C/C++ Programming

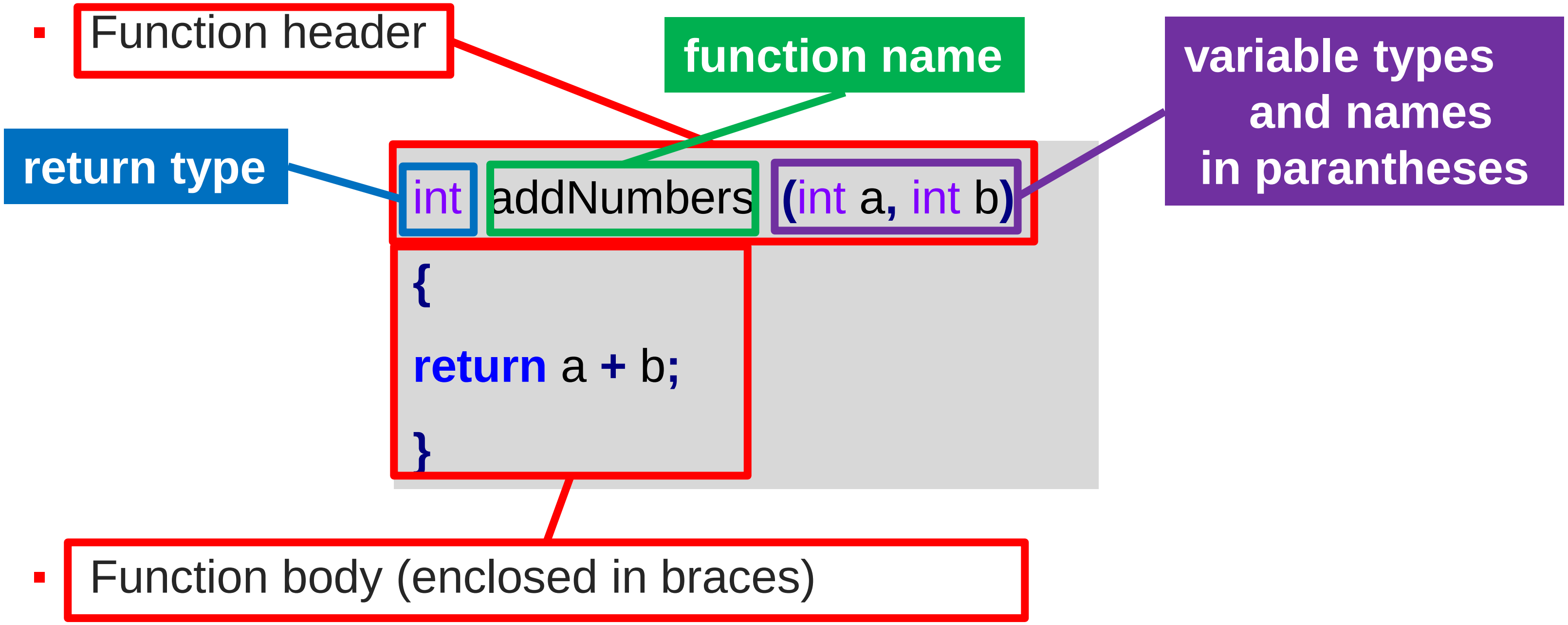
Lecture 1

Background: Lecture Outline

- Assumption
  - Basic programming background from CENG240
  - Interest in C/C++ programming as prerequisite for data structures
- Lecture Focus
  - Transition from Python to C/C++ syntax
  - Introduction of procedural C/C++ programming
  - Emphasis of special features of C/C++
- Key Topics
  - Syntax, variables, data types, operators
  - Control structures, functions, header/source files, compilation process
- Basic Motivation
  - Performance: Closer to hardware, faster execution
  - Control: Manual memory management, explicit typing
  - Portability: Widely used in industry, embedded systems

Background: General Structure

- C/C++ programs consist of modules called functions
- Every statement is contained in a function
- Main components of a function



- Function header

function name

variable types and names in parantheses

return type

```
int addNumbers (int a, int b)
{
return a + b;
}
```

- Function body (enclosed in braces)

Background: Main Function

- A C/C++ program contains at least one function called main( )
- Always returns int (standard requirement)
- Return value communicates exit status to operating system
- Optional input arguments (details later)
  - argc: number of arguments
  - argv: array of argument strings

example statement

```
int main (int argc, char* argv[])
{
        std::cout << "Empty main function" << std::endl;
        return 0;
}
```

Code: Lecture1_main_function Block 1

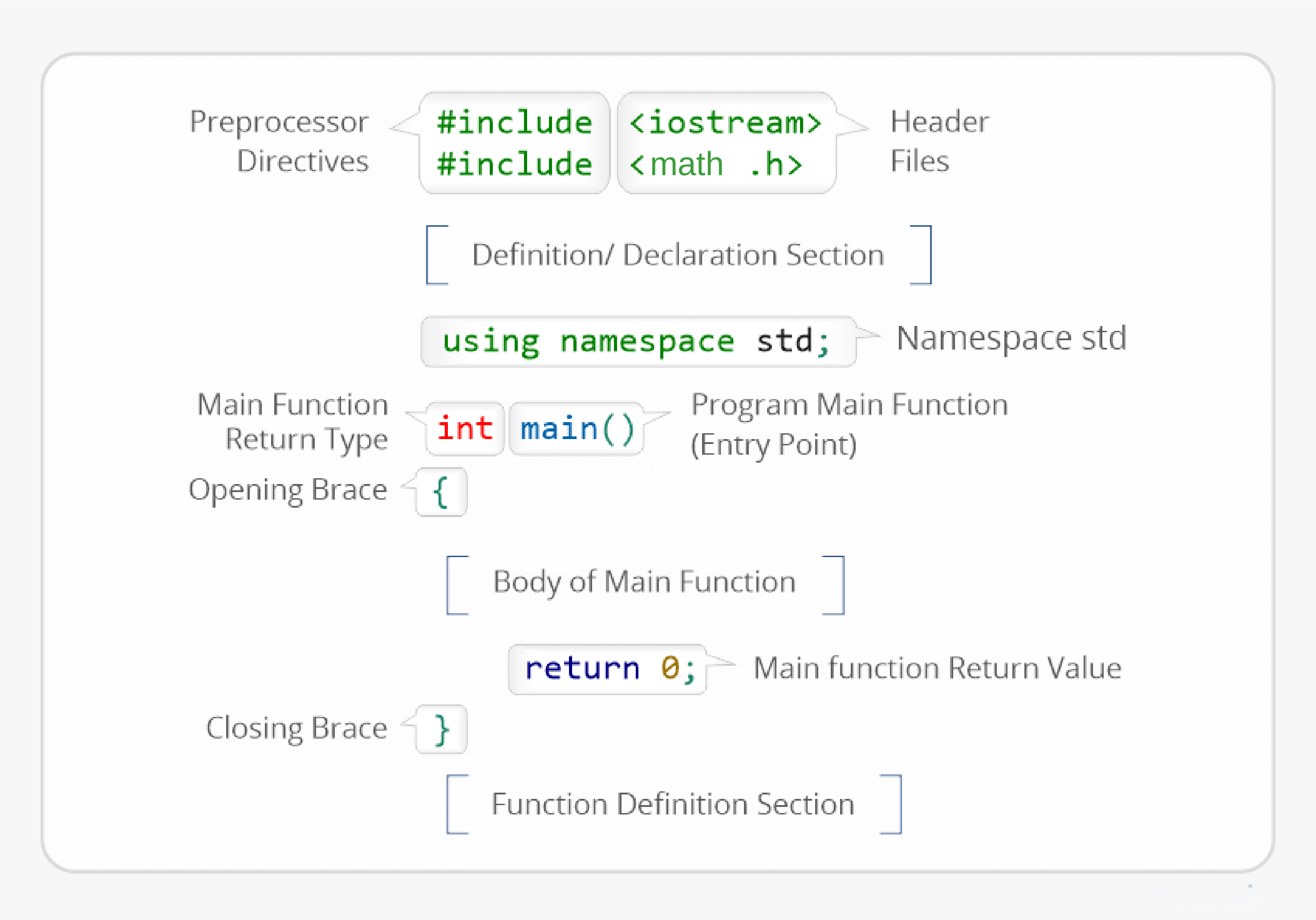
Background: Basic Syntax

- Statements end with a semicolon `;`
- Braces `{}` define code blocks (instead of indentation in Python)
- Case sensitivity: `Var` and `var` are different identifiers
- Comments
  - `//` single-line
  - `/* ... */` multi-line

```
int main(int argc, char* argv[]) {
    std::cout << "Simple main function" << std::endl;
    // We open a code block
    {
        // Next, we declare a variable
        int a;
        // Next, we assign a value to the variable
        a = 5;
    }
    // The variable a is no longer there
    return 0;
}
```

Code: Lecture1_main_function Block 2

Background: Structure of a C/C++ Program



```
#include <iostream>
#include <math .h>

using namespace std;

int main()
{

    return 0;
}
```

Preprocessor Directives
Header Files
Definition/ Declaration Section
Namespace std
Main Function Return Type
Program Main Function (Entry Point)
Opening Brace
Body of Main Function
Main function Return Value
Closing Brace
Function Definition Section

Background: Structure of a C/C++ Program

```cpp
#include <iostream>           // Preprocessor directive
using namespace std;      // Namespace declaration

// Function prototype (declaration)
int add(int a, int b);

int main() {               // Main function
    int result = add(2, 3); // Function call statement
    cout << "Result = " << result << endl; // Debug output statement
    return 0; // Return statement
}

// Function definition
int add(int a, int b) {
    return a + b;
}
```

Code: Lecture1_main_function Block 3

Background: Preprocessor Directives

- Header files contain predefined values, routines, or libraries
  - Their filenames usually end in .h or .hpp
  - Compiler must be told what to do before compiling the program

    → Include a preprocessor directive to use these routines
- Preprocessor directives begin with a pound sign (#)
- #include preprocessor directive tells the compiler to include a file
  - #include <cmath>
  - #include <iostream>

    modern C++ header
  - #include “math.h”
  - #include “my_lib.h”

    old-style C header or own header
- #define preprocessor directives represent substitution “Macros”
  - #define PI 3.14159 → “PI” in program is replaced by “3.14159”
  - #define SQUARE(x) ((x) * (x))

Background: Variables

- Definition: A variable is a named memory location that stores a value of a specific data type
- Variable Declaration
  - int size;
  - float temperature;

Allocate memory and tell the compiler type and name

- Variable Assignment
  - size = 12;
  - temperature = 20.56;

Assign a value to the variable

- Additional Facts
  - Combined declaration and initialization: int size = 12;
  - Uninitialized variables contain “garbage”

Background: Data Types

- A data type tells the compiler
  - What kind of value the variable can hold
  - How much memory to allocate
  - What operations are valid on it

```
int int_val = 20;
float float_val = 15.723;
double double_val = 0.55;
char char_val = 'a';
bool bool_val = true;
```

- Fundamental data types
  - int → integer numbers (typically 4 bytes)
  - float → single-precision decimal (4 bytes)
  - double → double-precision decimal (8 bytes)
  - char → single character (1 byte)
  - bool → true/false (1 byte, conceptually 1 bit)
  - void → no value (used for nothing)
  - auto → type deduction from initializer

```
auto x = 10;      // int
auto y = 3.14;    // double
auto c = 'A';     // char
```

Background: Common Data Types

Type	Bytes	Range
short	2	−32768 to 32767 ($-2^{15}+1$ to $2^{15}-1$)
unsigned short	2	0 to 65,535 (0 to $2^{16}-1$)
int	4	−2,147,483,648 to 2,147,483,647 ($-2^{31}+1$ to $2^{31}-1$)
unsigned int	4	0 to 4,294,967,295 (0 to $2^{32}-1$)
long	4 or 8	Depends on operating system
unsigned long	4 or 8	
long long	8	−9.22e18 to 9.22e18 ($-2^{63}+1$ to $2^{63}-1$)
unsigned long long	8	0 to $1.84 \cdot 10^{19}$ (0 to $2^{64}-1$)
float	4	
double	8	
long double	16	even higher precision (up to 18–19 digits)

Background: Arithmetic Operators

- Arithmetic
  - Addition: +
  - Subtraction: -
  - Multiplication: *
  - Division: /
- Compound Assignment
  - `b op = a` means `b = b op a;`
- Modulus
  - %
- Increment/Decrement
  - Increment: ++
  - Decrement: --

```
int a = 10;
int b = 20;
float c = 5.3;
float d = 2.5;

int e = a + b; // Result: e = 30
float f = d - c; // Result: f = -2.8
float g = c*d; // Result: g = 13.25
int h = b/a; // Result: h = 2
b += a; // Meaning: b = b + a = 30;
d *= c; // Meaning: d = d * c = 13.25;

int j = b & a; // Result j = 0

a++; // Result: a = 11

b--; // Result b = 19
```

Background: Integer Constant Suffixes

Explanation

- Suffixed integer constants to explicitly specify their type
- Control memory usage, avoid overflows, match function signatures

```
unsigned int x = 42u;
long z = 1000000l;
long long w = 10000000000ll;
unsigned long v = 42ul;
```

Examples

- u → unsigned
- l → long
- ll → long long

Enumerations

- User-defined type consisting of a set of named integer constants
- Improves readability and avoids using “magic numbers”

```
enum Day {
    MONDAY,      // 0
    TUESDAY,     // 1
    WEDNESDAY, // 2
    THURSDAY,    // 3
    FRIDAY         // 4
};
Day today = WEDNESDAY;
```

Background: Variable Qualifiers

- const: Immutable Variable
  - Value fixed after initialization
- volatile: Prevents Optimization
  - Variable may change unexpectedly
  - Hardware registers, multi-threading
- static:
  - Scope restriction in files
  - Life-time extension in functions

    → function output: 1 2 3 …
- extern: Cross-File Declaration
  - Life-time extension in functions

```
const double pi = 3.14159;
// pi = 3.14;  // ERROR: not allowed
```

```
volatile int flag;  // may change
while (flag == 0) {
    // compiler will NOT remove loop
}
```

```
static int localVar = 42;
// Visible only inside this source file
```

```
void counter() {
    static int count = 0; // retains value
    count++;
    cout << count << endl;
}
```

```
File1.cpp
int globalCounter = 0;  // definition
File2.cpp
extern int globalCounter;  // declaration
```

Background: Boolean Operators

- Comparison
  - Boolean result
  - Equal: ==
  - Not equal: !=
  - Less: <
  - Less or equal: <=
  - Greater: >
  - Greater or equal: >=
- Logical Operators
  - On booleans
  - And: &&
  - Or: ||
  - Not: !

```
#include <iostream>
using namespace std;

int main() {
    int x = 5, y = 10;

    // Comparison examples
    cout << (x == y) << endl;   // 0 (false), 5 != 10
    cout << (x < y) << endl;    // 1 (true), 5 < 10

    // Logical operator examples
    bool a = true, b = false;
    cout << (a && b) << endl;   // 0, true AND false
    cout << (a || b) << endl;   // 1, true OR false
    cout << (!a) << endl;       // 0, NOT true

    return 0;
}
```

Code: Lecture1_main_function Block 4

Background: Boolean Operators

- Bitwise
  - Operators are applied bit by bit
  - And: &
  - Or: |
  - XOR: ^
  - Not: ~
  - Shift left: <<
  - Shift right: >>

```
#include <iostream>
#include <bitset>
using namespace std;

int main() {
    short a = 30000;   // 0111 0101 0011 0000
    short b = 12345;   // 0011 0000 0011 1001
    cout << "a      = " << bitset<16>(a) << endl;
    cout << "b      = " << bitset<16>(b) << endl;
    cout << "a & b  = " << bitset<16>(a & b) << endl;
    // 0011 0000 0011 0000
    cout << "a ^ b  = " << bitset<16>(a ^ b) << endl;
    // 0100 0101 0000 1001
    cout << "a << 3 = " << bitset<16>(a << 3) << endl;
    // 1010 1001 1000 0000
    cout << "a >> 5 = " << bitset<16>(a >> 5) << endl;
    // 0000 0011 1010 1001
}
```

Code: Lecture1_main_function Block 5

Background: Casting

- Basic Information
  - Casting means converting one data type into another
  - Necessary when types don’t match in an expression or assignment
- Implicit Casting
  - Done automatically by the compiler
  - “Smaller” types are promoted to “larger” types to prevent data loss

```
int a = 5;
double b = 2.5;
double result = a * b;   // int a → converted double -> result = 12.5
```

  - Data loss if wrong data type is chosen as result

```
int a = 5;
double b = 2.5;
int result = a * b;   // result is converted to int -> result = 12
```

Background: Casting

- Explicit Casting
  - Done manually by the programmer
- C-Style Cast
  - Write target data type in parenthesis before variable name

```
double pi = 3.14;
int x = (int) pi;   // cast from double to int result = 3 (truncated not rounded!)
char letter = 'A'; // 'A' has ASCII value 65
float value = (float)letter; // value = 65.0
```

- C++-Style Cast

```
double pi = 3.14;
int x = static_cast<int>(pi);   // truncates to 3
```

  - For later: `const_cast, reinterpret_cast, dynamic_cast`

Background: Keywords in C/C++

- Definition
  - Keywords are reserved words in the language (case-sensitive!)
  - Have special meaning to the compiler
  - Cannot be used as variable names, function names, or identifiers
- Common Keywords
  - Data types: int, float, double, char, void, bool
  - Control flow: **if**, **else**, **switch**, **case**, **for**, **while**, **do**, **break**, **continue**
  - Storage classes: static, extern, register, auto
  - Qualifiers: const, volatile, signed, unsigned, long, short
  - Other: **return**, **goto**, **sizeof**, **typedef**, **struct**, **enum**
- Some keywords differ between C and C++
  - bool, **namespace**, **new**, **delete** are C++-only

Basics: Debugging

- Necessity
  - Trace program flow and see variable values at runtime
  - Detect logic errors and identify unexpected behavior
- Usage of “cout” in C++
  - Requires include of “iostream” (display) and “iomanip” (formatting)
  - Common commands
    - Text output: “`std::cout <<`‟
    - Newline: “`std:endl`”
  - Formatting
    - Fixed-point notation: `cout << fixed`
    - Floating-point precision: `cout << setprecision(n)`
    - Scientific notation: `cout << scientific`

Basics: Debugging

```
#include <iostream>
#include <iomanip>   // for formatting
using namespace std; // cout belongs to namespace std

int main() {
    int x = 5;
    double y = 2.578;
    // Printing variable values
    cout << "x = " << x << ", y = " << y << endl;
    // Formatting floating-point output
    cout << fixed << setprecision(2); // Display as decimal with two fractional digits
    cout << “x ” << x << “ y (2 decimals) = " << y << endl;
    return 0;
}
```

Code: Lecture1_main_function Block 6

Conditionals: If Else

- General Structure

```
if(“logical statement”){
        statements
}
else if(“logical statement”){
        other statements
}
…..
else{
        other statements
}
```

- Note
  - If there is a single statement, braces are not needed

```
#include <iostream>
using namespace std;

int main() {
                int temp;
                cout << "Enter temperature in °C: ";
                cin >> temp; // Terminal input

                if (temp > 30) {
                    cout << "It's hot!" << endl;
                }
                else if (temp > 15) {
                    cout << "It's mild." << endl;
                }
                else
                    cout << "It's cold." << endl;
    return 0;
}
```

Code: Lecture1_main_function Block 7

Conditionals: Switch Case

- Definition
  - Control structure to test a variable against multiple **constant values**
  - Alternative to writing many else if statements
- How it works
  - The value of the expression is compared to each case
  - When a match is found, the statements under that case run
  - break; ends the switch, otherwise execution continues
  - If no case matches, default: is executed (optional)
- Restrictions
  - Works only with integral types (int, char, enums), not double or string
  - Case labels must be constant values, not variables

Conditionals: Switch Case Example

```
#include <iostream>
using namespace std;

int main() {
    int a, b;
    char op;
    cout << "Enter operation with '+', '-', '*'. E.g. 5 + 3";
    cin >> a >> op >> b;
    switch (op) {
        case '+': // Compare 'op' to '+'
                cout << a << " + " << b << " = " << (a + b) << endl;
                break;
        case '-': cout << a << " - " << b << " = " << (a - b) << endl; break;
        case '*': cout << a << " * " << b << " = " << (a * b) << endl; break;
        default: cout << "Invalid operator" << endl;
    }
    return 0;
}
```

Code: Lecture1_main_function Block 8

Loops: Overview

- Purpose
  - Repeat a statement or group of statements multiple times
- **for** loop
  - Used when the number of iterations is known

```
for (initialization; condition; update) { ... }
```

- **while** loop
  - Used when the number of iterations is not known in advance

```
while (condition) { ... }
```

- **do while** loop
  - Similar to while, but guarantees at least one execution

```
do { ... } while (condition);
```

Loops: for Example

```
#include <iostream>
using namespace std;

int main() {
    // for loop
    for (int i = 1; i <= 5; i++) {
        cout << i << " ";
    }
    cout << endl;

    // while loop
    int j = 1;
    while (j <= 5) {
        cout << j << " ";
        j++;
    }
    cout << endl;
```

```
// do-while loop
    int k = 1;
    do {
        cout << k << " ";
        k++;
    } while (k <= 5);
    cout << endl;

    return 0;
}
```

Expected Output

```
> 1 2 3 4 5
> 1 2 3 4 5
> 1 2 3 4 5
```

Code: Lecture1_main_function Block 9

Functions: Explanation

- Definition
  - A function is a block of code that performs a specific task
  - Use of functions promotes modularity and code reuse
- Basic Structure

```
return_type function_name(parameters) {
    // function body
    return value;   // optional (depends on return_type)
}
```

  - Return type: Type of value returned (int, double, void, …)
  - Name: Identifier for the function
  - Parameters: Input values (optional)
  - Body: Statements to execute

Functions: Examples

```cpp
// No parameters, no return value
void greet() {
    cout << "Hello, world!" << endl;
}

// Parameters, no return value
void printSquare(int x) {
    cout << x << "^2 = " << (x * x) << endl;
}
```

```cpp
// No parameters, return value
int getFive() {
    return 5;
}

// Parameters, return value
double average(double a, double b) {
    double c = (a + b) / 2.0;
    return c;
}
```

- Remarks
  - C/C++ functions can have at most one output variable
  - Input parameters are copied when passed to a function
  - Memory for variables declared in a function is allocated when the function is called and released when it ends

Functions: Declaration and Definition

- Function Declaration
  - Tells the compiler the function’s name, return type, and parameters
  - Placed at the beginning of the cpp file or in a header file (.h, .hpp)
  - Ends with a semicolon ;
- Function Definition
  - Contains the actual body of the function
  - Placed in the main file or another source file (.cpp)
- Call to Function
  - Done from main.cpp or other .cpp files
  - If declared in a .h file, the header file needs to be included

Functions: Example in Single Main File

```
#include <iostream>
using namespace std;
// Function declarations (prototypes)
int add(int a, int b);
double average(double x, double y);

int main() {
    int a = 6, b = 4;

    cout << "Sum: " << add(a, b) << endl;
    cout << "Average: " <<
            average(a, b) << endl;

    return 0;
}
```

Code: Lecture1_main_function Block 10

```
// Function definitions (after main)
int add(int a, int b) {
    return a + b;
}

double average(double x, double y) {
    return (x + y) / 2.0;
}
```

- Remarks
  - Declarations are before main such that the compiler knows the function prototype
  - Definitions are generally after main for readability

Functions: Header and Source Files

File: math_utils.h

```
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

int add(int a, int b); // function prototype
double average(double x, double y);
#endif
```

File: math_utils.cpp

```
#include "math_utils.h"

int add(int a, int b) {         // function
definition
    return a + b;
}
double average(double x, double y) {
    return (x + y) / 2.0;
}
```

Code: Lecture1_header

File: main.cpp

```
#include <iostream>
#include "math_utils.h"    // include
declarations
using namespace std;

int main() {
    int a = 10, b = 20;
    cout << "Sum: " << add(a, b) << endl;
    cout << "Average: " <<
              average(a, b) << endl;
    return 0;
}
```

❑ Remarks

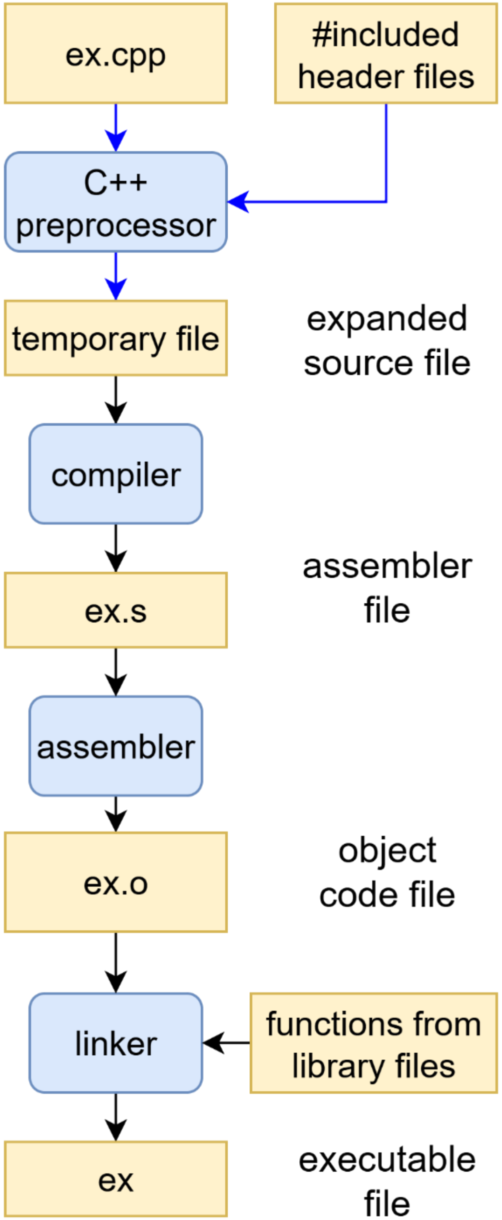
- Declarations and definitions are separated
- Header file has to be included

Compilation: Files

- Header Files (.h)
  - Contain function declarations, constants, macros
  - Included with #include into source files
- Source Files (.cpp, .c)
  - Contain function definitions (actual code)
  - Each compiled separately into object files
- Object Files (.o or .obj)
  - Machine code, but contains unresolved references (functions/variables used)
  - One object file per source file
- Static Libraries (.a or .lib)
  - Collection of object files bundled together
  - Linked directly into the program → increases binary size
- Dynamic Libraries (.so on Linux, .dll on Windows)
  - Program contains only references → smaller binary
  - Shared at runtime and multiple programs can reuse the same library

Compilation: Build Process

- Preprocessing
  - Handles #include, #define, conditional compilation
  - Produces a “pure” source file
- Compilation (proper)
  - Translates C/C++ source into assembly code (human-readable low-level instructions)
- Assembly
  - The assembler converts assembly code into machine code
  - Produces object files (.o, .obj)
- Linking
  - Combines object files + libraries
  - Produces the final executable binary



Compilation: Makefile Components

- ❑ Variables
  - Store compiler, flags, file lists

  → easier to reuse

```
CXX = g++
CXXFLAGS = -Wall -O2 -std=c++17
```

- ❑ Targets
  - What you want to build (e.g., my_ex, clean)
  - Each target has a recipe (commands to run)

```
all: my_ex
```

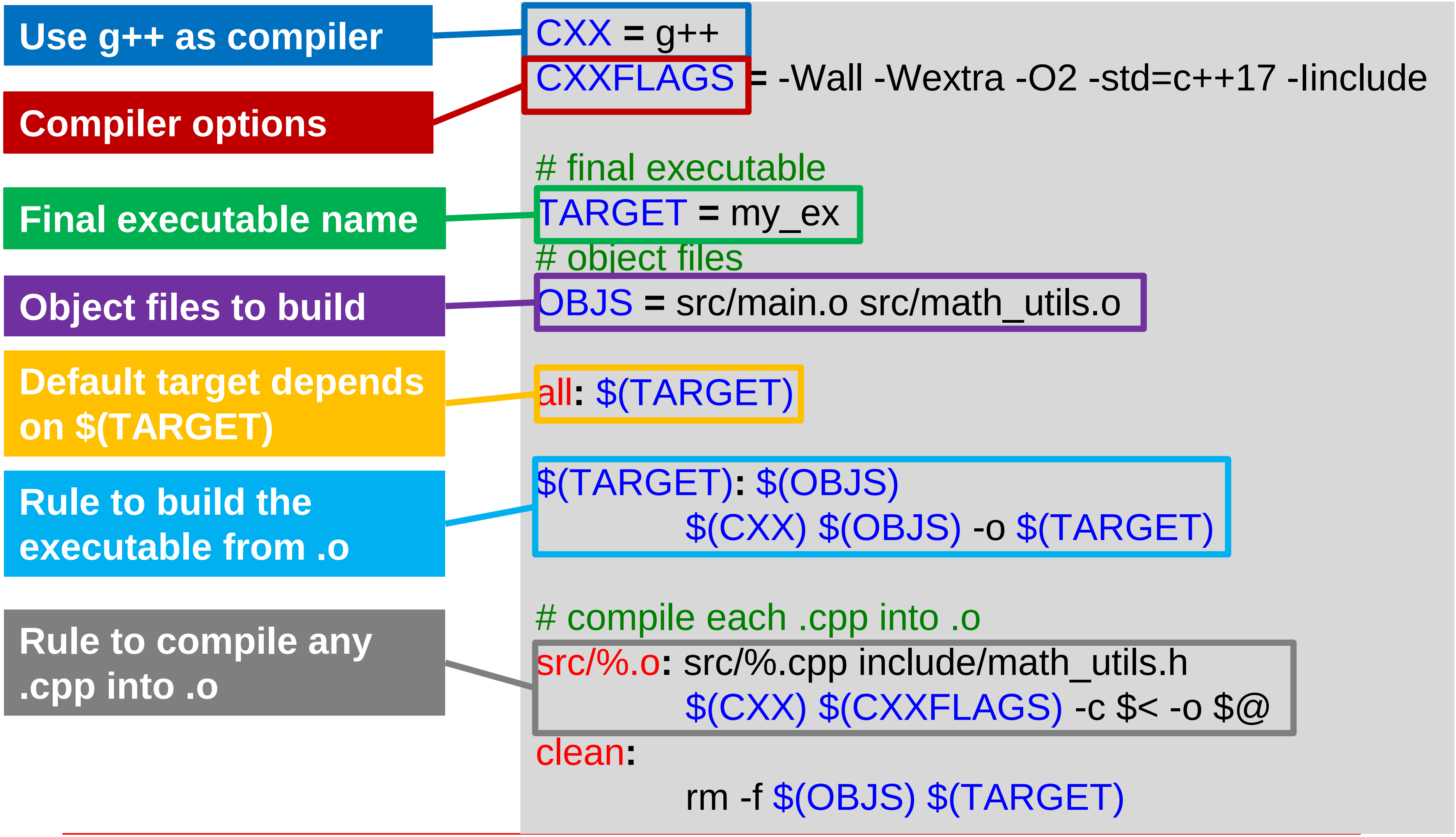
- ❑ Dependencies and Rules
  - Files needed to create a target
  - Commands under a target (must start with Tab)

```
my_ex: main.o utils.o
	$(CXX) $(CXXFLAGS) main.o utils.o -o app
```

- ❑ Pattern Rules
  - Generic instructions (e.g., compile all .cpp → .o)
- ❑ Special Targets
  - clean to remove build artifacts

```
clean:
	rm -f *.o my_ex
```

Compilation: Makefile Example



- **Use g++ as compiler**
- **Compiler options**
- **Final executable name**
- **Object files to build**
- **Default target depends on $(TARGET)**
- **Rule to build the executable from .o**
- **Rule to compile any .cpp into .o**

```
CXX = g++
CXXFLAGS = -Wall -Wextra -O2 -std=c++17 -Iinclude

# final executable
TARGET = my_ex
# object files
OBJS = src/main.o src/math_utils.o

all: $(TARGET)

$(TARGET): $(OBJS)
	$(CXX) $(OBJS) -o $(TARGET)

# compile each .cpp into .o
src/%.o: src/%.cpp include/math_utils.h
	$(CXX) $(CXXFLAGS) -c $< -o $@
clean:
	rm -f $(OBJS) $(TARGET)
```

C and C++ Comparison

Aspect	C	C++
Paradigm	Procedural (function-oriented)	Multi-paradigm: Procedural + Object-Oriented + Generic
Standard Library	Small (stdio, string.h, math.h)	Large (includes C libs + STL: vector, string, map)
I/O	printf, scanf (format strings)	cin, cout with stream operators (>>, <<)
Data Types	Basic types, structs, pointers	Adds bool, string, references, function overloading
Memory Management	malloc, calloc, free	new, delete, RAII, smart pointers (modern C++)
Namespaces	Not supported	Supported (namespace std)
Error Handling	Return codes	Exceptions (try, catch)

Evolution of C++ Standards

- C++98 / C++03
  - First standardized versions
  - Templates, exceptions, namespaces, STL (Standard Template Library)
- C++11 (“Modern C++”)
  - Major upgrade: auto, nullptr, lambdas, range-based for loop, smart pointers, std::thread, move semantics
- C++14
  - Small improvements: generic lambdas, auto return type, relaxed constexpr
- C++17
  - More powerful: structured bindings (auto [a,b]), if constexpr, inline variables
- C++20
  - Big step: concepts (type constraints), ranges, coroutines, modules
- C++23 (latest published)
  - Incremental: standard library refinements, std::expected, improved ranges

C/C++ Programming

Lecture 1