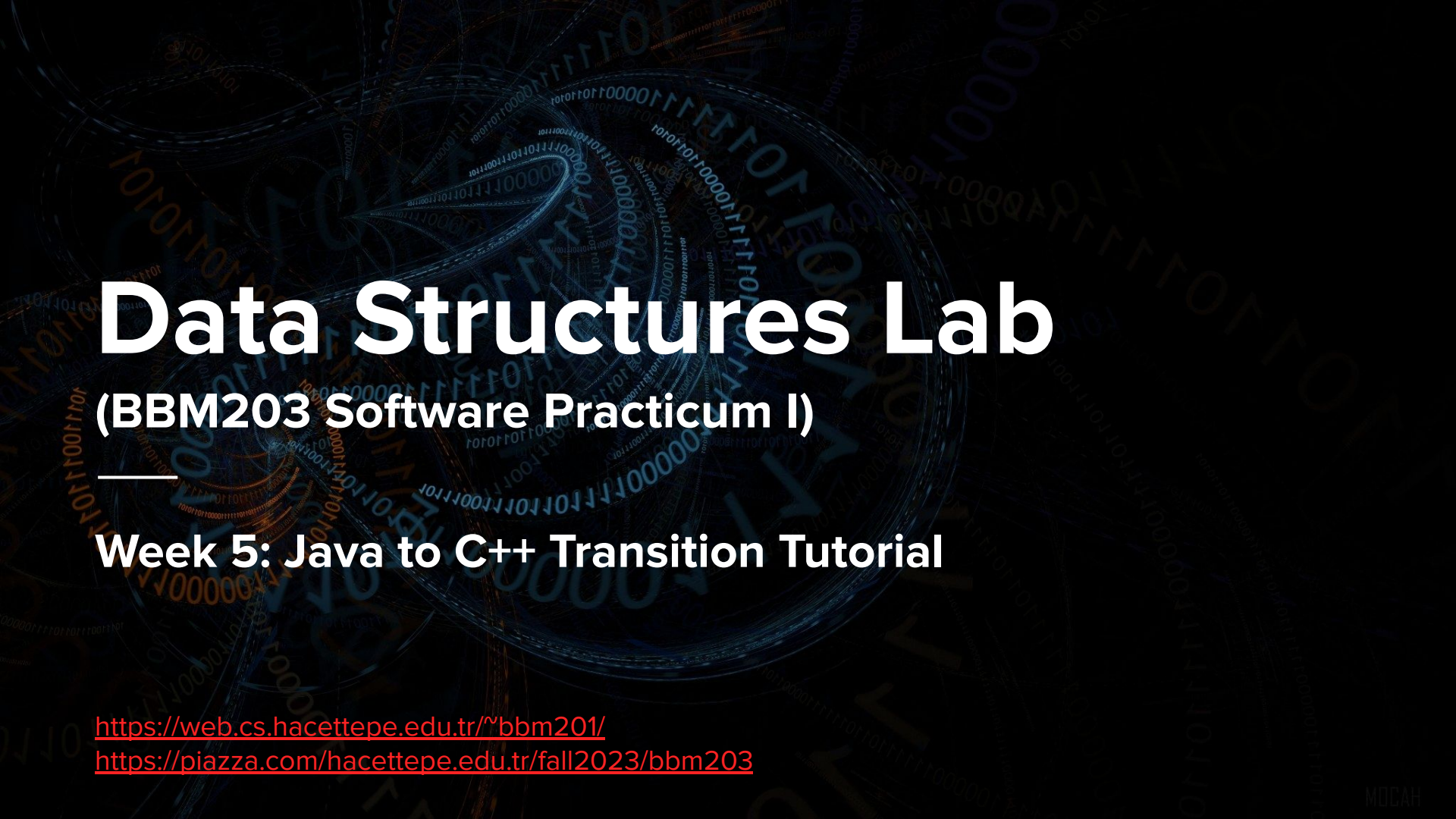


Data Structures Lab

(BBM203 Software Practicum I)

Week 5: Java to C++ Transition Tutorial

https://web.cs.hacettepe.edu.tr/~bbm201/
https://piazza.com/hacettepe.edu.tr/fall2023/bbm203

Topics

- ➔ Libraries
- ➔ File I/O
- ➔ Command Line Arguments
- ➔ Preprocessor
- ➔ More Debugging

File I/O

File I/O

- Handled using <fstream>
- **File Pointers**: bookmarks, keeping track of where we are in file. Does not contain actual data, merely a means of access and manipulation

- Files are on disk (in blocks)
- The OS provides mechanisms for access
- Each process has a FD table with int indices to open files. When a file is opened, it is added to the FD table, interfaced by the file pointer.
- The file pointers abstract this process and gives a handle

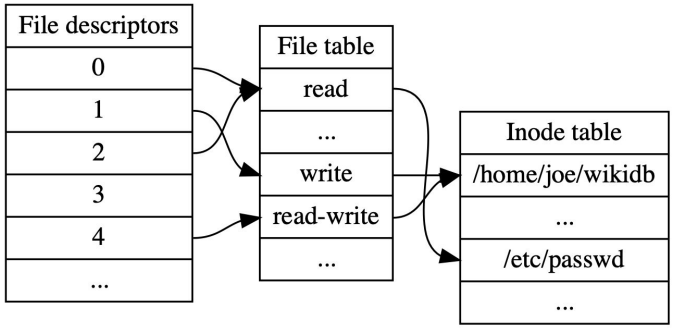
```
FILE* myfile = fopen(“example.txt”, “r”)
```

File I/O

- When a file is opened with fopen function, it returns a file pointer.

```
FILE* myfile = fopen(“example.txt”, “r”)
```

- This opens the example.txt file in READ mode, returning a FILE *.
- Keeps track of the current position in the file, enabling
  - Read, write, access operations
- Always close a file after opened
  - Free up resources (the slot in FD table)
  - Saving changes
  - Preventing data corruption/loss



File I/O

```
#include <iostream>
#include <fstream> // For file operations
#include <string>

void main() {

    // Reading from a file
    std::ifstream inFile("sample.txt"); // Opening file in read mode
    if (!inFile) {
        std::cerr << "Error opening file for reading!" << std::endl;
        return 2;
    }
    std::string line;
    while (std::getline(inFile, line)) { // Read file line by line
        std::cout << line << std::endl;
    }
    inFile.close(); // Close the file after reading
}
```

POSIX-based I/O

```cpp
#include <iostream>
#include <unistd.h>     // For read(), write(), and close()
#include <fcntl.h>      // For open()

int main() {
    char buffer[256];

    // 1. Reading from stdin (file descriptor 0): stdin
    std::cout << "Please enter some text: ";
    ssize_t bytesRead = read(0, buffer, sizeof(buffer) - 1);
    if (bytesRead == -1) {
        perror("Error reading from stdin");
        return 1;
    }
    buffer[bytesRead] = '\0';  // Null-terminate the string

    // 2. Writing to stdout (file descriptor 1)
    write(1, "You entered: ", 13);
    write(1, buffer, bytesRead);
}
```

Command Line Arguments

Command Line Arguments

- argc: # of arguments
- argv[ ]: array of string for each argument
- Can you rename them?

```
#include <iostream>

int main (int argc, char* argv[]) {
    std::cout << "Total arguments: " << argc << std::endl;
    for (int i = 0; i < argc; i++) {
        std::cout << "argv[" << i << "]: " << argv[i] <<
std::endl;
    }
    return 0;
}
```

Command Line

```cpp
#include <iostream>
#include <cstdlib> // for atoi

int main(int argc, char* argv[]) {
    if (argc != 4) {
        std::cerr << "Usage: calc <operation> <operand1>
<operand2>" << std::endl;
        return 1;
    }

    int op1 = atoi(argv[2]);
    int op2 = atoi(argv[3]);

    if (std::string(argv[1]) == "add") {
        std::cout << op1 + op2 << std::endl;
    } else if (std::string(argv[1]) == "sub") {
        std::cout << op1 - op2 << std::endl;
    } else {
        std::cerr << "Unsupported operation" << std::endl;
        return 2;
    }
    return 0;
}
```

Preprocessing

Preprocessing

Modifies the source code before compilation

- File inclusion “#include “MyClass.h”
- Macros: “#define …”
- Conditional Compilation: “#if”, “#ifdef”, “#ifndef” …
- #pragma (commands to the compiler, ensuring header is compiled once)

Including Files

- Basically copy-pasting the contents of the included file
  - Including .h vs .cpp files

- Two forms:
  - **# include <vector>** for standard libraries
  - **# include filename** for user defined files

Conditional Compilation

- Mainly used to avoid circular includes
- Use guards: “#ifndef”, “#define”, “#endif”

```cpp
#ifndef A_H
#define A_H

#include "B.h" // We include B.h because we want to use class B

class A {
public:
    A();
    ~A();

    void doSomethingWithB();

private:
    B* bInstance; // Pointer to an instance of class B
};

#endif
```

```cpp
#ifndef B_H
#define B_H

#include "A.h" // We include A.h because we want to use class A

class B {
public:
    B();
    ~B();

    void doSomethingWithA();

private:
    A* aInstance; // Pointer to an instance of class A
};

#endif
```

Introduction to Debugging Tools

Command-line Utilities

- Multiple tools are available to debug your code with or without a graphical user interface.
- We are going to learn about the basic usage of **gdb** for debugging **logic errors** and **runtime errors** with breakpoints and **valgrind** for checking **memory leaks** and **memory errors**.

The GNU Debugger (gdb)

The GNU Debugger is a command-line debugger utility that runs on many Unix-like systems and works for many programming languages.

- Start your program, specifying anything that might affect its behaviour.
- Make your program stop on specified conditions.
- Examine what has happened when your program has stopped.
- Change things in your program so you can experiment with correcting the effects of one bug and go on to learn about another.

The GNU Debugger (gdb)

The first thing to do to use **gdb** or **valgrind** for debugging your program is to provide the `-g` flag to the **gcc** or **g++** compiler.

```
g++ -g -o my_program my_program.cpp
```

If you fail to provide the `-g` flag to the compiler, you will not be able to see the associated line numbers with the source files.

The GNU Debugger

After proper compilation, the following command could be used to invoke gdb:

```
gdb --args ./my_program 3 5
```

If you don’t have command line arguments for your program, then you can do:

```
gdb ./my_program
```

```
$ gdb --args ./my_program 3 5
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 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-linux-gnu".
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 ./my_program...
(gdb)
```

The GNU Debugger

- gdb accepts a set of useful commands along with the functionality of stopping on its own in the face of trouble.
- gdb can help debug **segmentation faults**, **runtime exceptions**, and l**ogic errors.**
- To access the whole list of commands, you can refer to this cheatsheet.

Gdb Command	Description
set listsize n	Set the number of lines listed by the list command to n [set listsize]
b function	Set a breakpoint at the beginning of function [break]
b line number	Set a breakpoint at line number of the current file. [break]
info b	List all breakpoints [info]
delete n	Delete breakpoint number n [delete]
r args	Start the program being debugged, possibly with command line arguments args. [run]
s count	Single step the next count statements (default is 1). Step into functions. [step]
n count	Single step the next count statements (default is 1). Step over functions. [next]
finish	Execute the rest of the current function. Step out of the current function. [finish]
c	Continue execution up to the next breakpoint or until termination if no breakpoints are encountered. [continue]
p expression	print the value of expression [print]
l optional_line	List next listsize lines. If optional_line is given, list the lines centered around optional_line. [list]
where	Display the current line and function and the stack of calls that got you there. [where]
h optional_topic	help or help on optional_topic [help]
q	quit gdb [quit]

Extra: gdb watch and display, gdb assignment

GDB Example 1 - Segfault

GDB Example 2 - Exception

GDB Example 3 - Logic Error

Valgrind

- **Valgrind** is a GPL'd system for **debugging** and **profiling** Linux programs.
- Valgrind has a suite of useful tools such as: **Memcheck, Cachegrind, Callgrind, Massif, …**
- We are going to be focusing on the usage of **memcheck** in this lab.
- If you want to more about that tool suite, you can click here.

```
==147== HEAP SUMMARY:
==147==     in use at exit: 2,000 bytes in 1 blocks
==147==   total heap usage: 2 allocs, 1 frees, 74,704 bytes allocated
==147==
==147== LEAK SUMMARY:
==147==    definitely lost: 2,000 bytes in 1 blocks
==147==    indirectly lost: 0 bytes in 0 blocks
==147==      possibly lost: 0 bytes in 0 blocks
==147==    still reachable: 0 bytes in 0 blocks
==147==         suppressed: 0 bytes in 0 blocks
==147== Rerun with --leak-check=full to see details of leaked memory
```

Memcheck: a memory error detector

- **memcheck** is a memory debugging tool that reports memory errors and leaks and offers memory profiling.
- It uncovers subtle memory issues that may not result in immediate errors.
- It provides comprehensive reports for in-depth memory analysis.
- It tracks memory allocation and deallocation, preventing resource leaks.

Common Options

The following command could be used to invoke the memcheck tool:

```
valgrind ./my_program 3 5
```

If you want to trace memory leaks in detail, you can use:

```
valgrind --leak-check=full ./my_program 3 5
```

To see information on the sources of uninitialised data in your program, you can use:

```
valgrind --track-origins=yes ./my_program 3 5
```

```
$ ./valgrind my_program 3 5
==118== Memcheck, a memory error detector
==118== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==118== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright
info
==118== Command: ./my_program 3 5
==118==
Great success.
Great success.
==118==
==118== HEAP SUMMARY:
==118==     in use at exit: 0 bytes in 0 blocks
==118==   total heap usage: 2 allocs, 2 frees, 73,728 bytes allocated
==118==
==118== All heap blocks were freed -- no leaks are possible
==118==
==118== For lists of detected and suppressed errors, rerun with: -s
==118== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

You are encouraged to check the manual for the **memcheck** tool here.

Valgrind Example 1 - Invalid Delete

Valgrind Example 2 - Invalid Read

Valgrind Example 3 - Invalid Write

Valgrind Example 4 - Memory Leak

Valgrind Example 5 - Uninitialized Values

More Useful Resources For Practice:

- https://www.w3resource.com/cpp-exercises/basic/index.php
- https://www.w3schools.com/cpp/cpp_exercises.asp
- https://www.hackerrank.com/domains/cpp
- https://algoleague.com/

Questions?