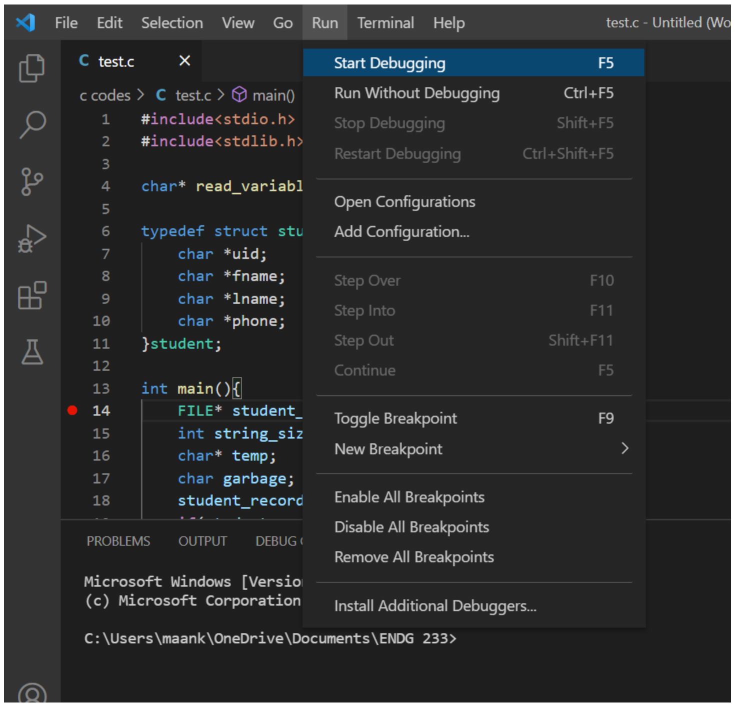
In vscode, when hovering the mouse cursor over the tab to the left of the line number, you will see a greyed red circle representing the possibility to add a break point at that line. By clicking you insert a breakpoint at that line and then it shows in bright red



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

char* read_variable_string(FILE* source, int size);

typedef struct student{
    char *uid;
    char *fname;
    char *lname;
    char *phone;
}student;

int main(){
    FILE* student_record;
    int string_size, size;
    char* temp;
    char garbage;
    student_record=fopen("source.txt","r");
```

```
Microsoft Windows [Version 10.0.19043.1288]
(c) Microsoft Corporation. All rights reserved.

C:\Users\maank\OneDrive\Documents\ENDG 233>
```

With breakpoints injected to the program, you can run the program without debugging (the IDE will skip the breakpoints and run in normal mode) or with debugging.
NOTE: the play button will run the program without debugging, to run program with debugging activated select ***"start debugging"*** from the ***"Run"*** menu.

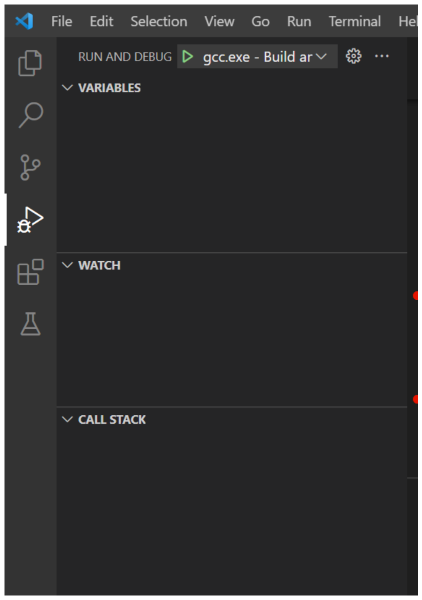


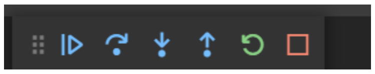
```
#include<stdio.h>
#include<stdlib.h>

char* read_variabl

typedef struct stu
    char *uid;
    char *fname;
    char *lname;
    char *phone;
}student;

int main(){
    FILE* student_
    int string_siz
    char* temp;
    char garbage;
    student_record
```

```
Microsoft Windows [Versio
(c) Microsoft Corporation

C:\Users\maank\OneDrive\Documents\ENDG 233>
```

When you run with debugging, the program will pause at the breakpoint and memory content will be displayed in panels usually docked on the left side of the IDE. Note that you can have multiple break points in the code.
The panels appearing to the left have sub containers for:

- Variables: shows you local and global variables values at the breakpoint
- Watch: can be set to monitor a specific variable or expression
- Call stack: shows the nested call stack … similar to activation records.





When the breakpoint is reached, the control tab will show at the top of the window
Buttons functionality from left to right:

1- Continue to next breakpoint if any or program completion
2- Step over, is a single step without moving into function evaluation
3- Step into, same as step over but will move to the function implementation if the next step is inside a function call
4- Step out, will take you back out of a function implementation
5- Restart debugging
6- Stop debugging

NOTE: for mac users, Xcode has the same functionality but the lay out might differ

Terminal_based debubgging:

When using command-line based compilers, debugging options breakdown to two alternatives:

Terminal output through prints:

Inject print statements to trace your code execution and variable values at points of your program.

- Usually print statements inside conditions are used to verify if a branch gets executed
- Printing variable values before and after function execution to verify expected return value from function.

Terminal- based debugger (GNU debugger GDB):

gdb is a standalone application that you can use to debug your program in terminal using breakpoints similar to how the IDE works. To be able to use gdb on a program, the program must be compiled with the debugging option enabled. This is done by injecting a "***-g***" into the compilation command such as

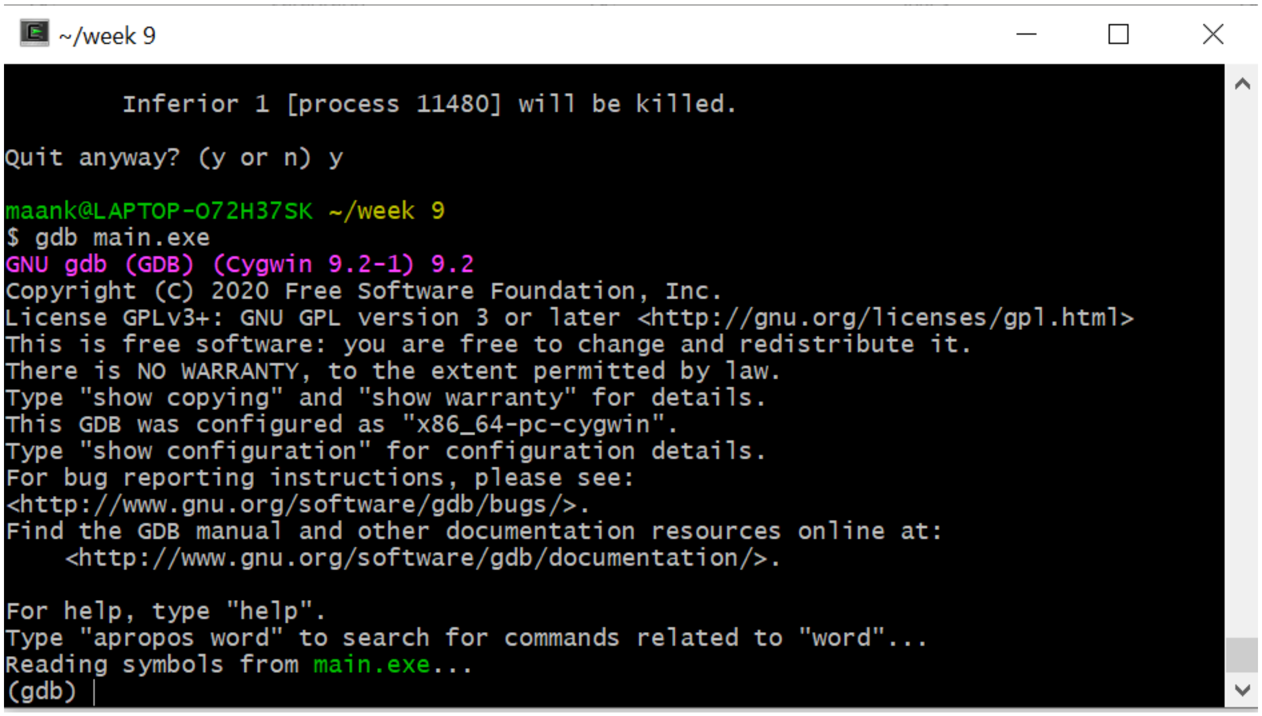
*gcc -Wall **-g** -o out prog.c*

once a program is compiled with debugging option, the generated executable can be passed over to gdb for debugging.

To run gdb with a given file, you need to type the command:

gdb executable_file_name.exe

This will start the gdb in the terminal with the runnable loaded, and then you can use a wide range of options to control the program execution and monitor your program output as it runs.



~/week 9

```
        Inferior 1 [process 11480] will be killed.

Quit anyway? (y or n) y

maank@LAPTOP-O72H37SK ~/week 9
$ gdb main.exe
GNU gdb (GDB) (Cygwin 9.2-1) 9.2
Copyright (C) 2020 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-pc-cygwin".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from main.exe...
(gdb) |
```

Some of the useful options you can use in the gdb:

- break: specifies a breakpoint, you can use this to insert a breakpoint at a specific line in your code file, or attach it to a function name so that whenever the function is called the program would pause:
- delete:
- run: starts the execution of the program and will pause whenever a breakpoint is reached
- continue, step, and next: all of these are controls for the debugger to move forward with program execution similar to what we discussed in the IDE section. "***step***" is a single step which will move you inside function implementation while "***next***" acts as the step over in the IDE section
- finish: continues function execution and then pauses
- print: prints the value of a specified variable when the program is at a break point
- watch: prints the value of a variable whenever it changes
- q: quits gdb

note that gdb keeps history of your commands in a session, using arrows you can access your previously typed commands, furthermore it can autocomplete when you press the tab button when typing a command. Finally, pressing enter without providing a command will re-execute your most recent command, for example if you enter the step command and then keep pressing enter it will keep executing the step command.

What to do:

The following is an example with screenshots highlighting some of these commands during execution as an example. Download the program files `main.cpp, point.cpp,` and `point.h` to follow with the exercise. The output is shown in the screenshots with the input commands outlined by a red box.

1- Compile the program with debugging option

```
g++ -Wall -g -o main main.cpp point.cpp
```

2- Open gdb with main executable file that we generated from the compilation

```
gdb main.exe
```

3- Insert breakpoints for each of the following:

   a. The line in main code at which the for loop starts

   ```
   break main.cpp:15
   ```

   b. A breakpoint for function display defined in the point class

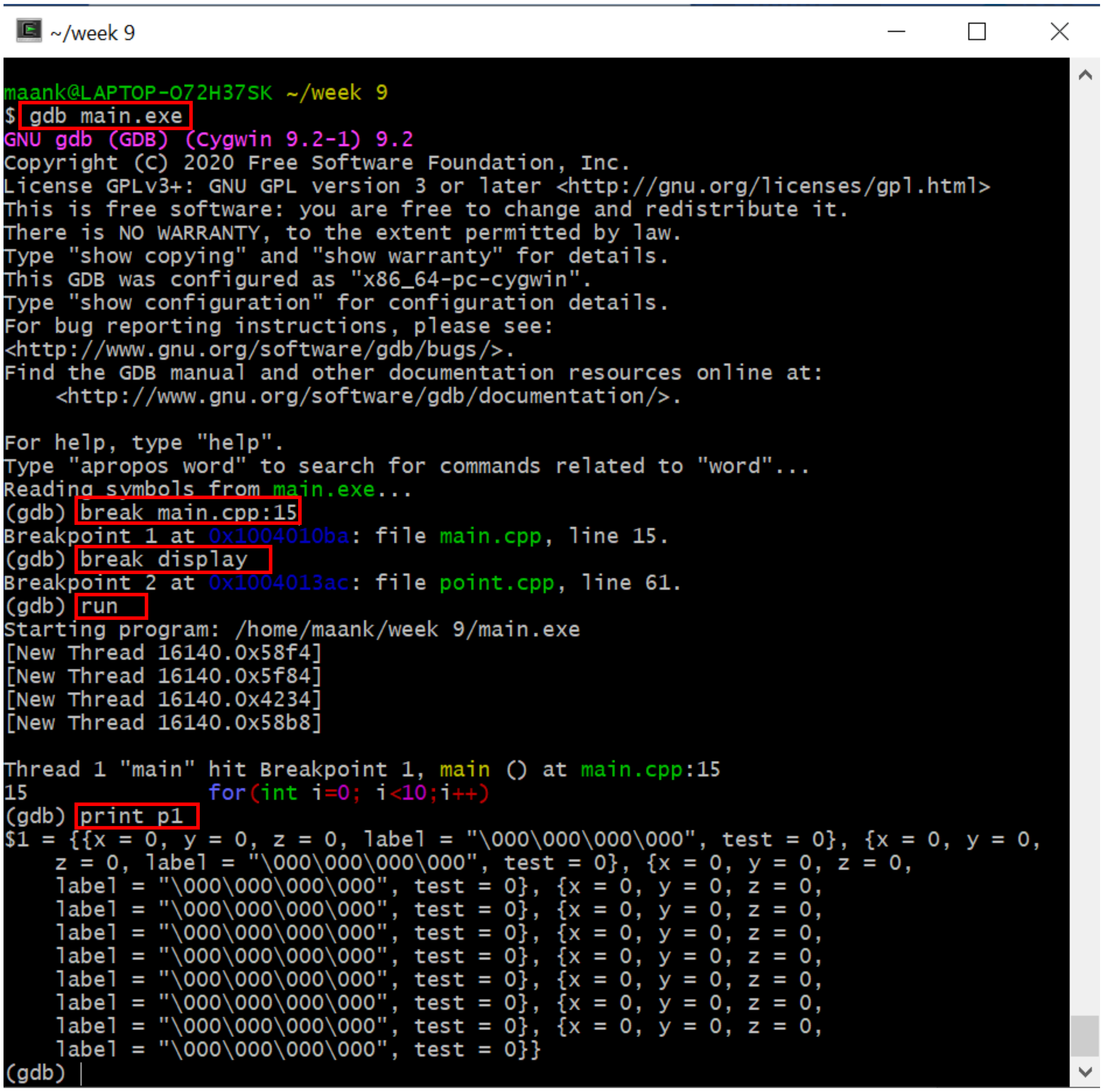
   ```
   break display
   ```

4- Let's run the program:

```
Run
```

5- And then we print the array of points

```
Print p1
```



```
maank@LAPTOP-072H37SK ~/week 9
$ gdb main.exe
GNU gdb (GDB) (Cygwin 9.2-1) 9.2
Copyright (C) 2020 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-pc-cygwin".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from main.exe...
(gdb) break main.cpp:15
Breakpoint 1 at 0x1004010ba: file main.cpp, line 15.
(gdb) break display
Breakpoint 2 at 0x1004013ac: file point.cpp, line 61.
(gdb) run
Starting program: /home/maank/week 9/main.exe
[New Thread 16140.0x58f4]
[New Thread 16140.0x5f84]
[New Thread 16140.0x4234]
[New Thread 16140.0x58b8]

Thread 1 "main" hit Breakpoint 1, main () at main.cpp:15
15                for(int i=0; i<10;i++)
(gdb) print p1
$1 = {{x = 0, y = 0, z = 0, label = "\000\000\000\000", test = 0}, {x = 0, y = 0,
    z = 0, label = "\000\000\000\000", test = 0}, {x = 0, y = 0, z = 0,
    label = "\000\000\000\000", test = 0}, {x = 0, y = 0, z = 0,
    label = "\000\000\000\000", test = 0}, {x = 0, y = 0, z = 0,
    label = "\000\000\000\000", test = 0}, {x = 0, y = 0, z = 0,
    label = "\000\000\000\000", test = 0}, {x = 0, y = 0, z = 0,
    label = "\000\000\000\000", test = 0}, {x = 0, y = 0, z = 0,
    label = "\000\000\000\000", test = 0}, {x = 0, y = 0, z = 0,
    label = "\000\000\000\000", test = 0}, {x = 0, y = 0, z = 0,
    label = "\000\000\000\000", test = 0}}
(gdb) |
```

6- Execute two steps, this will take us inside the display function implementation

```
step
Followed by empty line enter
```

7- Lets skip the function using finish

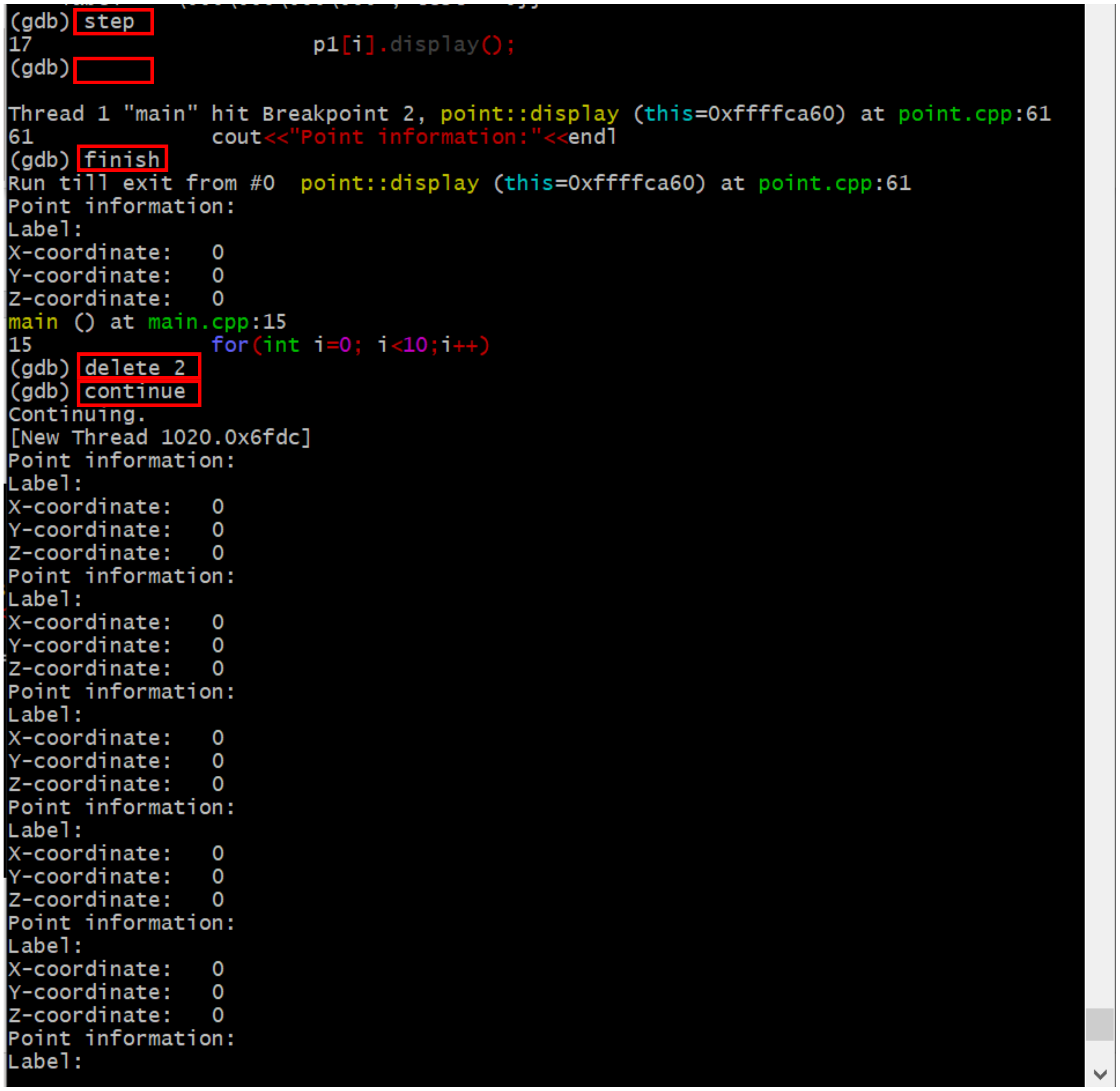
```
finish
```

8- Delete the breakpoint for the display function, this is the second breakpoint and we delete them by their number

```
delete 2
```

9- Continue program till end

```
continue
```



```
(gdb) step
17                      p1[i].display();
(gdb)

Thread 1 "main" hit Breakpoint 2, point::display (this=0xffffca60) at point.cpp:61
61                      cout<<"Point information:"<<endl
(gdb) finish
Run till exit from #0  point::display (this=0xffffca60) at point.cpp:61
Point information:
Label:
X-coordinate:    0
Y-coordinate:    0
Z-coordinate:    0
main () at main.cpp:15
15                      for(int i=0; i<10;i++)
(gdb) delete 2
(gdb) continue
Continuing.
[New Thread 1020.0x6fdc]
Point information:
Label:
X-coordinate:    0
Y-coordinate:    0
Z-coordinate:    0
Point information:
Label:
X-coordinate:    0
Y-coordinate:    0
Z-coordinate:    0
Point information:
Label:
X-coordinate:    0
Y-coordinate:    0
Z-coordinate:    0
Point information:
Label:
X-coordinate:    0
Y-coordinate:    0
Z-coordinate:    0
Point information:
Label:
X-coordinate:    0
Y-coordinate:    0
Z-coordinate:    0
Point information:
Label:
```

gdb is a capable debugger that is most of the time used as the debugger of choice for IDEs to provide you with the functionality you are seeking but is overlayed by a graphical interface to make it more user friendly.
Do your bit of becoming a better developer and read more about gdb, and some of its other features in the official documentation found here:
https://sourceware.org/gdb/current/onlinedocs/gdb/.
Second item on the documentation is a very useful sample session, check it out it will help you understand some othe concepts that you might need.
https://sourceware.org/gdb/current/onlinedocs/gdb/Sample-Session.html#Sample-Session
There are multiple cheat sheets with the most common commands and functionality you will need to use gdb proficiently available online.

What to hand in
Using gdb, replicate the steps provided previously in addition to adding a breakpoint to the default constructor of the class point, this will be placed between steps 2 and 3. After step 4 you will need to use continue multiple times before reaching step 5, provide answers for the following:

- How many times do you need to use the continue command?
- Why is the default constructor getting called these many times?

Provide screenshots showing your commands and output similar to the provided screenshots in the walkthrough.