Lab 1

Linux / g++ / CMake

ISDN3000E · Programming for Integrative Systems

Today’s Agenda

Learning Philosophy

> "I hear and I forget. I see and I remember. **I do and I understand.**"

— Confucius (paraphrased)

💻 Each module has hands-on exercises.
📁 Lab materials: `isdn3000e-lab1-tasks/`

Today’s Agenda

🐧 Module 1: The Shell (60 min)

Learning Philosophy

> "I hear and I forget. I see and I remember. **I do and I understand.**"

— Confucius (paraphrased)

💻 Each module has hands-on exercises.
📁 Lab materials: `isdn3000e-lab1-tasks/`

Today’s Agenda

🐧 Module 1: The Shell (60 min)

Your command-line superpower

Learning Philosophy

> "I hear and I forget. I see and I remember. **I do and I understand.**"

— Confucius (paraphrased)

💻 Each module has hands-on exercises.
📁 Lab materials: `isdn3000e-lab1-tasks/`

Today's Agenda

🐧 Module 1: The Shell (60 min)

Your command-line superpower

⚙️ Module 2: g++ Compiler (45 min)

Learning Philosophy

> "I hear and I forget. I see and I remember. **I do and I understand.**"

— Confucius (paraphrased)

💻 Each module has hands-on exercises.
📁 Lab materials: `isdn3000e-lab1-tasks/`

Today’s Agenda

🐧 Module 1: The Shell (60 min)

Your command-line superpower

⚙️ Module 2: g++ Compiler (45 min)

From source code to executable

Learning Philosophy

> "I hear and I forget. I see and I remember. **I do and I understand.**"

— Confucius (paraphrased)

💻 Each module has hands-on exercises.
📁 Lab materials: `isdn3000e-lab1-tasks/`

Today's Agenda

🐧 Module 1: The Shell (60 min)

Your command-line superpower

⚙️ Module 2: g++ Compiler (45 min)

From source code to executable

📦 Module 3: CMake (45 min)

Learning Philosophy

> "I hear and I forget. I see and I remember. **I do and I understand.**"

— Confucius (paraphrased)

💻 Each module has hands-on exercises.
📁 Lab materials: `isdn3000e-lab1-tasks/`

Today's Agenda

🐧 Module 1: The Shell (60 min)

Your command-line superpower

⚙️ Module 2: g++ Compiler (45 min)

From source code to executable

📦 Module 3: CMake (45 min)

Automating the build process

Learning Philosophy

> "I hear and I forget. I see and I remember. **I do and I understand.**"

— Confucius (paraphrased)

💻 Each module has hands-on exercises.
📁 Lab materials: `isdn3000e-lab1-tasks/`

Today's Agenda

🐧 Module 1: The Shell (60 min)

Your command-line superpower

⚙️ Module 2: g++ Compiler (45 min)

From source code to executable

📦 Module 3: CMake (45 min)

Automating the build process

❓ Q&A (30 min)

Learning Philosophy

> "I hear and I forget. I see and I remember. **I do and I understand.**"

— Confucius (paraphrased)

💻 Each module has hands-on exercises.
📁 Lab materials: `isdn3000e-lab1-tasks/`

Today's Agenda

🐧 Module 1: The Shell (60 min)

Your command-line superpower

⚙️ Module 2: g++ Compiler (45 min)

From source code to executable

📦 Module 3: CMake (45 min)

Automating the build process

❓ Q&A (30 min)

Wrap-up and discussion

Learning Philosophy

> "I hear and I forget. I see and I remember. **I do and I understand.**"

— Confucius (paraphrased)

💻 Each module has hands-on exercises.
📁 Lab materials: `isdn3000e-lab1-tasks/`

Module 1

The Shell 🐧

Your textual interface to the computer

What is the Shell?

GUI vs CLI

What is the Shell?

GUI vs CLI

- **GUI** — Graphical User Interface
  - Click buttons, drag windows
  - Limited to what designers provide

What is the Shell?

GUI vs CLI

- **GUI** — Graphical User Interface
  - Click buttons, drag windows
  - Limited to what designers provide
- **CLI** — Command Line Interface
  - Type commands, compose programs
  - **Unlimited** expressiveness

What is the Shell?

GUI vs CLI

- **GUI** — Graphical User Interface
  - Click buttons, drag windows
  - Limited to what designers provide
- **CLI** — Command Line Interface
  - Type commands, compose programs
  - **Unlimited** expressiveness

Why Learn the Shell?

1. **Speed** — Faster than clicking
2. **Automation** — Script repetitive tasks
3. **Power** — Combine simple tools
4. **Remote** — SSH into servers
5. **ROS** — All ROS tools use the terminal

Opening a Terminal

🐧 Linux

`Ctrl + Alt + T`

or search "Terminal"

🍎 macOS

`Cmd + Space`

then type "Terminal"

🪟 Windows

Install **WSL2**

Windows Subsystem for Linux

Opening a Terminal

🐧 Linux

`Ctrl + Alt + T`

or search "Terminal"

🍎 macOS

`Cmd + Space`

then type "Terminal"

🪟 Windows

Install **WSL2**

Windows Subsystem for Linux

The Prompt

```
user@machine:~$
```

- `user` — Your username
- `machine` — Computer name
- `~` — Current directory (home)
- `$` — Ready for input (non-root)

Your First Commands

Navigation

```
 pwd           # Where am I?
ls            # What's here?
ls -la        # Show details + hidden
cd folder     # Go into folder
cd ..         # Go up one level
cd ~          # Go home
```

File Operations

```
 mkdir mydir   # Create directory
touch file.txt # Create empty file
cp a.txt b.txt # Copy file
mv a.txt b.txt # Move/rename file
rm file.txt   # Delete file (careful!)
rm -r folder  # Delete folder
```

Your First Commands

Navigation

```
 pwd           # Where am I?
ls            # What's here?
ls -la        # Show details + hidden
cd folder     # Go into folder
cd ..         # Go up one level
cd ~          # Go home
```

File Operations

```
 mkdir mydir    # Create directory
touch file.txt # Create empty file
cp a.txt b.txt # Copy file
mv a.txt b.txt # Move/rename file
rm file.txt   # Delete file (careful!)
rm -r folder  # Delete folder
```

💡 Pro Tips

- Press `Tab` to auto-complete paths
- Use `↑` and `↓` to navigate command history
- `Ctrl+C` to cancel a running command
- `Ctrl+L` to clear the screen

Understanding Paths

Absolute Paths

Start from root `/`

```
/home/student/projects/robot
/usr/bin/python3
/etc/ros/rosdep/sources.list
```

Always leads to the same place

Relative Paths

Start from current directory

```
./build/main        # Current folder
../data/config.yaml # Parent folder
~/catkin_ws         # Home shortcut
```

Result depends on where you are

Special Directories

`.` current | `..` parent | `~` home | `/` root

Reading & Searching Files

Viewing Content

```
 cat file.txt       # Print entire file
head -n 10 file.txt # First 10 lines
tail -n 10 file.txt # Last 10 lines
less file.txt       # Scroll through file
```

Finding Things

```
 # Find files by name
find . -name "*.cpp"

# Search inside files
grep "error" log.txt
grep -r "TODO" src/

# Count matches
grep -c "pattern" file
```

Reading & Searching Files

Viewing Content

```
 cat file.txt       # Print entire file
head -n 10 file.txt # First 10 lines
tail -n 10 file.txt # Last 10 lines
less file.txt       # Scroll through file
```

Finding Things

```
 # Find files by name
find . -name "*.cpp"

# Search inside files
grep "error" log.txt
grep -r "TODO" src/

# Count matches
grep -c "pattern" file
```

💡 Modern Alternatives

- `bat` instead of `cat` (syntax highlighting)
- `fd` instead of `find` (simpler syntax)
- `ripgrep` (`rg`) instead of `grep` (faster)
- `eza` instead of `ls` (better formatting)

Pipes & Redirection

The Unix Philosophy: Do one thing well, then compose.

The Pipe Operator `|`

Connect the output of one program to the input of another.

```
 ls -la | grep ".cpp"           # List only .cpp files
cat log.txt | grep "ERROR" | wc -l  # Count error lines
history | tail -20              # Last 20 commands
```

Pipes & Redirection

The Unix Philosophy: Do one thing well, then compose.

The Pipe Operator `|`

Connect the output of one program to the input of another.

```
 ls -la | grep ".cpp"           # List only .cpp files
cat log.txt | grep "ERROR" | wc -l  # Count error lines
history | tail -20              # Last 20 commands
```

Redirection

```
 echo "Hello" > file.txt    # Write to file (overwrite)
echo "World" >> file.txt   # Append to file
./program < input.txt      # Read from file
./program 2> errors.log    # Redirect errors only
./program &> all.log       # Redirect everything
```

🧪 Exercise: Shell Scavenger Hunt

Time: 15 minutes

Download and explore `isdn3000e-lab1-tasks/module1-shell/`:

1. Find all `.txt` files in the `hunt/` directory
2. Find the file containing the word "SECRET"
3. Count how many files contain the word "TODO"
4. **Bonus:** Find the 3 largest files

🧪 Exercise: Shell Scavenger Hunt

Time: 15 minutes

Download and explore `isdn3000e-lab1-tasks/module1-shell/`:

1. Find all `.txt` files in the `hunt/` directory
2. Find the file containing the word "SECRET"
3. Count how many files contain the word "TODO"
4. **Bonus:** Find the 3 largest files

Hints

```
find hunt/ -name "*.txt"                # Task 1
grep -r "SECRET" hunt/                  # Task 2
grep -rl "TODO" hunt/ | wc -l           # Task 3
find hunt/ -type f -exec ls -s {} \; | sort -n | tail -3  # Bonus
```

Module 2

The g++ Compiler ⚙️

From source code to executable

What is Compilation?

`Source Code (.cpp) → Compiler → Executable (binary)`

1. Preprocess

`#include`, `#define`

Expands macros

2. Compile

C++ → Assembly

Syntax check

3. Assemble

Assembly → Object

Machine code

4. Link

Objects → Executable

Resolve symbols

What is Compilation?

`Source Code (.cpp) → Compiler → Executable (binary)`

1. Preprocess

`#include`, `#define`

Expands macros

2. Compile

C++ → Assembly

Syntax check

3. Assemble

Assembly → Object

Machine code

4. Link

Objects → Executable

Resolve symbols

g++ in Action

```
g++ main.cpp -o myprogram   # Compile and link
./myprogram                 # Run the program
```

Your First C++ Program

`hello.cpp`

```cpp
#include <iostream>

int main() {
    std::cout << "Hello, ISDN3000E!"
              << std::endl;
    return 0;
}
```

Build & Run

```
# Compile
g++ hello.cpp -o hello

# Run
./hello

# Output
Hello, ISDN3000E!
```

Your First C++ Program

`hello.cpp`

```cpp
#include <iostream>

int main() {
    std::cout << "Hello, ISDN3000E!"
              << std::endl;
    return 0;
}
```

Build & Run

```bash
# Compile
g++ hello.cpp -o hello

# Run
./hello

# Output
Hello, ISDN3000E!
```

Breaking it Down

- `#include <iostream>` — Include standard I/O library
- `int main()` — Entry point of program
- `std::cout` — Standard output stream
- `return 0` — Exit successfully

Essential g++ Flags

Commonly Used

Flag	Purpose
`-o name`	Output filename
`-std=c++17`	C++ standard
`-Wall`	Enable warnings
`-g`	Debug symbols

Include & Link

Flag	Purpose
`-I path`	Include directory
`-L path`	Library directory
`-l name`	Link with library
`-c`	Compile only

Essential g++ Flags

Commonly Used

Flag	Purpose
`-o name`	Output filename
`-std=c++17`	C++ standard
`-Wall`	Enable warnings
`-g`	Debug symbols

Include & Link

Flag	Purpose
`-I path`	Include directory
`-L path`	Library directory
`-l name`	Link with library
`-c`	Compile only

💡 Recommended: `g++ -std=c++17 -Wall -Wextra -g main.cpp -o main`

Multi-File Projects

`math_utils.h`

```
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

int add(int a, int b);
int multiply(int a, int b);

#endif
```

Declaration (interface)

`math_utils.cpp`

```
#include "math_utils.h"

int add(int a, int b) {
    return a + b;
}

int multiply(int a, int b) {
    return a * b;
}
```

Implementation

`main.cpp`

```
#include <iostream>
#include "math_utils.h"

int main() {
    std::cout << add(2, 3)
              << std::endl;
    return 0;
}
```

Usage

Multi-File Projects

`math_utils.h`

```cpp
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

int add(int a, int b);
int multiply(int a, int b);

#endif
```

Declaration (interface)

`math_utils.cpp`

```cpp
#include "math_utils.h"

int add(int a, int b) {
    return a + b;
}

int multiply(int a, int b) {
    return a * b;
}
```

Implementation

`main.cpp`

```cpp
#include <iostream>
#include "math_utils.h"

int main() {
    std::cout << add(2, 3)
              << std::endl;
    return 0;
}
```

Usage

Building Multi-File Projects

```bash
# Compile each source file separately
g++ -c main.cpp -o main.o
g++ -c math_utils.cpp -o math_utils.o

# Link object files together
g++ main.o math_utils.o -o calculator
```

Header Guards

The Problem

If a header is included twice → **redefinition errors**

The Traditional Solution

```
#ifndef UNIQUE_NAME_H
#define UNIQUE_NAME_H
// Header content
#endif
```

The Modern Solution ✅

```
#pragma once

// Header content
```

💡 Use `#pragma once` for new projects - simpler and supported by all modern compilers.

Common Compilation Errors

Compile-Time Errors

```
error: expected ';' after expression
```

→ Missing semicolon

```
error: 'cout' was not declared
```

→ Missing `#include` or `std::`

```
error: use of undeclared identifier
```

→ Typo or missing declaration

Link-Time Errors

```
undefined reference to 'add(int, int)'
```

→ Declaration exists but no implementation

→ Or forgot to include `.cpp` in compilation

Fix: Make sure all `.cpp` files are compiled and linked!

Common Compilation Errors

Compile-Time Errors

```
error: expected ';' after expression
```

→ Missing semicolon

```
error: 'cout' was not declared
```

→ Missing `#include` or `std::`

```
error: use of undeclared identifier
```

→ Typo or missing declaration

Link-Time Errors

```
undefined reference to 'add(int, int)'
```

→ Declaration exists but no implementation

→ Or forgot to include `.cpp` in compilation

Fix: Make sure all `.cpp` files are compiled and linked!

💡 Debugging Tips

1. Read the **first** error message carefully
2. Check the **line number** mentioned
3. Compile with `-Wall -Wextra` for more hints

🧪 Exercise: Multi-File Project

Time: 20 minutes

Navigate to `isdn3000e-lab1-tasks/module2-gpp/`:

1. Split `calculator.cpp` into 3 files
2. Compile manually with g++
3. **Bonus:** Add `subtract()` function

Expected Structure

```
module2-gpp/
├── calculator.h
├── calculator.cpp
└── main.cpp
```

Module 3

CMake 📦

Automating the build process

Why CMake?

The Problem

As projects grow, manual compilation becomes painful:

```
g++ -c main.cpp -o main.o
g++ -c utils.cpp -o utils.o
g++ -c network.cpp -o network.o
g++ -c database.cpp -o database.o
g++ -c graphics.cpp -o graphics.o
# ... 50 more files ...
g++ *.o -o myapp -lpthread -lssl
```

Imagine typing this every time...

Why CMake?

The Problem

As projects grow, manual compilation becomes painful:

```
 g++ -c main.cpp -o main.o
g++ -c utils.cpp -o utils.o
g++ -c network.cpp -o network.o
g++ -c database.cpp -o database.o
g++ -c graphics.cpp -o graphics.o
# ... 50 more files ...
g++ *.o -o myapp -lpthread -lssl
```

Imagine typing this every time...

The Solution

CMake: **C**ross-platform **Make**

- Describe your project **once**
- CMake generates build instructions
- Works on Linux, macOS, Windows
- **ROS uses CMake** for all packages!

CMake Basics

`CMakeLists.txt`

```cmake
cmake_minimum_required(VERSION 3.16)

# Project name and language
project(Calculator LANGUAGES CXX)

# Set C++ standard
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# Create executable from sources
add_executable(calculator
    main.cpp
    math_utils.cpp
)
```

Build Process

```bash
# Create build directory
mkdir build && cd build

# Generate build files
cmake ..

# Compile
make

# Run
./calculator
```

Or use the modern way:

```bash
cmake -B build
cmake --build build -- -j$(nproc)
./build/calculator
```

Key CMake Commands

Project Structure

```
 # Minimum CMake version
cmake_minimum_required(VERSION 3.16)

# Project declaration
project(MyProject
    VERSION 1.0
    LANGUAGES CXX
)

# C++ standard
set(CMAKE_CXX_STANDARD 17)
```

Building Targets

```
 # Create an executable
add_executable(myapp
    main.cpp
    utils.cpp
)

# Create a library
add_library(mylib
    lib.cpp
)

# Link libraries to target
target_link_libraries(myapp
    PRIVATE mylib
)
```

Include Directories & Libraries

```cmake
# Add include paths
target_include_directories(myapp
    PRIVATE ${CMAKE_SOURCE_DIR}/include
)

# Find and use external packages
find_package(Threads REQUIRED)
target_link_libraries(myapp PRIVATE Threads::Threads)

# Compiler warnings
target_compile_options(myapp PRIVATE -Wall -Wextra)
```

Include Directories & Libraries

```cmake
# Add include paths
target_include_directories(myapp
    PRIVATE ${CMAKE_SOURCE_DIR}/include
)

# Find and use external packages
find_package(Threads REQUIRED)
target_link_libraries(myapp PRIVATE Threads::Threads)

# Compiler warnings
target_compile_options(myapp PRIVATE -Wall -Wextra)
```

💡 PUBLIC vs PRIVATE vs INTERFACE

- **PRIVATE** → Only for this target
- **PUBLIC** → This target AND its dependents
- **INTERFACE** → Only for dependents

A Complete Example

Project Structure

```
my_project/
├── CMakeLists.txt
├── include/
│   └── calculator.h
├── src/
│   ├── calculator.cpp
│   └── main.cpp
└── build/        # Created by cmake
```

CMakeLists.txt

```cmake
cmake_minimum_required(VERSION 3.16)
project(Calculator LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

add_executable(calculator
    src/main.cpp
    src/calculator.cpp
)

target_include_directories(calculator
    PRIVATE ${CMAKE_SOURCE_DIR}/include
)

target_compile_options(calculator
    PRIVATE -Wall -Wextra
)
```

Out-of-Source Builds

× In-Source Build

```
my_project/
├── CMakeLists.txt
├── main.cpp
├── CMakeCache.txt      # Generated
├── CMakeFiles/         # Generated
├── Makefile            # Generated
└── calculator          # Generated
```

Source mixed with build artifacts 😕

✅ Out-of-Source Build

```
my_project/
├── CMakeLists.txt
├── main.cpp
└── build/
    ├── CMakeCache.txt
    ├── CMakeFiles/
    ├── Makefile
    └── calculator
```

Clean separation! Easy to `rm -rf build`

Out-of-Source Builds

× In-Source Build

```
my_project/
├── CMakeLists.txt
├── main.cpp
├── CMakeCache.txt      # Generated
├── CMakeFiles/         # Generated
├── Makefile            # Generated
└── calculator          # Generated
```

Source mixed with build artifacts 😕

✅ Out-of-Source Build

```
my_project/
├── CMakeLists.txt
├── main.cpp
└── build/
    ├── CMakeCache.txt
    ├── CMakeFiles/
    ├── Makefile
    └── calculator
```

Clean separation! Easy to `rm -rf build`

💡 Always Use Out-of-Source Builds

```
 cmake -B build      # Configure into 'build' folder
cmake --build build # Build from that folder
```

🧪 Exercise: CMake Project

Time: 20 minutes

Navigate to `module3-cmake/starter/` :

1. Write a `CMakeLists.txt`
2. Build using CMake
3. Run the executable
4. **Bonus:** Add compiler warnings

Build Commands

```
 cd starter
cmake -B build
cmake --build build
./build/calculator
```

Summary

🐧 The Shell

- Navigate with `cd`, `ls`, `pwd`
- Manipulate with `cp`, `mv`, `rm`
- Search with `grep`, `find`
- Compose with `|` pipes

⚙️ g++

- `g++ -o out src.cpp`
- Use `-Wall -Wextra`
- Separate headers & source
- Link with `g++ *.o`

📦 CMake

- `cmake_minimum_required`
- `project()`, `add_executable()`
- Out-of-source builds
- `cmake -B build && cmake --build build`

What’s Next?

Lab 2: Introduction to Git / Clion / Debug / AI Coding Agent

Q&A

🙋

Thank you!