ENPM702

Introductory Robot Programming

L2: Introduction to C++

v2.0

Lecturer: Z. Kootbally

Semester/Year: Summer/2025

MARYLAND APPLIED GRADUATE ENGINEERING

Table of Contents

Changelog

- **v2.0**:
  - Move the section on I/O to reading material slides.
  - Minor visual improvements.
- **v1.0**: Original version.

Conventions

bad practice	
best practice	
code syntax	
example	
exercise	
file	
folder	
C+ guideline	
note	
question	
TODO	
terminology	
warning	
web link	link
package	
resource	
terminal	command

Prerequisites

- [ ] Canvas:
  - [ ] Get the slides.
  - [ ] Get the C++ standard.
  - [ ] Create a project with lecture2.cpp
  - [ ] Update CMakeLists.txt
  - [ ] Run.

```
enpm702_summer2025_cpp
└ lecture2
  └ src
    └ lecture2.cpp
```

Note

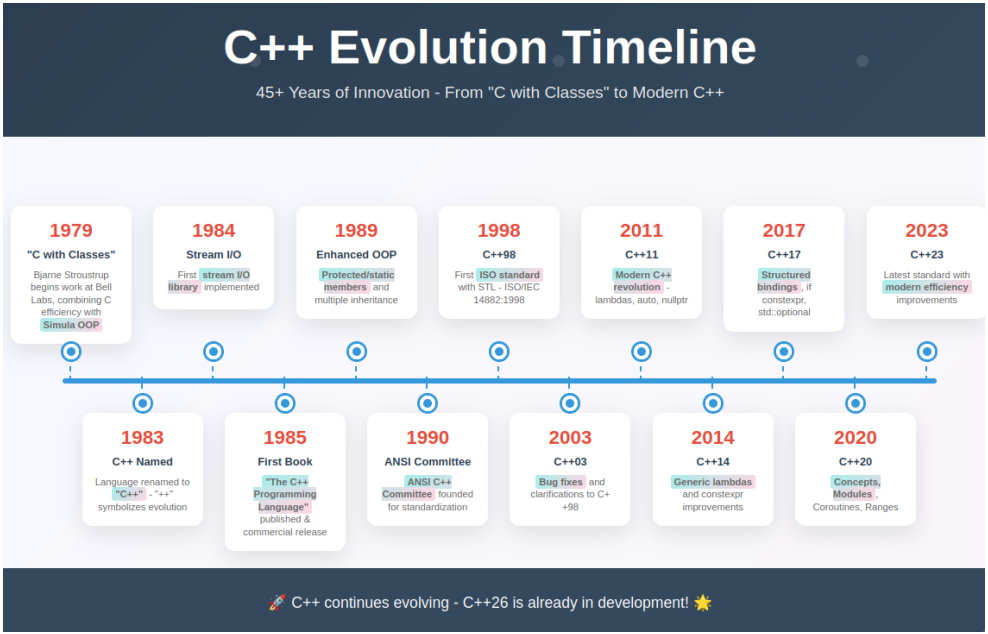
- Moved section on version control to a separate slide deck.
- A recording on version control will be released.
- Reading Materials for next week.
- Quiz #1 next week.
- Team formation by next week.
- Assignment #1 due date changed.

What is C++?

C++ is a high-level, general-purpose programming language that was developed as an extension of the C programming language. It was created by Bjarne Stroustrup at Bell Labs in the early 1980s.

Resources

- **cppreference.com**: An extensive reference site for C++ and its standard library. It includes detailed information on syntax, functions, and C++ standards.
- **cplusplus.com**: A comprehensive resource for learning C++ with tutorials, references, and examples. It's great for beginners and intermediate learners.


C++ Evolution Timeline
45+ Years of Innovation - From "C with Classes" to Modern C++
1979
"C with Classes"
Bjarne Stroustrup begins work at Bell Labs, combining C efficiency with Simula OOP
1983
C++ Named
Language renamed to "C++" - "++" symbolizes evolution
1984
Stream I/O
First stream I/O library implemented
1985
First Book
"The C++ Programming Language" published & commercial release
1989
Enhanced OOP
Protected/static members and multiple inheritance
1990
ANSI Committee
ANSI C++ Committee founded for standardization
1998
C++98
First ISO standard with STL - ISO/IEC 14882:1998
2003
C++03
Bug fixes and clarifications to C++98
2011
C++11
Modern C++ revolution - lambdas, auto, nullptr
2014
C++14
Generic lambdas and constexpr improvements
2017
C++17
Structured bindings, if constexpr, std::optional
2020
C++20
Concepts, Modules, Coroutines, Ranges
2023
C++23
Latest standard with modern efficiency improvements
🚀 C++ continues evolving - C++26 is already in development! 🌟

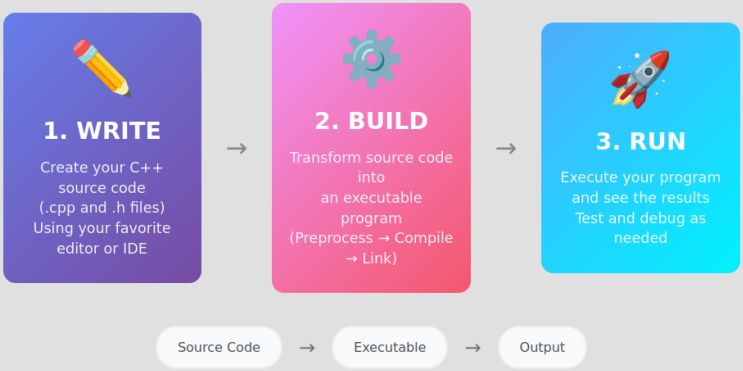
Hardware Interfacing Capabilities

As a compiled language, C++ offers robust mechanisms for low-level hardware interaction.

- **Direct Memory Access**: Achieved through the use of pointers, allowing precise manipulation of memory addresses.
- **Memory-Mapped I/O**: Provides the ability to directly read from and write to hardware registers mapped into memory space.
- **Inline Assembly**: Allows embedding assembly code snippets within C++ code for highly optimized or specific CPU operations.
- **System Calls and APIs**: Interfaces with operating system kernels and hardware-specific Application Programming Interfaces for device driver communication.
- **Specialized Libraries**: Utilizes low-level and platform-specific libraries to interact with peripherals and hardware components.

C++ Development Process

Three Essential Phases Every Developer Follows



```cpp
#include <iostream>

int main(){
    std::cout << "hello, world\n";
    return 0;
}
```

- The function `main()` is the entry point of any C++ program and **must** be defined. Each C++ program can have only one `main()` function.
  - `main()` must return `int` and not `void` (see 📖 ***6.6.1***, page 66).
  - `return 0` can be omitted in `main` (C99 §5.1.2.2.3). Reaching the closing curly bracket returns a value of 0.
- `std::cout << "hello, world\n";`
  - `std::cout <<` is used to print a message in the terminal. We have to include `iostream` to be able to use `std::cout`.
  - The message to print in the terminal is `hello, world`.
  - \n inserts a new line in the terminal after printing `hello, world`
  - The semicolon (`;`) indicates the termination of a statement.

Comments

Comments are primarily for the benefit of developers who read and maintain the code. They provide explanations, clarify code logic, and offer insights into the programmer's thought process. Comments are not typically processed by tools or generated into separate documents.

```
// this is a single-line comment

/*
this is a multi-line comment
*/
```

Documentation

Documentation is created for a broader audience, including developers, users, and other stakeholders. It serves as a comprehensive guide or reference for a software project, describing its purpose, features, usage, and implementation details.

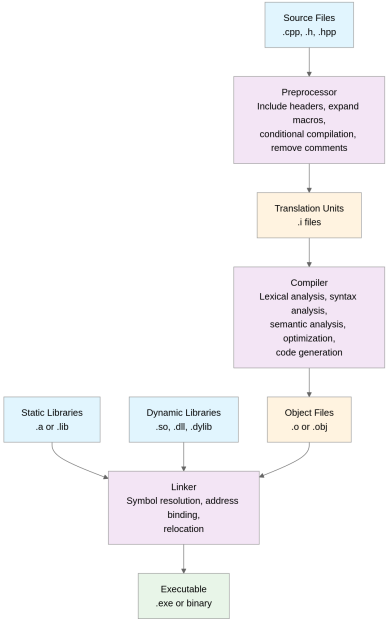
- **Doxygen** is the *de facto* tool to create and generate documentation (HTML, LaTeX, man pages, and RTF) for C++ code (see **examples**).
- To document and generate documentation:
  1. Install the vs code Doxygen extension.
  2. Use Doxygen **commands** to document your programs.
  3. Doxygen tools:
     - `sudo apt install doxygen doxygen-gui`
  4. Create/Edit Doxyfile with `doxywizard`
  5. Configure the settings.
  6. Save the settings: **File** → **Save**
  7. Generate documentation:
     - `cd <folder-of-Doxyfile>`
     - `doxygen` or `doxygen <Doxyfile-name>`

Programming Guideline

When working as a professional programmer, you will likely be required to conform to your employer's coding conventions.

- In this course we use the CPP Core Guidelines.
  - As we progress and learn new concepts, you should refer to this website and follow the guidelines.

One way to lose points on assignments is by failing to follow this guideline.


Source Files
.cpp, .h, .hpp
Preprocessor
Include headers, expand macros,
conditional compilation,
remove comments
Translation Units
.i files
Compiler
Lexical analysis, syntax analysis,
semantic analysis,
optimization,
code generation
Static Libraries
.a or .lib
Dynamic Libraries
.so, .dll, .dylib
Object Files
.o or .obj
Linker
Symbol resolution, address binding,
relocation
Executable
.exe or binary

Preprocessor

- The primary role of the preprocessor is to process directives and generate modified source code that is then passed to the compiler.
  - The preprocessor uses other tools to replace comments with white spaces.
  - The preprocessor uses other tools to add/remove white spaces to reformat the code.
  - Preprocessor directives (start with `#`) are handled by the preprocessor.
    - For instance, the preprocessor recursively does a blind copy-paste whenever `#include` is used.
    - Other directives you may find in C++: `#define`, `#undef`, `#ifdef`, `#endif`, etc
- `g++ -std=c++17 -E lecture2.cpp` shows the intermediate code generated by the preprocessor, before it is passed to the compiler.

Compiler

The compiler takes the preprocessed code and generates object code.

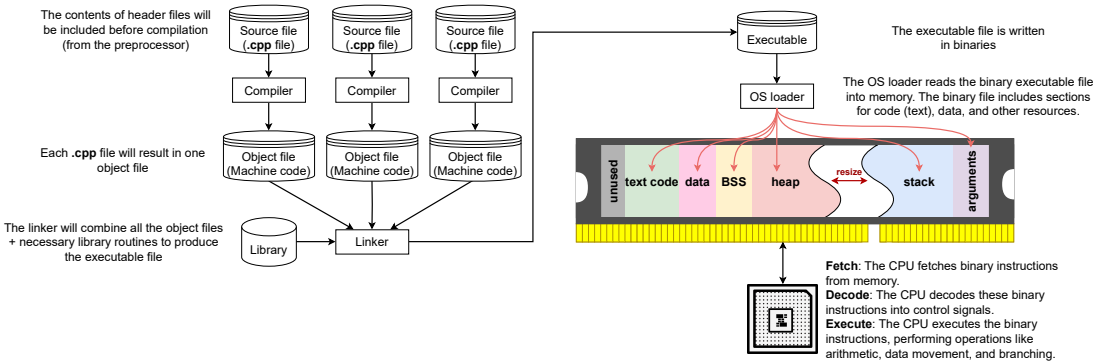
- Each object code contains data and symbols for a specific part of the program.
  - The command `g++ -std=c++17 -c lecture2.cpp` generates only the object file **lecture2.o** and no executable.
  - The command `objdump -D lecture2.o` disassembles the object file.

Linker

The linker plays a crucial role in transforming individual object files and libraries into a complete, executable program or library. It resolves dependencies between different parts of the program, calculates memory addresses, links with external libraries, and prepares the final output for execution or further use.

Memory Model

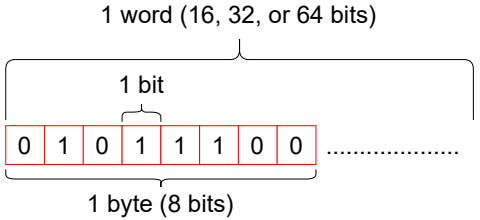
Memory Model



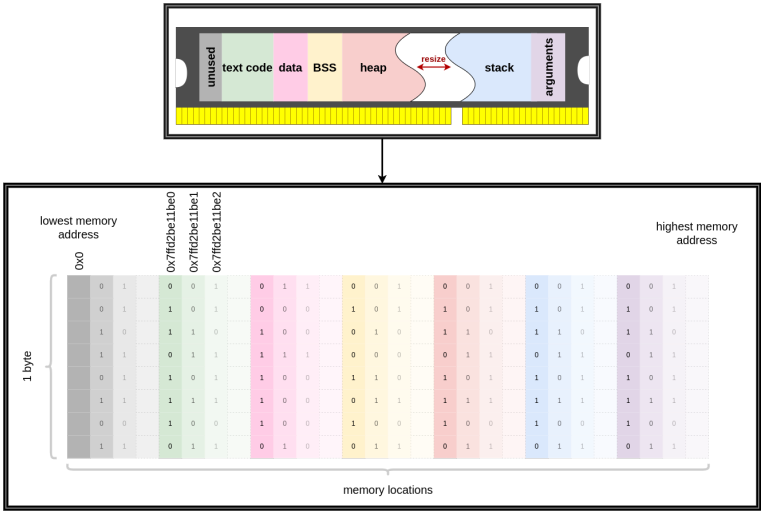
Bits, Bytes, and Words

In computer science, understanding the basic units of data storage and processing is essential. These units are the bit, the byte, and the word.

- **Bit** – A bit (binary digit) is the most basic unit of data in computing. It can have a value of either 0 or 1. Bits are used to represent and store data in binary form.
- **Byte** – A byte is a group of 8 bits. It is the basic addressable element in many computer architectures.
- **Word** – A word is a fixed-sized unit of data that is handled as a single entity by the processor. Words are used for processing and memory access. The size of a word depends on the computer architecture (commonly 16, 32, or 64 bits).



Memory Model



Memory Segments

- In computer memory, the address 0x0 (or simply 0) typically represents the lowest addressable memory location in the system. This address points to the very beginning of the memory, and it is often used as a sentinel or a null reference in programming.
- **text code segment** – Instructions to execute (from the executable) are stored in this segment.
- **data segment** – Initialized global and static variables are stored in this segment.
- **BSS segment (Block Started by Symbol)** – Uninitialized global and static variables are stored in this segment.
- **heap segment** – Region of memory used for dynamic memory allocation.
- **stack segment** – A region of memory used for managing the execution of functions or methods in a program.
- **arguments segment** – If you pass arguments to the `main()` function, they will be stored in this segment.

Challenges and Issues with Memory Management

Correctly using memory is one of the trickiest parts of working with C++. Allocating too much memory in a computer program can lead to various issues and undesirable consequences.

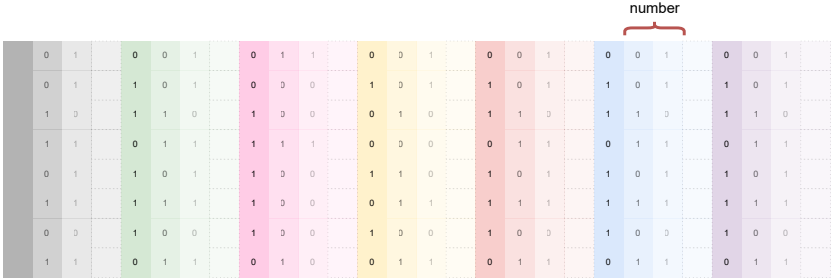
- **Out of Memory Error** – When a program requests more memory than is available in the system's physical RAM and swap space (virtual memory), it may trigger an **Out of Memory** error. This error typically results in the program being terminated, as the operating system cannot fulfill the memory request.
- **Performance Degradation** – Even if the system doesn't run out of memory, allocating excessive memory can lead to severe performance problems. When the system starts using swap space (paging data in and out of disk storage), it can significantly slow down program execution.
- **Fragmentation** – Repeatedly allocating and deallocating large amounts of memory can lead to memory fragmentation, where free memory is divided into small, non-contiguous blocks. Fragmentation can make it challenging to allocate large contiguous memory blocks, which may be necessary for certain operations.
- **Crashes and Instability** – If your program exceeds the system's physical memory and swap space, it can lead to program crashes or system instability. This can affect not only your program but also other running applications.

Variables

Variables

A variable (e.g., `number`) is a symbolic name for a storage location that holds data. Variables are used to hold information that can be referenced and manipulated within a program.

Variables



Characteristics

- **Name (identifier)** – Each variable in the same scope has a unique name (also called an identifier).
- **Type** – Variables have a data type that determines what kind of values they can store.
- **Scope** – The region of code where a variable can be accessed is known as its scope. Variables can have local, global, or other types of scope based on where they are declared.
- **Lifetime** – The duration for which a variable exists in memory during the program's execution is known as its lifetime.
- **Value** – The actual data stored in a variable.

Name (identifier)

- Identifiers can only contain letters, numbers, and underscores,
- Identifiers must begin with a letter or underscore.
- Identifiers are case sensitive.
- Identifiers can not be a reserved keyword.

```
int break1;      // OK
int break_1;     // OK
int Break1;      // OK
int BREAK;       // OK
int _break1;     // OK
int 1Break;      // Error: expected unqualified-id before numeric constant
```

- NL.8: Use a consistent naming style
- NL.10: Prefer underscore_style names
- NL.19: Avoid names that are easily misread

In this course we use the lowercase and underscore convention (snake_case), e.g., `my_variable`, `number`, `average_number`.

Example

```cpp
#include <iostream>

int main(){
    int number = 20;
}
```

Characteristics

- Name (identifier) – `number`
- Type – `int`
- Value – `20`

Variable Types

Variable types in C++ are mainly divided into three types.

1. **Primitive data types** – Built-in C++ data types which can be used directly by the user to declare variables (see next lecture).
2. **Standard Library types** – Type from the Standard Library which can be used by importing the correct header (with `#include`).
3. **User defined data types** – Data types defined by the user (e.g., C++ classes and structures).

The type of the variable is used by the compiler to:

1. Know how much memory is needed to store the variable.
2. Know what kind of value you want to store in the variable.

The `sizeof` Operator

The compiler uses the `sizeof` operator to determine the size (in bytes) of a type or variable.

- `sizeof(type)`: returns size in bytes of the object representation of type.
- `sizeof(expression)`: Returns size in bytes of the object representation of the type that would be returned by expression, if evaluated.
- The size of a type is dependent of the platform. For instance, the size of an `int` is 4 bytes on my machine but it may be different on your computer. Try running the code above to see what outputs you get.

```cpp
int number = 20;
std::cout << sizeof(number) << '\n'; // 4 bytes on my machine
std::cout << sizeof(int) << '\n'; // 4 bytes on my machine
```

Memory Allocation

What exactly happens when the program below executes?

```cpp
#include <iostream>

int main(){
  int number = 20;
  std::cout << number << '\n';
}
```

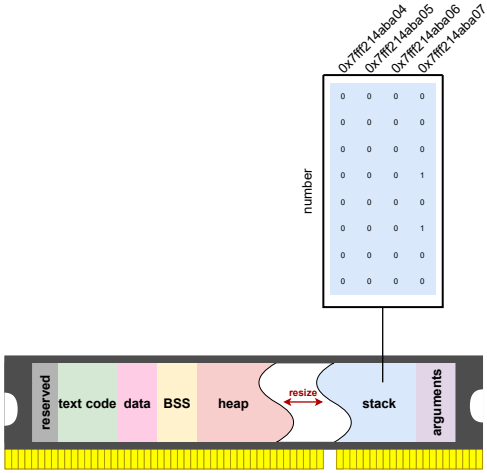
Memory Allocation

```
int number = 20;
```

- The CPU will: **1**) Reserve 4 bytes of memory (**int** is 4 bytes on my machine), **2**) Associate the address of the first byte of those 4 bytes to the variable **number**, **3**) Write **20** in binary in those 4 bytes, and **4**) Restrict the type of data that we are going to store in those 4 bytes to only **int**.
- In C++ you can get the memory address of a variable using the **&** operator.

```
std::cout << &number << '\n';
```

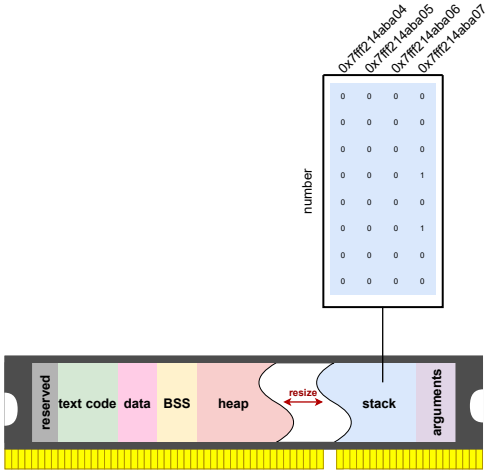
- When I ran the program on my machine, the result was 0x7fff214aba04. I may get a different result if I run the same program now.
- **For now, be aware that a variable declared within a function resides in the stack segment.**



Memory Allocation

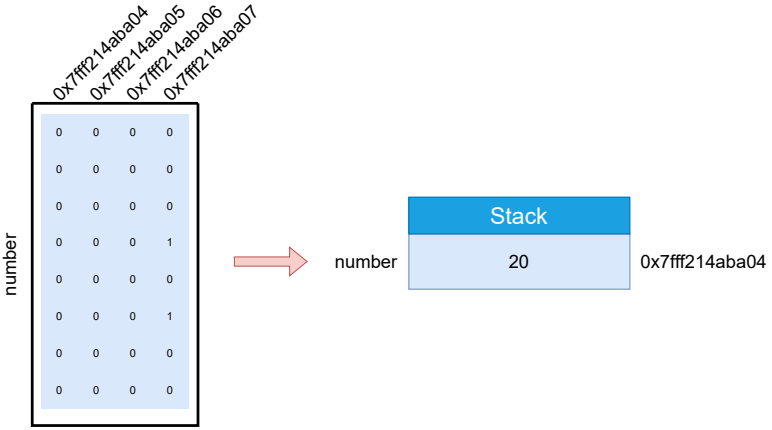
```
std::cout << number << '\n';
```

- The CPU will:
  1. Go to the address of the variable **number** (0x7fff214aba04 in this example)
  2. Grab the binary content from 0x7fff214aba04 to 0x7fff214aba07 (since **number** is **int** and **int** is 4 bytes on my machine)
  3. Convert this binary into decimal.
  4. Print the decimal value.



Representation

From now on in this course, whenever we look at what is happening in memory, we will use a simplified representation.



Variable Declarations

A variable **declaration** is a statement that specifies a variable's data **type** and **name**. Declaring a variable reserves memory space for storing data and allows the compiler to perform type-checking on operations involving that variable.

```
int number; // number is declared
```

- If declaring more than one variable of the same type, they can all be declared in a single statement.

```
int number1, number2, number3; // Yuck !!
```

ES.10: Declare one name (only) per declaration

Variable Assignments

After a variable has been declared, you can give it a value using the assignment operator (=). This is called **copy assignment** (or just **assignment**).

```cpp
int number; // declaration
number = 1; // assignment
std::cout << number << '\n'; // 1
number = 2; // assignment
std::cout << number << '\n'; // 2
```

When we assigned value 2 to the variable `number`, the value 1 that was there previously is overwritten. Normal variables can only hold one value at a time. This is different from Python!

Variable Initializations

Variable assignments require 2 steps: 1) Declare the variable and 2) Assign a value to the variable.

```
int number; // declaration
number = 1; // assignment
```

These two steps can be combined using **initialization**, which consists of declaring a variable and assigning it a value at the same time. The value used to initialize a variable is called an **initializer**.

```
int number = 1;    // 1 is the initializer
std::cout << number << '\n'; // 1
number = 2;        // assignment
std::cout << number << '\n'; // 2
```

ES.20: Always initialize an object

Variable Initializations

There are different ways to initialize a variable.

- **Copy initialization** – This way of initializing variables was inherited from the C language. Copy initialization is not used much in modern C++.

```
int a = 1;  // copy initialization
```

- **Direct initialization** – Direct initialization was initially introduced to allow for more efficient initialization of complex objects (those with class types). However, like copy initialization, direct initialization is not used much in modern C++.

```
int a(1);   // direct initialization
```

- **Uniform initialization** – The modern way to initialize objects in C++ is to use a form of initialization that makes use of braces.

```
int a{1};   // uniform initialization
```

  - Prior to the introduction of uniform initialization, some types of initialization required using copy initialization, and other types of initialization required using direct initialization. Uniform initialization was introduced to provide a more consistent initialization syntax for all features (vectors, arrays, class attributes, etc), hence the name.


In this course we use uniform initialization.

Zero Initialization

In most cases, when a variable is initialized with empty braces, the variable is initialized to zero (or empty, if that is more appropriate for a given type). In such cases where zeroing occurs, this is called **zero initialization**.

```
int a{};                 // initialized to 0
std::cout << a << '\n'; // 0
double b{};              // initialized to 0.0
std::cout << b << '\n'; // 0
std::cout << std::fixed << std::setprecision(1) <<  b << '\n'; // 0.0
```

- By default `std::cout` omits the 0 after the decimal point.
- `std::fixed` is a manipulator in C++ that ensures floating-point numbers are output in fixed-point notation, meaning the number of digits after the decimal point is constant.
- When used in conjunction with `std::setprecision`, it controls the number of digits displayed after the decimal point. You need the header `iomanip`

Zero Initialization

When to use {} and when to use {0}?

- Use {0} when you are actually using that value.
- Use {} if the value is temporary and will be replaced.

```
int a{};    // the value of a will be replaced later
int b{0};   // we plan to use the value of b
a = b + 3;  // value of b is used and a is assigned a new value
```

Uninitialized Variables

When the variable **number** is assigned a memory location by the compiler, the default value of that variable is the value (garbage) that already exists in that memory location!

```
int number;  // uninitialized
std::cout << number << '\n';   // garbage
```

Using the value from an uninitialized variable is one example of Undefined Behavior (UB)

Undefined Behavior

Undefined Behavior

Undefined behavior (UB) in programming refers to the outcome of executing code that breaks the formal language rules. When a program encounters undefined behavior, there are no guarantees about what will happen; the program could produce correct results, incorrect results, crash, or even behave erratically.

Examples

- **Accessing Array Out-of-Bounds** – Trying to access elements that are out-of-bounds of an array.
- **Null Pointer Dereference** – Dereferencing a null pointer.
- **Integer Overflow** – For `signed` integers, overflow is undefined.
- **Reading Uninitialized Variables** – Using the value of a local variable that hasn't been initialized.
- ...

Compiler

The compiler will usually notify about uninitialized variables but an uninitialized variable is just **one example** of UB. The compiler will not always catch situations where UB can happen. Use **-Wall** to decrease chances of UB (modify CMakeLists.txt).

- If you want to ensure that this flag is applied to all targets in your project, you can add the line near the top of your CMakeLists.txt.

```
project(lecture2)

# Enable all warnings
add_compile_options(-Wall)

add_executable(lecture2 src/lecture2.cpp)
```

- If you want to apply **-Wall** only to specific targets:

```
add_executable(lecture2 src/lecture2.cpp)
target_compile_options(lecture2 PRIVATE -Wall)
```

Integral Types

Integral Types

Integral types in C++ are data types that represent whole numbers without fractional or decimal components.

They include `int`, `char`, `short`, `long`, `long long`, `unsigned int`, `unsigned char`, `unsigned short`, `unsigned long`, and `unsigned long long`.

Each type has its own size and range, depending on the underlying architecture and compiler implementation.

Signedness and Size Modifiers

- A **signedness modifier** is used to specify positive and negative numbers.
  - Integral types can be either `signed` or `unsigned`. `signed` types can represent both positive and negative numbers, while `unsigned` types represent only non-negative values.
- **Size modifiers** are used with integral types to specify the size of the integer. They affect the number of bits used to represent the integer, which in turn affects the range of values that can be stored in the variable.
  - `short` is used for values of at least 16 bits, `long` is used for values of at least 32 bits, and `long long` is used for values of at least 64 bits.

Type, Size, and Range

Type	Typical Size (byte)	Typical Range
[signed] char	1	-128 to 127
[unsigned] char	1	0 to 255
[signed] int	4	-2,147,483,648 to 2,147,483,647
unsigned [int]	4	0 to 4,294,967,295
[signed] short [int]	2	-32,768 to 32,767
unsigned short [int]	2	0 to 65,535
[signed] long [int]	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long [int]	8	0 to 18,446,744,073,709,551,615
[signed] long long [int]	8	-9,223,372,036,854,775,808 to -9,223,372,036,854,775,807
unsigned long long [int]	8	0 to 18,446,744,073,709,551,615

Modifiers between square brackets can be omitted.

- e.g., **signed long int**, **signed long**, **long int**, and **long** are all the same.
- e.g., **signed int** is the same as **int**.
- e.g., **unsigned int** is the same as **unsigned**.

Floating-point Number Types

Floating-point Number Types

Floating-point number types in C++ are data types that represent real numbers with a fractional part. They include `float`, `double`, and `long double`. Each type has its own precision and range, depending on the underlying architecture and compiler implementation.

Precision and Range

The precision of a floating-point type refers to the number of significant digits it can represent. `float` typically provides single-precision floating-point numbers with about 7 decimal digits of precision. `double` offers double-precision floating-point numbers with about 15 decimal digits of precision. `long double` provides extended precision, often with greater precision than `double`.

Signedness

Floating-point data types are always signed (can hold positive and negative values).

Type	Typical Size (byte)	Range	Precision
`float`	4	$\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$	6-9 significant digits, typically 7
`double`	8	$\pm 2.23 \times 10^{-308}$ to $\pm 1.80 \times 10^{308}$	15-18 significant digits, typically 16
`long double`	8, 12, or 16	$\pm 3.36 \times 10^{-4932}$ to $\pm 1.18 \times 10^{4932}$	33-36 significant digits

When printing **`float`** numbers we must add the suffix **`f`** or **`F`** at the end of a value. This is because the compiler interprets decimal values without the suffix as **`double`**.

```cpp
std::cout << 1.05 << '\n';  // this is a double
std::cout << 1.05f << '\n'; // this is a float
std::cout << 1f << '\n';    // error
std::cout << 1.0f << '\n';  // OK
#include <iostream>
#include <iomanip> // for output manipulator std::setprecision()

int main(){
    std::cout << std::setprecision(9);      // show 9 digits of precision
    std::cout << 0.33333333333f << '\n';    // 0.333333343
    std::cout << std::setprecision(15) << '\n'; // show 15 digits of precision
    std::cout << 8.3642345343222323232322 << '\n'; // 8.36423435343223 (15 digits)
}
```

Boolean Type

Boolean Type

The Boolean type `bool` in C++ is 1 byte and takes one of the two values `true` or `false`. Usually, 1 (`true`) and 0 (`false`) are assigned to Boolean variables as their default numerical values. Although any numerical value can be assigned to a Boolean variable in C++, all values other than 0 are considered to be `true` and stored as 1 while 0 is considered to be `false`.

Example

```
bool is_today_sunny{true};
bool is_today_cloudy{false};
std::cout << is_today_sunny << '\n';  // 1
std::cout << is_today_cloudy << '\n'; // 0
```

To output **true** or **false** instead of 1 or 0 you can use **std::boolalpha**.
std::noboolalpha will display 1 or 0.

```
bool is_today_sunny{true};
bool is_today_cloudy{false};
std::cout << std::boolalpha << is_today_sunny << '\n';  // true
std::cout << std::boolalpha << true << '\n';            // true
std::cout << std::noboolalpha << true << '\n';          // 1
std::cout << std::boolalpha << is_today_cloudy << '\n'; // false
std::cout << std::boolalpha << false << '\n';           // false
std::cout << std::noboolalpha << false << '\n';         // 0
```

Literals

📜 Literals

A literal (or literal constant) is a notation for representing a fixed value in source code. Literals are used to assign values to variables or constants, and they represent the most basic types of values that can be used in programming. To write numeric literals you can use an **integer-suffix** or a **floating-point suffix**.

Example

- Integral literals.

```
int dec = 12;    // decimal
```

- Floating-point literals.

```
float pi = 3.14159;
double e = 2.71;
float exp = 1.23e4;
```

- Character literals.

```
char a = 'a';
char newline = '\n';
```

- String literals.

```
std::string hello = "Hello";
```

- Boolean literals.

```
bool a = true;
bool b = false;
```

- Literals with `cout`

```
std::cout << 1 << '\n';
std::cout << 1.5f << '\n';
std::cout << true << '\n';
```

Expressions

Expressions

An expression is a combination of literals, variables, operators, and function calls that can be executed to produce a **single value**. The process of executing an expression is called **evaluation**, and the single value produced is called the **result** of the expression.

Example

```
int main(){
    int a{1};            // initialize variable a with literal value 1
    int b{2 + 3 - 1};    // initialize variable b with computed value 4
    int c{2 * 2 + 1};    // initialize variable c with computed value 5
    int d{b};            // initialize variable d with variable value 4
}
```

Type Conversion

📜 Type Conversion

The process of converting a value from one data type to another data type is called a type conversion. Type conversion can be invoked either implicitly (as needed by the compiler) or explicitly (as needed by the user).

```
float f{1}; // initialize float variable with int 1
```

The compiler will determine whether it can convert the value from the current type (`int`) to the desired type (`float`). If a valid conversion can be found, then the compiler will produce a new value of the desired type. If the compiler can not find an acceptable conversion, then the compilation will fail with an error.

Implicit Type conversion

Implicit type conversion is performed automatically by the compiler when one data type is required, but a different data type is supplied.

Type conversions do not change the value or type of the value or object being converted.

```cpp
#include <typeinfo> // needed for typeid
#include <iostream>

int main(){
    double num1 = 1.5;
    int num2 = num1; // 1.5 converted to 1
    std::cout << "Value of num1 : " << num1 << '\n'; // 1.5
    std::cout << "Type of num1 : " << typeid(num1).name() << '\n'; // double
    std::cout << "Value of num2 : " << num2 << '\n';
}
```

`typeid()`

The output from `typeid(num1).name()` is hard to understand.

- When C++ code is compiled, the compiler generates **mangled** names for functions, variables, and other symbols. These mangled names are often unreadable and can be difficult to interpret in debugging or error messages.
- The `c++filt` utility is used to convert these mangled names back into their original human-readable form, which is known as **demangling**.

ToDo

Demangling can be performed by adding `c++filt -t` to the execution of your program:

```
./lecture2 | c++filt -t
```

Implicit Type conversion

Implicit type conversion happens in the following cases.

- When initializing (or assigning a value to) a variable with a value of a different data type:

```
double d{1}; // 1 implicitly converted to type double
d = 3;       // 3 implicitly converted to type double
```

- When the type of a return value is different from the function's declared return type:

```
double my_function(){
    return 1; // 1 implicitly converted to type double
}
```

- When using binary operators:

```
double d{1/3.0}; // 1 implicitly converted to type double
```

- When using a non-Boolean value in an if-statement:

```
if (3){ // 3 implicitly converted to type bool
}
```

- When an argument passed to a function is a different type than the function parameter:

```
void my_function(double x){}
my_function(2); // 2 implicitly converted to type double
```

The Standard Conversions

The C++ standard defines how different fundamental types can be converted to other types. These conversion rules are called the **standard conversions**.

The standard conversions can be divided into 4 categories, each covering different types of conversions:

- Numeric promotions.
- Numeric conversions.
- Arithmetic conversions.
- Other conversions (which includes various pointer and reference conversions).

Numeric Promotion

Numeric promotion is the conversion of a **smaller numeric type** to a larger numeric type. Promotions occur within the same family (integral with integral and floating-points with floating-points).

Smaller numeric type refers to a data type that occupies less memory or has a smaller range of values compared to another data type within the same family (either integral types or floating-point types).

Floating-point Promotion – §7.6

`float` can be promoted to `double`.

```
double num1{5.0};    // no promotion necessary
double num2{4.0f};   // float -> double
```

Integral Promotion – §7.7

- `char` can be promoted to `int`.
- `short` can be promoted to `int`.
- `bool` can be promoted to `int`, with `false` becoming 0 and `true` becoming 1.

```
short s{1};
int num1 = s; // short -> int
int num2 = 'a'; // char -> int
int num3 = true; // bool -> int
std::cout << num1 << '\n';      // 1
std::cout << num2 << '\n';      // 97
std::cout << num3 << '\n';      // 1
std::cout << sizeof(s) << '\n'; // 2
```

Numeric Conversion

Numeric conversions cover additional type conversions not covered by numeric promotions.

- **Integral conversions (excluding integral promotions)** – §7.8

```
short s = 1;            // int -> short
long l = 1;             // int -> long
char c = s;             // short -> char
bool b = 3;             // int -> bool
```

- **Floating-point conversions (excluding floating-point promotion)** – §7.9

```
float f = 3.0;          // double -> float
long double ld = 3.0;   // double -> long double - This is not a promotion!!
```

- **Floating-integral conversions** – §7.10

```
int i = 5.8;            // double -> int
int j = 3.453f;         // float -> int
bool b = 3.0;           // double -> bool
```

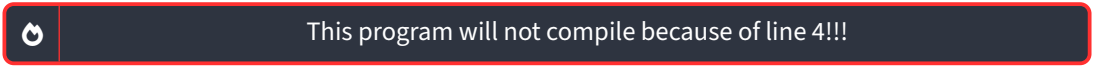
- **Integral-floating conversions** – §7

```
double d = 5; // int -> double - This is not a promotion!!
```

Exercise #1

Predict the outputs of the following program.

```
1 int a;
2 int b = 3.2;
3 int c (1.3);
4 int d {3.5};
5 std::cout << a << '\n';
6 std::cout << b << '\n';
7 std::cout << c << '\n';
8 std::cout << d << '\n';
```




```
1 int a;        // garbage
2 int b = 3.2; // b = 3: Implicit conversion double to int
3 int c (1.3); // c = 1: Implicit conversion double to int
4 int d {3.5}; // Error: narrowing conversion of '3.5e+0' from 'double' to 'int'
```

- The last three conversions perform a **narrowing conversion**, which is a conversion of a value where the loss of its precision may happen.
- Even though the compiler knows how to convert a **`double`** value to an **`int`** value, such conversions are disallowed **only when using uniform initialization** (line 4).

Unlike a numeric promotion (which is always safe), a numeric conversion may (or may not) result in the loss of data or precision.

```
int x = 3.5;            // 0.5 is dropped: data loss
std::cout << x << '\n'; // 3
int x = 3.0;            // no data loss
std::cout << x << '\n'; // 3
```

Why is uniform initialization the only one disallowing narrowing?

- C++ developers believe that implicit narrowing should not exist and disallowed it in the release of uniform initialization in C++11.
- Maybe you made a mistake? Did you really intend to do narrowing conversion?
- This is one of the reasons why uniform initialization is the preferred method to initialize variables in modern C++.

If you really want to do a narrowing conversion, you should make the implicit narrowing conversion explicit by using `static_cast`.

```
int b = static_cast<int> (3.2);
int c(static_cast<int> (1.3));
int d{static_cast<int> (3.5)};
```

Arithmetic Conversions

In C++, certain operators require that their operands be of the same type.

- The binary arithmetic operators: `+`, `-`, `*`, `/`, `%`
- The binary relational operators: `<`, `>`, `<=`, `>=`, `==`, `!=`
- The binary bitwise arithmetic operators: `&`, `^`, `|`
- The conditional operator `?:` (excluding the condition, which is expected to be of type bool).

If one of these operators is invoked with operands of different types, one or both of the operands will be implicitly converted to matching types using a set of rules called the **usual arithmetic conversions** (§8, ¶11)

Usual Arithmetic Conversion

For usual arithmetic conversions, the compiler has a prioritized list of types.

1. `long double` (highest)
2. `double`
3. `float`
4. `unsigned long long`
5. `long long`
6. `unsigned long`
7. `long`
8. `unsigned int`
9. `int` (lowest)

Rules

Two rules are used for usual arithmetic conversions.

Rule #1

If one of the operands is of a type on the priority list, the operand with the lower priority is promoted or converted to the type of the operand with the higher priority.

Rule #2

If neither operand is of a type on the priority list, both operands are numerically promoted.

Example: Rule #1

```cpp
#include <iostream>
#include <typeinfo> // needed for typeid

int main(){
    int i = 42;
    double d = 3.14;
    std::cout << "Type of result: " << typeid(i + d).name() << '\n';  // double
    std::cout << "Value of result: " << i + d << '\n';                // 45.14

    unsigned int ui = 100;
    long l = 5000;
    std::cout << "Type of result: " << typeid(ui + l).name() << '\n';  // long
    std::cout << "Value of result: " << ui + l << '\n';                // 5100

    unsigned short us = 10;
    unsigned long ul = 700000;
    std::cout << "Type of result: " << typeid(us + ul).name() << '\n';  // unsigned long
    std::cout << "Value of result: " << us + ul << '\n';                // 700010
}
```

Example: Rule #2

```cpp
#include <iostream>
#include <typeinfo> // needed for typeid

int main(){
    short s1 = 100;
    char c = 50;
    std::cout << "Type of result: " << typeid(s1 + c).name() << '\n'; // int
    std::cout << "Value of result: " << s1 + c << '\n'; // 150

    unsigned char uc = 200;
    bool b1 = true;
    std::cout << "Type of result: " << typeid(uc + b1).name() << '\n'; // int
    std::cout << "Value of result: " << uc + b1 << '\n'; // 201

    bool b2 = false;
    short s2 = 32767;
    std::cout << "Type of result: " << typeid(b2 + s2).name() << '\n'; // int
    std::cout << "Value of result: " << b2 + s2 << '\n'; // 32767
}
```

Exercise #2

Predict the outputs of the following program.

```cpp
#include <iostream>
#include <typeinfo>  // needed for typeid

int main() {
    int a{3};
    int b{2};
    std::cout << "Type of result: " << typeid(a / b).name() << '\n';
    std::cout << "Value of result: " << a / b << '\n';
}
```

Exercise #3

Modify the code so that the type is **int** and the result is **1.5**.

Do not modify lines 5 and 6.

```
1 #include <iostream>
2 #include <typeinfo>  // needed for typeid
3
4 int main() {
5     int a{3};
6     int b{2};
7     std::cout << "Type of result: " << typeid(a / b).name() << '\n';
8     std::cout << "Value of result: " << a / b << '\n';
9 }
```

Constants

Constants

A constant is an expression with a fixed value.

In programming there are many cases where it is useful to define variables with values that can not be changed (e.g., number of days/week=7, $\pi = 1.141598$, speed of light=$186,282\ mi/s$, etc).

> In C++, constants are initialized when they are created and new values can not be assigned to them.

Constants

C++ has 3 types of constants:

- Literal constants (already covered in slide 56).
- Constant variables.
- Symbolic constants.

Constant Variables

A variable whose value can not be changed is called a **constant variable**.

- NL.26: Use conventional const notation

```
const double pi{3.141598};  // "west const" style is preferred
double const pi{3.141598};  // "east const" style no preferred
```

- Constant variables must be initialized.

```
const double pi;            // Error: uninitialized const 'pi'
```

- Once defined, a constant variable can not be assigned a new value.

```
const double pi{3.141598};  // initialization
pi = 3.14;                  // Error: assignment of read-only variable 'pi'
```

Symbolic Constant

Symbolic constants are created with macros. A macro is a piece of code in a program that is processed by the preprocessor.

```cpp
#include <iostream>
#define PI 3.14159  // symbolic constant

int main() {
    std::cout << "pi: " << PI << '\n'; // preprocessor replaces PI with 3.14159
}
```

In modern C++, always use constant variables instead of macros.

- **Macros Lack Type Checking** – Since macros are processed by the preprocessor and replaced with their corresponding values before the compilation stage, the compiler does not perform any type checking on them.

```
#define SQUARE(x) ((x) * (x))

int a = 5;
double result = SQUARE(a); // No type checking, 'a' can be of any type
```

- **Macros Make Debugging Difficult** – Because macros are replaced by the preprocessor with their literal values before compilation, the original macro name does not appear in the compiled code. This makes it difficult to trace errors or understand the code during debugging since the debugger will only show the expanded value, not the macro name.

```
#define PI 3.14159

double area = PI * 10 * 10;
// During debugging, you'll see 'area' as 3.14159 * 10 * 10,
// not as PI * 10 * 10, making it harder to connect the value to the original macro
```

- Other issues.

Constant Expressions

A **constant expression** is an expression that can be evaluated by the compiler at compile-time. To be a constant expression, all the values in the expression must be known at compile-time.

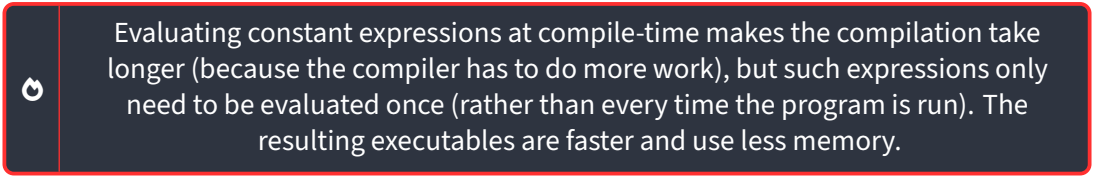
- When the compiler encounters a constant expression, it will replace the constant expression with the result of evaluating the constant expression.

```
int main(){
    std::cout << 1 + 2 << '\n';
}
```

- Each time the program is run, the output will always be 3.
- In reality, 1 + 2 is not evaluated at run-time but at compile-time. In other words, the compiler will optimize this code and will compile the following code.

```
int main(){
    std::cout << 3 << '\n';
}
```

- Try it at https://godbolt.org/ (assembly code generated by the compiler).


Evaluating constant expressions at compile-time makes the compilation take longer (because the compiler has to do more work), but such expressions only need to be evaluated once (rather than every time the program is run). The resulting executables are faster and use less memory.

Compile-time Constants

A **compile-time constant** is a constant whose value is known at compile-time.

- Constant variables may or may not be compile-time constants.

```
const int a{1};              // a is a compile-time const
const int b{2};              // b is a compile-time const
std::cout << a + b << '\n'; // a + b is a compile-time expression
```

- Try it at https://godbolt.org/

```
std::cout << "Enter an integer: ";
int input{};
std::cin >> input;
const int a{1};              // a is a compile-time const
const int b{input};          // b is a run-time const
std::cout << a + b << '\n'; // a + b is a run-time expression
```

The constexpr Keyword

Because compile-time constants generally allow for better optimization, we typically want to use compile-time constants wherever possible.

- When using **const**, our constant variables could end up as either a compile-time constant or a runtime constant, depending on whether the initializer is a compile-time expression or not.
- If you want to use **only** compile-time constants and not run-time constants, you can clearly tell the compiler about your intention with the use of **constexpr**.
  - The compiler will error if you are using a run-time constant instead of a compile-time constant.

```
constexpr int a{1};      // OK: a is a compile-time const
constexpr int b{2};      // OK: b is a compile-time const
std::cout << "Enter an integer: ";
int input{};
std::cin >> input;
constexpr int c{input}; // Error
```

Con.5: Use constexpr for values that can be computed at compile time

Resources

- More on constexpr

Performance Considerations

- `const` Performance:
  - **Runtime Overhead** – If the `const` value is determined at runtime, there may be some runtime overhead associated with its calculation or assignment.
  - **Compiler Optimizations** – The compiler may still optimize `const` values, but since they may be set at runtime, these optimizations are not as guaranteed or as extensive as with `constexpr`.
- `constexpr` Performance:
  - **Compile-Time Calculation** – Since `constexpr` values are calculated at compile-time, they incur no runtime cost, which can lead to faster execution, especially in performance-critical code.
  - **Reduced Memory Footprint** – By computing values at compile-time, `constexpr` can reduce the memory footprint, as these values do not need to be stored in runtime variables.
  - **Better Inlining** – Functions declared `constexpr` can be inlined more effectively, further reducing function call overhead.

Type Deduction

Type Deduction

Type deduction is a feature that allows the compiler to deduce the type of an object from the object's initializer. To use type deduction, the `auto` keyword is used in place of the variable's type.

Examples

```
auto a{3.0};     // 3.0 is a double literal, so variable a will be type double
auto b{1 + 2};   // 1 + 2 evaluates to an int, so b will be type int
auto c{b};       // variable b is an int, so c will be type int
```

Type deduction will not work for objects that do not have initializers or empty initializers.

```
auto a;    // Error: declaration of 'auto a' has no initializer
auto b{};  // Error: could not deduce template parameter 'auto'
```

Constant Type Deduction

In most cases, type deduction will drop the **const** qualifier from deduced types.

```
int main(){
    const int a{5}; // a is const
    auto b{a};      // b is int (const is dropped)
    b = 1;          // OK
}
```

If you want a deduced type to be **const**, you can use the **const** or **constexpr** keywords in conjunction with the **auto** keyword:

```
int main(){
    constexpr int a{5};  // a is const int
    constexpr auto b{a}; // b is const int
    b = 1;           // Error: assignment of read-only variable ‘b’
}
```

Compound Statement

A **compound statement** (also called a **block**, or **block statement**) is a group of zero or more statements.

- Blocks begin with a { symbol, end with a } symbol, with the statements to be executed being placed in between.
- Blocks can be used anywhere a single statement is allowed.
- No semicolon is needed at the end of a block.

```
int main(){          // start outer block
    int a{};
    {                // start nested block 1
        int b{};
        {            // start nested block 2
            int c{};
        }            // end nested block 2
    }                // end nested block 1
}                    // end outer block
```

Keep the nesting level of your functions to 3 or less. If your function has a need for more nested levels, consider refactoring your function into sub-functions.

Scopes

Scopes

A variable's scope determines where the variable can be accessed within the source code.

- When an identifier can be accessed, we say it is **in scope**.
- When an identifier can not be accessed, we say it is **out of scope**.

Scope is a compile-time property, and trying to use an identifier when it is not in scope will result in a compile error.

Local Scope

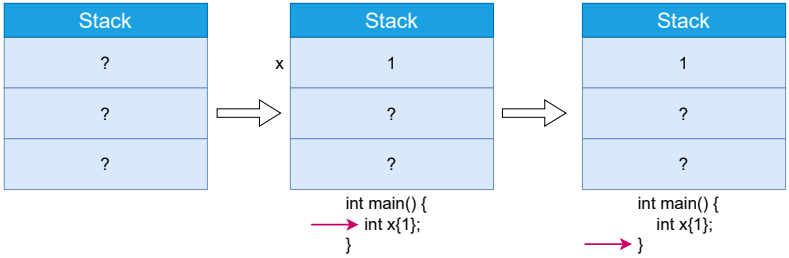
Function parameters, as well as variables defined inside the function body, are called **local variables**. Local variables have a **local scope**, which is delimited by the braces.

- Variables within the function body are created at the point of definition.
- Local variables are destroyed in the opposite order of creation at the end of the set of braces in which it is defined.

```
int main() {
    int a{1};
    {
        int b{2};
        std::cout << a << '\n'; // 1
        std::cout << b << '\n'; // 2
    } // b goes out of scope here.
    std::cout << b << '\n'; // Error: b is out of scope
    int c{3};
} // a and c go out of scope here.
```

What happens when a variable goes out of scope?

The variable's lifetime ends at the point where it "goes out of scope". This means that the variable is not accessible anymore and the memory occupied by the variable is deallocated. However, the variable's previous value or garbage is at the memory location where the variable used to be.



Global Scope

Global variables have **file scope** (also informally called **global scope** or **global namespace scope**), which means they are visible from the point of declaration until the end of the file in which they are declared. By convention, global variables are declared at the top of a file, below the includes, but above any code.

```cpp
#include <iostream>

int global_var{1};

void my_function(){
    global_var++;
}

int main() {
    std::cout << global_var << '\n';     // 1
    global_var++;                        // 2
    my_function();                       // 3
    std::cout << global_var << '\n';     // 3
}
```

Global variables are evil because they can be accessed from anywhere in a program and it can be modified by any function. This can cause unexpected results.

In general, do not use global variables! If you really need to use global variables, make them read-only (`const` or `constexpr`)

R.6: Avoid non-const global variables

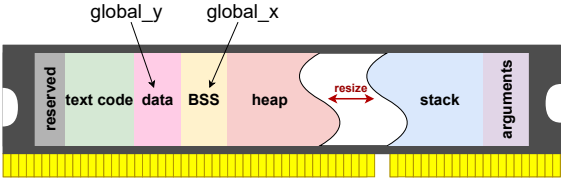
Memory

Initialized global variables are set in the **data** segment while **uninitialized** global variables are set in the **BSS** (Block Started by Symbol) segment.

```cpp
#include <iostream>

int global_x;
int global_y{1};

int main(){
    std::cout << &global_x << '\n';
    std::cout << &global_y << '\n';
}
```



Naming Collisions

Naming Collisions

C++ requires that all identifiers be non-ambiguous. **Naming collisions** (or naming conflicts) occur when two or more identifiers (e.g., variables, functions, classes, etc.) in the same scope have the same name.

```cpp
#include <iostream>

int main() {
    int x{1};
    int x{2};  // Error: redeclaration of 'int x'
    std::cout << x << '\n';
}
```

Namespaces

A **namespace** is a declarative region that provides a scope to the identifiers (names of types, variables, functions, etc.) inside it. Namespaces are used to organize code and to avoid naming collisions in larger projects or when integrating with libraries. The C++ Standard Library itself is inside the `std` namespace.

- The namespace provides a scope region (called **namespace scope**) to the names declared inside of it.
- Any name declared inside the namespace will not be mistaken for identical names in other scopes.
- So far we have used the namespace `std` to use objects from the Standard Library (`std::cin`, `std::cout`, etc).
- To create a custom namespace (let's call it `MyNamespace`), the syntax is as follows:

```
namespace MyNamespace{
    // everything here belongs to the namespace MyNamespace
}
```

Namespaces

There are multiple ways to access objects from a namespace.

- Explicit use of the namespace using the scope resolution operator `::`
- The `using namespace` directive.
- The `using` directive variants.

Explicit use of the Namespace

```cpp
#include <iostream>

// namespace MyNamespace
namespace MyNamespace {
int x{3};
int y{4};
} // end namespace MyNamespace

int main() {
    std::cout << MyNamespace::x << '\n';  // 3
    std::cout << MyNamespace::y << '\n';  // 4
}
```

The `using namespace` Directive

```cpp
#include <iostream>

// namespace MyNamespace
namespace MyNamespace {
int x{3};
int y{4};
} // end namespace MyNamespace


using namespace MyNamespace;

int main() {
    std::cout << x << '\n';  // no need to use MyNamespace::x
    std::cout << y << '\n';  // no need to use MyNamespace::y
}
```

The using Directive Variants

```cpp
#include <iostream>

// namespace MyNamespace
namespace MyNamespace {
int x{3};
int y{4};
}  // end namespace MyNamespace


using MyNamespace::x;

int main() {
    std::cout << x << '\n';  // no need to use MyNamespace::x
    std::cout << y << '\n';  // Error: ‘y’ was not declared in this scope
}
```

Avoid `using namespace` overall.

```cpp
#include <iostream>

// custom namespace
namespace MyNamespace{
int cout{1};
} // end namespace MyNamespace

using namespace std;
using namespace MyNamespace;

int main() {
    cout << cout << '\n'; // Error: reference to ‘cout’ is ambiguous
}
```

Readability

Besides preventing naming collisions, namespaces make your program more readable. In the program below, it is not really clear right away which objects belong to the namespace `MyFirstNamespace`, `MySecondNamespace`, or `std`.

```cpp
#include <iostream>
#include <vector>
#include <array>
#include "outside_file.h" // MyFirstNamespace and MySecondNamespace defined here

using namespace std;
using namespace MyFirstNamespace;
using namespace MySecondNamespace;

int main() {
    array<array<int, 2>, 4> arr{{{1, 2}, {1, 2}}};
    vector<int> vect = {1, 2, 3, 4};
    cout << x << '\n';
    cout << y << '\n';
    cout << z << '\n';
}
```

Aliases

Aliases

In C++, you can create custom types using the `using` keyword. This is useful for creating type aliases, which can make your code more readable and maintainable.

Example

```cpp
#include <iostream>

// Create custom types
using Integer = int;
using Float = float;
using uint = unsigned int;

int main() {
    // Use the custom types
    Integer a{10};
    Float b{20.5f};
    uint age{30};

    std::cout << "Integer: " << a << '\n';
    std::cout << "Float: " << b << '\n';
    std::cout << "Age: " << age << '\n';
}
```

Next Class

- Quiz#1
- Lecture3: Normal Pointers
- Reading Material