

Tutorial on C++

Huy NGUYEN

2/19/23

Table of contents

Preface	6
1 Introduction	7
2 Introduction to C++	8
2.1 Statements	8
2.2 Function	8
2.3 Syntax and syntax error	8
2.4 Comments	8
2.5 Objects and Variables	9
2.6 Data	9
2.6.1 Objects and variables	10
2.6.2 Variable instantiation	10
2.6.3 Data types	10
2.6.4 Define multiple variables	10
2.7 Variable assignment and initialization	11
2.7.1 Variable assignment	11
2.7.2 Initialization	11
2.8 Introduction to iostream: cout, cin, and endl	13
2.8.1 The input and output (io) library	13
2.8.2 std::cout is buffered	14
2.8.3 std::endl vs \n	15
2.8.4 std::cin	15
2.9 Uninitialized variables and undefined behavior	16
2.9.1 Undefined behavior	16
2.10 Basic formatting	17
2.11 Literals and operators	17
2.11.1 Literals	17
2.11.2 Operators	17
2.11.3 Return values and side effects	17
2.12 Expression	18
2.12.1 Expression statement	18
2.12.2 Useless expression statements	18
2.13 Developing first program	18
2.13.1 Multiply by 2	18

3	Functions and Files	20
3.1	Introduction to functions	20
3.1.1	An example of a user-defined function	20
3.1.2	Function return values (value-returning functions)	22
3.1.3	Fixing our challenge program	24
3.1.4	Revisiting main()	25
3.1.5	Reusing functions	25
3.2	Void functions	27
	Void functions can't be used in expression that require a value	27
3.3	Introduction to function parameters and arguments	29
3.3.1	The problem	29
3.3.2	Function parameters and arguments	31
3.4	Introduction to local scope	33
3.4.1	Local variables	33
3.4.2	Local scope	34
3.5	Why functions are useful, and how to use them effectively	35
3.5.1	Why use functions?	35
3.5.2	Effectively using functions	36
3.6	Forward declarations and definitions	36
3.6.1	Reorder the function definitions	37
3.6.2	Use a forward declaration	37
3.6.3	Declaration and definition	38
3.7	Programs with multiple code files	38
3.8	Naming collisions and an introduction to namespaces	39
3.8.1	Namespace	39
3.9	Introduction to the preprocessor	41
3.10	Header files	41
3.11	How to design your first programs	42
3.11.1	Design step 1: Define your goal	43
3.11.2	Design step 2: Define requirements	43
3.11.3	Design step 3: Define your tools, targets, and backup plan	43
3.11.4	Design step 4: Break hard problems down into easy problems	43
3.11.5	Design step 5: Figure out the sequence of events	43
3.11.6	Implementation step 1: Outlining the main function	44
3.11.7	Implementation step 2: Implement each function	44
3.11.8	Implementation step 3: Final testing	45
4	Fundamental Data Types	46
4.1	Introduction to fundamental data types	46
4.1.1	Fundamental data types	46
4.1.2	Void	46
4.1.3	Object sizes and the size of operator	47
4.1.4	Signed integer	48

4.1.5	Floating point numbers	48
4.1.6	Rounding errors makes floating point comparision tricky	49
4.1.7	Boolean values	50
4.2	Introduction to if statement	52
4.3	Chars	52
4.4	An introduction to explicit type conversion via the <code>static_cast</code> operator	52
4.5	Const variables and symbolic constants	54
4.6	Compile-time constants, constant expressions, and <code>constexpr</code>	54
4.6.1	The <code>constexpr</code> keyword	55
4.7	Literals	55
4.8	String	56
4.8.1	String input with <code>std::cin</code>	56
4.8.2	String length	58
4.8.3	Literals for <code>std::string</code>	59
4.9	Introduction to <code>std::string_view</code>	59
5	Debugging C++ Programs	61
5.1	Syntax and semantic errors	61
5.2	The debugging process	62
5.2.1	Debugger	62
5.2.2	The call stack	63
6	Operators	64
6.1	Introduction	64
6.2	Arithmetic operators	66
6.3	Increment/decrements operators, and side effects	66
6.3.1	Side effects can cause undefined behavior	68
6.4	Comma and conditional operators	69
6.4.1	Comma as a separator	69
6.4.2	Conditional operator	70
6.5	Relational operators and floating point comparisons	70
7	Scope, duration and linkage	72
7.1	Compound statements	72
7.2	User-defined namespaces and the scope resolution operator	73
7.2.1	Define our own namespaces	74
7.3	Local variables	78
7.4	Introduction to global variables	79
7.5	Variable shadowing (name hiding)	80
7.6	Internal linkage	80
	The one-definition rule and internal linkage	81
7.7	External linkage and variable forward declarations	81
7.8	Why (non-const) global variables are evil	82

7.9	Sharing global constants across multiple files	82
7.9.1	Global constants as inline variables	83
7.10	Static local variables	85
7.10.1	Static local constants	86
7.11	Using declarations and using directives	86
7.12	Inline function	86
7.13	Constexpr and consteval functions	86
7.14	Chapter summary	87
8	Control flow and Error Handling	88
8.0.1	If statements and blocks	88
8.0.2	Common if statement problem	89
8.0.3	Switch statement basics	91
8.1	Goto statements	94
8.2	Introduction to loops and while statements	95
8.2.1	Nested loop	96
8.3	Do while statement	99
8.4	For statements	99
8.5	Break and continue	101
8.5.1	Continue	102
8.6	Halts	102
8.6.1	The <code>std::exit()</code> function	102
8.6.2	<code>std::abort()</code>	102
8.7	Introduction to testing the code	103
8.7.1	Test your programs in small pieces	103
8.8	Common semantic errors in C++	104
8.9	Detecting and handling errors	104
8.9.1	Handling the error within the function	104
8.9.2	Passing error back to the caller	105
8.9.3	Fatal errors	105
8.10	<code>std::cin</code> and handling invalid input	105
8.11	Assert and <code>static_assert</code>	109
9	Summary	110
	References	111

Preface

This is a Quarto book on learning C++ from the tutorial website: learncpp.com.

To learn more about Quarto books visit <https://quarto.org/docs/books>.

1 Introduction

This is a book created from markdown and executable code.

See Knuth (1984) for additional discussion of literate programming.

1 + 1

2 Introduction to C++

2.1 Statements

Statements are the instructions that cause the program to *perform some actions*. Most of the statements end with *semicolon*.

2.2 Function

In C++, statements are typically grouped into units called **functions**. Function is a collection of statements that execute sequentially (in order, from top to bottom).

Functions are written to do a **specific tasks**.

2.3 Syntax and syntax error

Syntax is the rules how the programs must be constructed in order to be considered valid. If we violate a rule, the compiler will complain and issue a **syntax error**.

Program's compilation will only complete once **all** syntax errors are resolved.

2.4 Comments

A comment is a programmer-readable note inserted directly into a source code of the program (compiler ignores the comments).

Single-line comments: // symbol begins a C++ single-line comment.

Multi-line comments: /* and */ pair of symbols denotes a multi-line comment.

Proper use of comments

- Describe *what* the library, program, or functions, do. These are typically placed on the top of the file or library.

- Second, inside the lib, program, or function, comments are used to describe *how* the code is going to accomplish the goal.
- Third, at the *statement* level, comments can describe *why* the code is doing something. The **bad** statement comment describe *what* the code is doing.

Comments are good way to remind the reason a programmer made one decision instead of another.



Best practice

Comment your code liberally, and write your comments as if speaking to someone who has *no idea* what the code does. Don't assume that you'll remember why you mad specific choices.

You can also *comment out* the code block if:

1. You're working on a new piece of code that won't compile yet.
2. You have written a new code that results the errors and you do not have time to fix it.
3. To find the source of the errors.
4. You want to replace one piece of code with another piece of code.

Summary

- At the library, program, or function level, use comments to describe *what*.
- Inside them, use comments to describe *how*.
- At the statement level, use comments to describe *why*.

2.5 Objects and Variables

2.6 Data

In Section 2.1, statements inside function perform actions that generate whatever result the program was designed to produce.

But, **how do programs actually produce results?**: bref, by *manipulating* data.

In computing, **data** is *any information* that can be moved, processed and stored by computer.



Key insight

Programs are collections of instructions that manipulate data to produce a desired results.

A program can acquire data to work in many ways: file, database, network, user providing input on keyboard, or from the programmer putting data directly into a source code.

Data on computer is typically stored in a efficient format (and thus is not human readable).

2.6.1 Objects and variables

- A single piece of data stored in memory somewhere is called **value**.
- An **object** is a region of storage that has a value and other associated properties. We use the object to store and retrieve values.
- Objects can be named or unnamed. A named object is called **variable**, the name of object is call **identifier**.

2.6.2 Variable instantiation

To create a **variable**, we have to **define** it. Here is an example of defining a variable named x:

```
int x;
```

When the program run, it will instantiate the **variable**: object will be created and assigned a memory address. Variable must be instantiated *before* they can be used to store value.

2.6.3 Data types

Variables are named region of storage that can store value. A **data type** tells the compiler what type of value the variable will be store.

2.6.4 Define multiple variables

We can define muliple variables in one statement by separating them by ,.

```
int a, b;
```

But, variables of different types must be defined in separate statements:

```
int a, double b; # false
int a; double b; # correct

# Correct and recommended
int a;
double b;
```



Best practice

Avoid defining multiple variables of the same type in a single statement. Instead, define each variable in a separate statement on its own line

2.7 Variable assignment and initialization

In previous session Section 2.5, we know how to define a variable. In this session, we'll explore how to actually *put values into variables* and use those values.

Recall the variable's definitions:

```
int x;  
int y;  
int z;
```

2.7.1 Variable assignment

After variable has been defined, you can give it a value by using `=`, **copy assignment** or **assignment**.

```
#include <iostream>  
  
int main() {  
  
    int width;  
    width = 5; // copy assignment of value 5 into variable width  
  
    width = 7;  
  
    return 0;  
}
```

2.7.2 Initialization

With *initialization*, we can define and assign value to variable at the same time. There are many ways to initiate:

```
int a;  
int a = 5; #copy initialization  
int c( 6 ) # direct initialization  
  
# List initialization methods  
int d { 7 };  
int e = { 8 };  
int f {}
```

Default initialization

We do not provide any initialization value.

Copy initialization

With `=`: copy the value on the right-hand side of the equals into the variable being created. It does not use much in modern C++.

Direct initialization

Similar to copy initialization, not popular in modern C++.

List initialization

This is the modern way to initialize. Prior to this, some types of initialization require using the direct one, and other types required indirect one. The *list initialization* introduce to provide more consistent initialization syntax.

List initialization disallows “narrowing conversion”: return errors if we initialize a variable using a value that the variable cannot safely hold:

```
int width { 4.5 }; # return error
```

Best practice

- Favor list initialization whenever possible.
- Initialize your variables upon creation.

2.8 Introduction to iostream: cout, cin, and endl

2.8.1 The input and output (io) library

io library is part of C++ standard library that works with input and output. We use the functionality in iostream to get *input* from keyboard and *output* data to console.

We use the library by including it:

```
#include <iostream>
```

One of the most useful is `std::cout`: send data to the console to be printed as text

```
#include <iostream>

int main()
{
    std::cout << "Hello World!\n";

    return 0;
}
```

It can print a *number* too:

```
#include <iostream>

int main()
{
    std::cout << 4;

    return 0;
}
```

It can print value of a variable:

```
#include <iostream>

int main()
{
    int x{ 5 };
    std::cout << x;
}
```

```
    return 0;
}
```

To print more than one thing in the same line:

```
#include <iostream>

int main()
{
    int x{ 5 };
    std::cout << "x is equal to " << x;

    return 0;
}
```

Use `std::endl` to make the cursor in the next line.

```
#include <iostream>

int main()
{
    int x{ 5 };
    std::cout << "Hi!" << std::endl;
    std::cout << "My name is Huy." << std::endl;

    return 0;
}
```

Best practice

Output a newline whenever a line of output is complete.

2.8.2 `std::cout` is buffered

Statements request that output be sent to the console. However, output is typically not sent to the console immediately. Instead, the requested output *gets in line*, and is stored in a region of memory set aside to collect later, called **buffer**. Periodically, buffer is **flushed**, all data is transferred to its destination.

2.8.3 std::endl vs \n

std::endl move the cursor to the next line and *flushes* the buffer. Sometimes, we do not need to flush the buffer and prefer the system do it periodically.

Thus, use of \n is preferred instead:



Best practice

Prefer \n over std::endl when ouputing text to the console.

2.8.4 std::cin

std::cin reads the data from keyboard using extraction operator >>.

```
#include <iostream>

int main()
{
    std::cout << "Enter a number: ";

    int x{ };
    std::cin >> x;

    std::cout << "You endtered: " << x << '\n';
    return 0;
}
```

We can input many variables in the same line:

```
#include <iostream>

int main()
{
    std::cout << "Enter two numbers separated by a space: ";

    int x{ };
    int y{ };
    std::cin >> x >> y;

    std::cout << "You entered " << x << " and " << y << ".\n";
    std::cout << "Thank you!\n";
}
```

```
    return 0;
}
```

Best practice

Initialize variable first, then use `std::in`.

2.9 Uninitialized variables and undefined behavior

C++ does not initialize the variable automatically. Thus, when a variable is given a memory address to use to store data, the default value of that variable is whatever value happens to already be in that memory address.

Let be clear that:


- Initialization = the object is given a known value at the point of definition.
- Assignment = The object is given a known value beyond the point of definition
- Uninitialized = The object has not been given a known value yet.

In some case, we do not know what the value printed from the following code because uninitialized `x` is given any value.

```
#include <iostream>

int main()
{
    int x;
    std::cout << x << '\n';

    return 0;
}
```

 Using uninitialized is one of the most common mistakes

2.9.1 Undefined behavior

Uninitialized variables are one example of undefined behavior: we cannot know the results, or they can change, be incorrect, sometime correct, or crash.

2.10 Basic formatting

The recommendations for basic formatting:

1. It fine to use either tab or spaces for indentation.
2. Function braces can be:

```
int main() {  
}
```

or

```
int main()  
{  
}
```

The author recommended the later.

3. Each statement should start one tab from the opening space.
4. Line should not be too long. Typically, 80 characters is the maximum length in a line.
5. If line is too long, split with an operator (eg. +, -). The operator should be placed at the beginning of the next line.
6. Use whitespaces to make the code easier to read.

2.11 Literals and operators

2.11.1 Literals

Literal is a fixed value that has been inserted directly into the source code.

2.11.2 Operators

Operator is the specific operation to be performed.

2.11.3 Return values and side effects

An operator that has some observable effect beyond producing return value is said to have a side effect. For example, `x = 5` is evaluated, side effect is to assign 5 to x; the changed value of x is observable, or `std::cout << too`.

2.12 Expression

Expression is a combination of literals, variables, operators, and function calls that calculates a single value.

The process of executing an expression is called **evaluation**. The single value is called **result**.

Expressions do not end in a semicolon, and cannot be compiled by themselves (they must exist within a statement), example that

```
int x{ 2 + 3 } ;
```

`2 + 3` is an expression and do not need a semicolon.

Expressions also involve operators with side effects:

```
x = 5;  
x = 2 + 3;  
std::cout << x;
```

2.12.1 Expression statement

Expression cannot compile by itself. We can add `=` to make it compile. We call this **expression statement**, an expression following by a semicolon.

2.12.2 Useless expression statements

We can make an expression without assigning or having any side effect, such that

```
2 * 3
```

2.13 Developing first program

2.13.1 Multiply by 2



Best practice

New programmers often try to write an entire program all at once, and then get overwhelmed when it produces a lot of errors. A better strategy is to add one piece at a time, make sure it compiles, and test it. Then when you're sure it's working, move on to the next piece.

There are many solutions, from bad to good. Let take time to find the good one (not override input value, avoid complexity).

- We should firstly try to make the program work
- Then, spend time to cleanup the code.

```
#include <iostream>

int main()
{
    std::cout << "Enter an integer: ";

    int num{ };
    std::cin >> num;

    std::cout << "Double " << num << " is: " << num * 2 << ".\n";
    std::cout << "Triple " << num << " is: " << num * 3 << ".\n";

    return 0;
}
```

3 Functions and Files

3.1 Introduction to functions

A **function** is a reusable sequence of statements designed to do a *particular job*. Function allows us to split the program into small, modular chunks which are easier to organize, test, and use. Besides the standard library of C++, we can write our own function, called **user-defined function**.

When a program is executing the statements sequentially inside one function and encounters a function call. A **function call** is an expression that tells CPU to **interrupt** the current function and *execute another function*. The CPU put a *bookmark* at the current point of execution, and then **calls** the function.

The function initiating the function call is the **caller**, the other is **called function**.

3.1.1 An example of a user-defined function

The general syntax of function:

```
#| label: func_syntax
returnType functionName() # This is function header
{
    # This is body of the function
}
```

The first line is **function header** telling the compiler that the function exists.

- In this session, *returnType* of `int` for `main()` and *void* otherwise.
- *functionName* is name of user-defined function.

Let see the example function:

```
#| label: func_doPrint

#include <iostream>      // for std::cout

// Definition of user-defined function doPrint()
```


```

void doPrint()
{
    std::cout << "In doPrint() \n";
}

int main()
{
    std::cout << "Starting main() \n";
    doPrint();
    std::cout << "Ending main() \n";

    return 0;
}

```

 Do not forget to include parentheses () after the function's name when making a function call.

- Function can be called more than **one** time in a *caller*.
- Any functions can call any other functions:

```

#| label: func_call_func

#include <iostream>      // for std::cout

// Definition of user-defined function doPrint()
void doB()
{
    std::cout << "In doB() \n";
}

void doA()
{
    std::cout << "Starting doA()\n";
    doB();
    std::cout << "Ending doA()\n";
}

int main()
{
    std::cout << "Starting main() \n";
    doA();
}

```

```

    std::cout << "Ending main() \n";

    return 0;
}

```

- **Nested functions** are not supported: we cannot declare a function inside another function.

```

#include <iostream>

int main()
{
    void foo() # Illegal: this function is nested inside function main()
    {
        std::cout << "foo!\n";
    }

    foo();
    return 0;
}

```

The right way to write is

```

void foo()
{
    std::cout << "foo!\n";
}

int main()
{
    foo();
    return 0;
}

```

3.1.2 Function return values (value-returning functions)

If we want to write a program that get number from user, then double it and return value to the console, we can start with:

```

#include <iostream>      // for std::cout

```

```

int main()
{
    // get a value from user
    std::cout << "Enter an integer: ";
    int num{};
    std::cin >> num;

    // print the value doubled
    std::cout << num << " doubled is: " << num * 2 << '\n';

    return 0;
}

```

We can also write a function that get value from user and let main() calls that function to double it:

```

#include <iostream>    // for std::cout

#| label: cod_problem
void getValueFromUser()
{
    std::cout << "Enter an integer: ";
    int input{};
    std::cin >> input;
}

int main()
{
    getValueFromUser();

    int num{};

    std::cout << num << " double is: " << num * 2 << '\n';

    return 0;
}

```

While this program is a good solution, it always returns 0 because the value from `getValueFromUser()` is not stored. We need that function return a value.

The function return a value require two conditions:

- Define the **return type** before the name of the function.

- A **return statement** in the body of the function. The function exits immediately, and a copy of return value backs to the *caller*. This is *return by value* process.

Let look at an example:

```
#include <iostream>

int returnValueFive()
{
    return 5;
}

int main()
{
    std::cout << returnValueFive() << '\n';
    std::cout << returnValueFive() + 2 << '\n';

    returnValueFive();

    return 0;
}
```

💡 When a called function returns a value, the caller may decide to use that value in an expression or statement(e.g by assigning it to a variable, or sending it to `std::cout`).

3.1.3 Fixing our challange program

Now, we return to the problem and use the code:

```
#include <iostream>

int getValueFromUser() // this function now returns an integer value
{
    std::cout << "Enter an integer: ";
    int input{};
    std::cin >> input;

    return input; // return the value the user entered back to the caller
}
```



```

int main()
{
    int num { getValueFromUser() }; // initialize num with the return value of getValueFromUser

    std::cout << num << " doubled is: " << num * 2 << '\n';

    return 0;
}

```

3.1.4 Revisiting main()

When we run a program, the operating system calls the function `main()`. It executes the `main()` from the top to bottom sequentially. Finally, `main()` returns an integer value (usually 0), and the program terminates. The return value from `main()` is called **status code**, and indicates that the program is successful or fail.



Best practice

- The main function should return the value 0 if the program runs normally.
- Make sure the functions with non-void return types return a value in all cases.
- Failure to return a value from a value-returning function will cause undefined behavior.



- Function `main` will implicitly return 0 if no return statement is provided. Thus, it is recommended to explicitly state the return value.
- Function can only return a **single value**. It can return not only literal, but also a variable or a call to function that return single value.
- The function author can decide what the return value means (status code, single value). It is good to document the function with a comment indicating what the return values mean.

3.1.5 Reusing functions

We can illustrate a good case for function reuse:

```

#include <iostream>

// Create a function that return an integer value from user's keyboard

```

```

int main()
{
    int x{};
    std::cout << "Enter an integer: ";
    std::cin >> x;

    int y{};
    std::cout << "Enter second integer: ";
    std::cin >> y;

    std::cout << x << " + " << y << " = " << x + y << '\n';

    return 0;
}

```

The central tenets of good programming: **Don't Repeat Yourself** (DRY). Let create a function:

```

#include <iostream>

// Create a function that return an integer value from user's keyboard
int getValueFromUser()
{
    int input{};
    std::cout << "Enter an integer: ";
    std::cin >> input;

    return input;
}

// main function return the sum of two integers from user
int main()
{
    int x{ getValueFromUser() };
    int y{ getValueFromUser() };

    std::cout << x << " + " << y << " = " << x + y << '\n';

    return 0;
}

```



Best practice

Don't repeat yourself. IF you need to do something more than once time, consider how to modify the code to remove as much as redundancy as possible.

3.2 Void functions

Functions are not required to return a value to the caller, we use the return type of **void**. For example:

```
#include <iostream>

void printHi()
{
    std::cout << "Hi!\n";
}

int main()
{
    printHi();

    return 0;
}
```

The function `printHi()` prints “Hi!”, but does not need to return anything back to the caller. A function that does not return a value is called a **non-value returning function (void-function)**.



Best practice

Do not put a return statement at the end of a non-value returning function.

Void functions can't be used in expression that require a value

Consider the following program:

```
#include <iostream>

void printHi()
{
```

```

        std::cout << "Hi!\n";
    }

    int main()
    {
        printHi();

        std::cout << printHi();

        return 0;
    }

```

An error occurs because the `printHi()` does not return value so it cannot provide the value to `std::cout` to print out.

Returning a value from a void function is a compile error.

Early return a return statement that is not the last statement in a function is called an **early return**.

For example:

```

#include <iostream>

void print()
{
    std::cout << "A";

    return;

    std::cout << "B";
}

int main()
{
    print();

    return 0;
}

```

The program returns “A” because it does not reach to second statements in ‘`print()`’.

3.3 Introduction to function parameters and arguments

3.3.1 The problem

We start by the codes:

```
#include <iostream>

void print()
{
    std::cout << "A";

    return;

    std::cout << "B";
}

int main()
{
    print();

    return 0;
}
```

We want to have a function that prints a doubled number:

```
#include <iostream>

int getValueFromUser()
{
    std::cout << "Enter an integer: ";
    int input{};
    std::cin >> input;

    return input;
}

void printDouble()
{
    std::cout << num << " doubled is: " << num * 2 << '\n';
}
```

```

int main()
{
    int num{ getValueFromUser() };

    std::cout << num << " doubled is: " << num * 2 << '\n';

    return 0;
}

```

The compiler does not know the definition of num in the function. Then, we can

```

#include <iostream>

int getValueFromUser()
{
    std::cout << "Enter an integer: ";
    int input{};
    std::cin >> input;

    return input;
}

void printDouble()
{
    int num{};
    std::cout << num << " doubled is: " << num * 2 << '\n';
}

int main()
{
    int num{ getValueFromUser() };

    printDouble();

    return 0;
}

```

The function run normally, but always returns 0 because function printDouble() does not access the input from user. Thus, we need some way to pass the value of variable to printDouble().

3.3.2 Function parameters and arguments

A **function parameter**: a variable used in header of a function, identically to variables defined inside the function, one difference: *they are initialized with a value provided by the caller of the function.*

An **argument** is a value that is passed from the *caller* to the function when a function call is made:

```
int main()
{
    doPrint();
    printValue(10);
    add(2, 3);

    return 0;
}
```

Use the commas for separation multiple arguments, and parameters.

Pass by value: when a function is called, all of the parameters are created as variables, and the value of each of the arguments is copied into the matching parameters.

```
// This function takes one parameter named x
// The caller will supply the value of x
void printValue(int x, int y)
{
    std::cout << x << '\n';
    std::cout << y << '\n';
}

int main()
{
    printValue(10, 20);

    return 0;
}
```

Now, we come back to our challenge program:

```
#include <iostream>

int getValueFromUser()
{
    std::cout << "Enter an integer: ";
```

```

    int input{};
    std::cin >> input;

    return input;
}

void printDouble(int input)
{
    std::cout << input << " doubled is: " << input * 2 << '\n';
}

int main()
{
    int num{ getValueFromUser() };
    printDouble(num);

    // printDouble(getValueFromUser());

    return 0;
}

```

The arguments can be also the expression, the variable:

```

#include <iostream>

int add(int x, int y)
{
    return x + y;
}

int multiply(int z, int w)
{
    return z * w;
}

int main()
{
    std::cout << add(5, 6) << '\n';
    std::cout << add(2 + 3, 3 * 4) << '\n';

    int a{ 5 };
    std::cout << add(a, a) << '\n';
}

```



```

std::cout << add(2, multiply(2, a)) << '\n';
std::cout << multiply(add(1, 3), multiply(2, 3)) << '\n';

return 0;
}

```

3.4 Introduction to local scope

3.4.1 Local variables

Function parameters as well as all variables defined inside the function body, are **local variables**.

Local variables are *created* at the point of definition and *destroyed* no earlier than the end of the set of the curly braces in which they are defined. We have definition of object's *lifetime* to be the time between its creation and destruction which happen when the program is running (calling runtime), not at compile time.

```

int add(int x, int y)
{
    int z{ x + y };

    return z;
} // z, y, and x destroyed here

```

Here is an example:

```

#include <iostream>

void doSomething()
{
    std::cout << "Hello!\n";
}

int main()
{
    int x{ 0 };
    doSomething();

    return 0;
} // x's lifetime ends here

```

3.4.2 Local scope

An identifier's **scope** determines where the identifier can be accessed within the source code. When an identifier can be accessed, it is **in scope**, while if we cannot access it, it is **out of scope**. Scope is a *compile-time* property.

A local variable's scope begins at the point of value definition, and stops at the end of the set of curly braces in which it is defined. Local variables defined in one function are also **not in scope** in other functions that are called.

Here is an example:

```
#include <iostream>

// x is not in scope anywhere in this function
void doSomething()
{
    std::cout << "Hello!\n";
}

int main()
{
    // x can not be used here because it's not in scope yet

    int x{ 0 }; // x enters scope here and can now be used within this function

    doSomething();

    return 0;
} // x goes out of scope here and can no longer be used
```



- *Out of scope*: an identifier cannot be accessed anywhere within the code
- *Going out of scope*: for objects, an object goes out of the scope at the end of the scope. A local variable's lifetime ends at the point where it goes out of scope.
- Not all types of variable are destroyed when they go out of scope.

Another example:

```
#include <iostream>

int add(int x, int y) // x and y are created and enter scope here
{
```

```

    // x and y are visible/usable within this function only
    return x + y;
} // y and x go out of scope and are destroyed here

int main()
{
    int a{ 5 }; // a is created, initialized, and enters scope here
    int b{ 6 }; // b is created, initialized, and enters scope here

    // a and b are usable within this function only
    std::cout << add(a, b) << '\n'; // calls function add() with x=5 and y=6

    return 0;
} // b and a go out of scope and are destroyed here

```

Key insight

Names used for function parameters or variables declared in a function body *only visible within the function that declares them*. Local variables within a function can be named without regard for the names of variables in other functions. This helps keep functions independent.

Best practice

Define the local variables as close to their first use as reasonable.

3.5 Why functions are useful, and how to use them effectively

3.5.1 Why use functions?

- *Organization:*
 - Simplify the main() function
 - A function is a mini-program that we can write separately
 - Help us to have more manageable chunks
- *Re-usability:*
 - Function can be called many times within a program
 - Follow “Do not repeat yourself” to minimize the copy/paste error
 - Can be shared with other program
- *Testing:*

- Functions reduces code redundancy, so less code to test
- Function is self-contained, when it is tested, we do not need to test it again unless we change it.
- Make easier to find bugs
- *Extensibility:*
 - When we need to extend the program to handle case it didn't handle before, function allow us to make change in one place
- *Abstraction:*
 - We only need: name, inputs, outputs and where it lives.
 - Do not need to know how it works, what other code it's dependent
 - Reduce the amount of knowledge

3.5.2 Effectively using functions

- When there are repeated group of statements
- Generally perform one and only one task

3.6 Forward declarations and definitions

We start by the simple program:

```
#include <iostream>

int main()
{
    std::cout << "The sum of 3 and 4 is: " << add(3, 4) << '\n';

    return 0;
}

int add(int x, int y)
{
    return x + y;
}
```

The compiler returns *error*: identifier add is not found. In the program, we defined add after main, and compiler reads the code sequentially so it cannot find add in main. There are two ways to address the issue:

3.6.1 Reorder the function definitions

We defined add before main:

```
#include <iostream>

int add(int x, int y)
{
    return x + y;
}

int main()
{
    std::cout << "The sum of 3 and 4 is: " << add(3, 4) << '\n';

    return 0;
}
```

It is good in the program. But in larger program, it is difficult to know which functions call which functions. For example, A calls B and B also calls A; we have trouble to find the appropriate order of functions. The second option will solve this problem.

3.6.2 Use a forward declaration

A **forward declaration** allows us to tell the compiler about the existence of a function before we define the function's body. The compiler encounter a call to the function, it will understanding and check whether we are calling function correctly, it doesn't know how or where the function is *defined*¹.

Here is an example of forward function declaration:

```
#include <iostream>

int add(int x, int y);

int main()
{
    std::cout << "The sum of 3 and 4 is: " << add(3, 4) << '\n';

    return 0;
}
```

¹this function does not exist in our Visual Studio yet.

```

}

int add(int x, int y)
{
    return x + y;
}

```

Forward declaration also helps to read the code from different files.

3.6.3 Declaration and definition

A **definition** actually implement (for function or types) or instantiates (for variables) the identifier.

```

int add(int x, int y)    // implement function add()
{
    int x{ }    // instantiates variable
}

```

A definition is needed to satisfy the *linker*. The **one definition rule** (ODR):

1. Within a file, a function, a variable, type, or template can only have one definition.
2. Within a given program, a variable or normal function can only have one definition
3. Types, templates, inline functions, and inline variables are allowed to have identical definitions in different files

3.7 Programs with multiple code files

We can create a another code file in the Solution Explorer. The code file contains the function called in main code file. Remember to declare the function in main source code so that the compiler checks and passes successfully.

After compiling, the program will link on the compiled files in the solution to execute.

In the project, we created a code file named *Add.cpp* to define the function add.

```

# In the add.cpp
int add(int x, int y)
{
    return x + y;
}

```

Together with the code file:

```
# In the main source code
#include <iostream>

int add(int x, int y);

int main()
{
    std::cout << "The sum of 3 and 4 is: " << add(3, 4) << '\n';

    return 0;
}
```



- Because the compiler compiles each code file individually, each code files that uses `std::cout`, `std::cin` needs to `#include<iostream>`. Similarly, we includes other libraries if we use the functions in them.
- Whenever you create a new code (.cpp) file, you will need to add it to your project so that it gets compiled.

3.8 Naming collisions and an introfuction to namespaces

C++ requires that all identifiers be **non-ambiguous**, otherwise the program produces an error as **name collision**. If the names conflict in the same file, there is a compile error; if it is in the separate files in the same program, there is a linker error.

C++ provides plenty of mechanisms for avoiding naming collisions.

- Local scopes: keep local variables defined inside functions from conflicting with each other
- But local scopes does not work for function names.

3.8.1 Namespace

Namespace is a region that allows us to declare names inside of it for the purpose of disambiguation; it provides a scope region called **namespace scope**.



- A name declared in a namespace will not be mistaken for an identical name declared in another scope.
- Within a namespace, all names are unique.
- Namespace often groups related identifiers in a large project to avoid collide with other identifiers.

The global namespace

Any name that is not defined inside a class, function, or a namespace is considered to be part of an implicitly defined namespace, **global namespace**.

Only declaration and definition statements can appear in the global namespace. We can define variable in the global namespace, *but this should generally be avoided..* Let read carefully the example code below:

```
#include <iostream> // handled by preprocessor

# All of the following statements are part of the global namespace
void foo();      # okay: function forward declaration in the global namespace
int x;           # compiles but strongly discouraged: uninitialized variable definition in the global namespace
int y { 5 };     # compiles but discouraged: variable definition with initializer in the global namespace
x = 5;           # compile error: executable statements are not allowed in the global namespace

int main()       # okay: function definition in the global namespace
{
    return 0;
}

void goo();      # okay: another function forward declaration in the global namespace
```

The std namespace

We usually typed `std::cout`: we tell the compiler to search for the function `cout` inside the namespace `std`. This approach avoid any collide with any functions in other library or namespace.



When you use an identifier that is defined a namespace (such as the `std` namespace), you have to tell the compiler that the identifier lives inside the namespace.

⚠️ Avoid using-directives (such as `using namespace std;`) at the top of your program or in header files. They violate the reason why namespaces were added in the first place.

3.9 Introduction to the preprocessor

Each code file goes through a preprocessing phase; a program called preprocessor make various changes to the text of the code file.

This phase works with the `#include`, `#define`. When we use `#include`, the preprocessor copies all the files to the compiler.

3.10 Header files

What do we do if we want to store all the declaration in one files? That the important role of **header file** (extension `.h`).

Header file contains the function and variable declarations, not *definitions*. We can create our own header file (paired with the function code) and `#include nameOfFile.h` in the main and in the functions that declared in the header file.

Best practice

- Use a .h suffix when naming your header files.
- If a header file is paired with a code file (e.g. add.h with add.cpp), they should both have the same base name (add).
- Source files should #include their paired header file (if one exists).
- Use double quotes to include header files that you've written or are expected to be found in the current directory. Use angled brackets to include headers that come with your compiler, OS, or third-party libraries you've installed elsewhere on your system.
- When including a header file from the standard library, use the version without the .h extension if it exists. User-defined headers should still use a .h extension.
- Each file should explicitly #include all of the header files it needs to compile. Do not rely on headers included transitively from other headers.
- Each header file should have a specific job, and be as independent as possible. For example, you might put all your declarations related to functionality A in A.h and all your declarations related to functionality B in B.h. That way if you only care about A later, you can just include A.h and not get any of the stuff related to B.
- Be mindful of which headers you need to explicitly include for the functionality that you are using in your code files
- Every header you write should compile on its own (it should #include every dependency it needs)
- Only #include what you need (don't include everything just because you can).
- Do not #include .cpp files.

The author discuss the use of **Header Guard** helps to avoid the problem of collision between files in the program when creating headers.

3.11 How to design your first programs

The most important to remember is to design the program *before you start coding*. It is like architecture. A little up-front planning saves both time and frustration in the long run.

3.11.1 Design step 1: Define your goal

Should address the goal in one or two sentences, often useful to express this as a user-facing outcome. Example:

- Allow the user to organize a list of names and associated phone numbers
- Model how long it takes for a ball dropped off a tower to hit the ground

3.11.2 Design step 2: Define requirements

This focuses on the “what”, not the “how”. For example:

- Phone numbers should be saved, so they can be recalled later
- The program should produce results within 10 seconds of the user submitting their request

A single problem may yield many requirements, and the solution is not “done” until it satisfies all of them.

3.11.3 Design step 3: Define your tools, targets, and backup plan

As a new programmer, the answers are simple:

- Writing a program for own use
- On own system
- Using IDE purchased or downloaded

3.11.4 Design step 4: Break hard problems down into easy problems

Try split the complex task to sub-task and continuing define smaller sub-task until they are manageable.

3.11.5 Design step 5: Figure out the sequence of events

Then, we determine how to link all tasks together. For example, if we are writing a calculator, we might do things in this order:

- Get first number from user
- Recognize the operator from user
- Get second number from user
- Calculate result
- Print result back to user

Then, we can implement them

3.11.6 Implementation step 1: Outlining the main function

Do not worry about input and output for the time being

```
int main() {  
    // Get first number from user  
    // getUserNumber();  
  
    // Recognize the operator from user  
    // getMathOperation()  
  
    // Get second number from user  
    // getUserNumber();  
  
    // Calculate result  
    // calculateResult();  
  
    // Print result  
    // printResult();  
  
    return 0;  
}
```

3.11.7 Implementation step 2: Implement each function

In this step, for each function, we do three things:

- Define the function prototypes (input, output)
- Write the function
- Test the function

```
#include <iostream>  
  
int getUserNumber() {  
  
    std::cout << "Enter an integer: ";  
  
    int number{};  
    std::cin >> number;
```

```

        return number;
    }


    int main() {

        // Get first number from user
        int value{ getUserNumber() };
        std::cout << value << '\n';    //debug code

        // Recognize the operator from user

        return 0;
    }

```

 Do not implement your entire program in one go. Work on it in steps, testing each step along the way before proceeding.

3.11.8 Implementation step 3: Final testing

Once your program is “finished”, the last step is to test the whole program and ensure it works as intended. If it doesn’t work, fix it.

Word of advice

- **Keep your programs simple to start:** make your first goal as simple as possible.
- **Add features over time:** Once you have your simple program working and working well, then you can add features to it
- **Focus on one area at a time:** Focus on one task at a time
- **Test each piece of code as you go.**
- **Do not invest in perfecting early code**

The good news is that once you become comfortable with all of these concepts, they will start coming more naturally to you. Eventually you will get to the point where you can write entire functions without any pre-planning at all.

4 Fundamental Data Types

4.1 Introduction to fundamental data types

i Recall

Variable is a names of a piece of memory that we will store a certain type of information (data) into.

All data on a computer is just a sequence of bits, we use a **data type** to tell the compiler **how to interpret** the contents of memory in some meaningful way.

When we give an object a value, the compiler and CPU covert it to bit and store in a memory; when we consult the value, the bit is recovered to the human-readable format.

4.1.1 Fundamental data types

Table belows summray the **data types** in C++:

4.1.2 Void

Void means no type. Then, a variables cannot be defined with a type of void.

void uses with function does not return value.

```
#include<iostream>

void writeValue(int x)
{
    std::cout << "The value is: " << x << '\n';
}
```

If we return value, compiler returns error.

Types	Category	Meaning	Example
float double long double	Floating Point	a number with a fractional part	3.14159
bool	Integral (Boolean)	true or false	true
char wchar_t char8_t (C++20) char16_t (C++11) char32_t (C++11)	Integral (Character)	a single character of text	'c'
short int long long long (C++11)	Integral (Integer)	positive and negative whole numbers, including 0	64
std::nullptr_t (C++11)	Null Pointer	a null pointer	nullptr
void	Void	no type	n/a

Figure 4.1: Data types

4.1.3 Object sizes and the size of operator

```
#| label: typeSize

#include <iostream>

int main()
{
    std::cout << "bool:\t\t" << sizeof(bool) << " bytes\n";
    std::cout << "char:\t\t" << sizeof(char) << " bytes\n";
    std::cout << "wchar_t:\t" << sizeof(wchar_t) << " bytes\n";
    std::cout << "char16_t:\t" << sizeof(char16_t) << " bytes\n";
    std::cout << "char32_t:\t" << sizeof(char32_t) << " bytes\n";
    std::cout << "short:\t\t" << sizeof(short) << " bytes\n";
    std::cout << "int:\t\t\t" << sizeof(int) << " bytes\n";
    std::cout << "long:\t\t\t" << sizeof(long) << " bytes\n";
    std::cout << "long long:\t" << sizeof(long long) << " bytes\n";
    std::cout << "float:\t\t\t" << sizeof(float) << " bytes\n";
    std::cout << "double:\t\t\t" << sizeof(double) << " bytes\n";
    std::cout << "long double:\t" << sizeof(long double) << " bytes\n";
}
```

```
    return 0;
}
```

4.1.4 Signed integer

Signed integer is integer with sign operator. These are the conventional definition of integer:

```
short s;           // short int
int i;
long l;            // long int
long long ll;      // long long int
```

4.1.5 Floating point numbers

For floating numbers, we have the appropriate value; `std::cout` displays from 6 to 8 digits. We can override this format by use `std::setprecision(#digits)` from headers `iomanip`.

The precision does not only impact the fractional number, they impact any number with too many significant digits:

```
#include <iostream>
#include <iomanip>      // for std::setprecision()

int main()
{
    float f{ 123456789.0f };
    std::cout << std::setprecision(9);
    std::cout << f << '\n';

    return 0;
}

#> 123456792
```

In the example, 1234567892 is greater than 123456789. The value of 123456789.0 has 10 significant digits, but float values typically have 7 digits of precision, then we lost precision



Best practice

Favor double over float type.

4.1.6 Rounding errors makes floating point comparison tricky

Floating point numbers are tricky to work due to non-obvious differences between binary and decimal. The fraction 0.1, in binary, 0.1 represented by *infinite* sequence 0.00011001100110011... Thus, we assign 0.1 to floating point number, we run into **rounding errors**.

Let see the following code:

```
#include <iostream>
#include <iomanip>      // for std::setprecision()

int main()
{
    double d{ 0.1 };
    std::cout << d << '\n'; // use default cout precision of 6
    std::cout << std::setprecision(17);
    std::cout << d << '\n';

    return 0;
}

#> 0.1
#> 0.1000000000000001
```

The double type guarantee precision for 16 digits, when there are more than that, there is not precision. Another one

```
#include <iostream>
#include <iomanip>      // for std::setprecision()

int main()
{
    std::cout << std::setprecision(17);

    double d1{ 1.0 };
    std::cout << d1 << '\n';

    double d2{ 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 }; // should equal 1.0
    std::cout << d2 << '\n';

    return 0;
}
```

```
#> 1  
#> 0.9999999999999999999999999999999
```

4.1.7 Boolean values

Boolean type takes only two value: 'true' or 'false'. In C++, the system represents it as an integer which 0 for 'false' and 1 for 'true'. If we want to print the values rather than number, we have to use the function `std::boolalpha`.

```
#include <iostream>
#include <iomanip>          // for std::setprecision()

int main()
{
    std::cout << true << '\n';
    std::cout << false << '\n';

    std::cout << std::boolalpha;

    std::cout << true << '\n';
    std::cout << false << '\n';

    return 0;
}

#> 1
#> 0
#> true
#> false
```

This is similar to the get information from user:

```
#include <iostream>
#include <iomanip>          // for std::setprecision()

int main()
{
    std::cout << "Enter a boolean value: ";

    bool b{};
    std::cin >> b;
```

```

        std::cout << "You enter: " << b << '\n';

        return 0;
    }

```

```

#> Enter a Boolean value: true
#> You entered: 0

```

We can do:

```

#include <iostream>
#include <iomanip>          // for std::setprecision()

int main()
{
    std::cout << "Enter a boolean value: ";

    bool b{};
    std::cin >> std::boolalpha;
    std::cin >> b;

    std::cout << "You enter: " << b << '\n';

    return 0;
}

```

Boolean return value

We can create a function that return boolean value:

```

#include <iostream>
#include <iomanip>          // for std::setprecision()

bool isEqual(int x, int y) {
    return (x == y);
}

int getInteger() {
    std::cout << "Enter an integer: ";
    int x{};
}

```

```

        std::cin >> x;

        return x;
    }
    int main() {

        int x{ getInteger() };
        int y{ getInteger() };

        std::cout << std::boolalpha;
        std::cout << "Does " << x << " equal " << y << "?\n";
        std::cout << isEqual(x, y) << '\n';

        return 0;
    }


```

4.2 Introduction to if statement

Conditional statement is common in programming language; they allow us to implement conditional behavior into our programs.

The simplest type of conditional statement in C++ is *if statement*, following the form:

```
if (condition) true_statement;
```

 If statements only conditionally execute a single statement. We talk about how to conditionally execute multiple statements in lesson 7.2 – If statements and blocks.

4.3 Chars

4.4 An introduction to explicit type conversion via the static_cast operator

If we want to explicitly change the type of data, we can use `static_cast<new_type>(expression)`.¹

¹We see similar term in R: `vctrs::vec_cast()` function that change the type of vector.

```

#include <iostream>

void print(int x)
{
    std::cout << x;
}

int main()
{
    print( static_cast<int>(5.5) ); // explicitly convert double value 5.5 to an int

    return 0;
}

```

We can use this to covert char to int:

```

#include <iostream>

int main() {

    char ch{ 'a' };
    std::cout << ch << '\n';

    std::cout << ch << " has value " << static_cast<int>(ch) << '\n';

    return 0;
}

```

Another exercise:

```

int main() {

    std::cout << "Please enter a single character: ";
    char ch{};
    std::cin >> ch;

    std::cout << "You entered '" << ch << "' << ", which has ASCII code " << static_cast<int>(ch) << '\n';

    return 0;
}

```

4.5 Const variables and symbolic constants

Const variable: initialized variable with value and cannot assign different values later.

Let look the code to create a const variable:

```
#include <iostream>

int main()
{
    const double gravity{ 9.8 };
    gravity = 9.9;

    return 0;
}

#> Error    C3892    'gravity': you cannot assign to a variable that is const
```

Age example:

```
#include <iostream>

int main()
{
    std::cout << "Enter your age: ";

    int age{};
    std::cin >> age;

    const int constAge { age };

    age = 5          //okie
    constAge = 6     // error: constAge is constant

    return 0;
}
```

We can set the const for parameter, but do not prefer to do that.

4.6 Compile-time constants, constant expressions, and constexpr

Constant expression is an expression that can be evaluated by the compiler at **compile time**.

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

i A const variable is a compile-time constant if its initializer is a *constant expression*.
Evaluating const expression takes longer at compile time, but saves time at runtime.

Runtime const: every const variable initialized with a non-constant expression is a *runtime constant*.
Runtime constants are constants whose initialization values are not known until runtime.

4.6.1 The constexpr keyword

Compile always implicitly keep track of const variables to know whether it's a runtime or compile-time const. A **constexpr** variable can **only** be compile-time const.

Let read the example:

```
// constexpr uses
constexpr double gravity{ 9.8 };    // ok: 9.8 is a const expression
constexpr int sum{ 4 + 5 };         //ok
constexpr int something{ sum };     // ok: sum is const expression

std::cout << "Enter you age: ";
int age{};
std::cin >> age;

constexpr int myAge{ age };         // compile error: age is not const expression
constexpr int f{ five() };         // compile error: return value of five() is not constant ex
```

Best practice

- Any variable that *should not be modifiable* after initialization and whose initializer is **known** at compile-time should be declared as constexpr.
- Any variable that *should not be modifiable* after initialization and whose initializer is **not** known at compile-time should be declared as const.

4.7 Literals

Literals are unnamed values inserted directly into the code. For example:

```

return 5;
bool myNameIsAlex{ true };
std::cout << 3.4;

```

Like objects, all literals have type. Their type is in

4.8 String

An example of string is "Hello world?\n"; a sequence of characters. String is not natural in C++ language. We use `std::string` to work with string by including the header `<string>` in the header

Here is an example:

```

#include <string>
#include <iostream>

int main()
{
    //std::string name{};          // empty string
    std::string name{ "Huy" };      // initialized with value
    name = "Quang";

    std::string myID{ "45" };

    std::cout << "My name is " << name << '\n';

    std::string empty{};
    std::cout << '[' << empty << ']';

    return 0;
}

```

4.8.1 String input with `std::cin`

Using strings with `std::cin` may yield some surprises. Let's see the example:

```

#include <string>
#include <iostream>

int main()

```



```

{
    std::cout << "Enter your full name: ";
    std::string name{};
    std::cin >> name;

    std::cout << "Enter your age: ";
    std::string age{};
    std::cin >> age;

    std::cout << "Your name is " << name << " and your age is " << age << '\n';

    return 0;
}

#> Input: Quang Huy
#> Your name is Quang and your age is Huy

```

Normally, the >>operator only extract the string before first space, and the rest is stored for the next >>operator.

To get the full string input, it is better to use `std::getline()`:

```

#include <string>
#include <iostream>

int main()
{
    std::cout << "Enter your full name: ";
    std::string name{};
    std::getline(std::cin >> std::ws, name);

    std::cout << "Enter your age: ";
    std::string age{};
    std::getline(std::cin >> std::ws, age);

    std::cout << "Your name is " << name << " and your age is " << age << '\n';

    return 0;
}

```

The input manipulator `std::ws` tells `std::` to ignore any leading whitespace (spaces, tabs, newlines) in the string. Here is another example:

```

#include <string>
#include <iostream>

int main()
{
    std::cout << "Pick 1 or 2: ";
    int choice{};
    std::cin >> choice;

    std::cout << "Now enter your name: ";
    std::string name{};
    std::getline(std::cin >> std::ws, name);        // no std::ws here
    std::cout << "Hello " << name << ", you chose " << choice << '\n';

    return 0;
}

```



Best practice

If using `std::getline()` to read strings, use `std::cin >> std::ws` input manipulator to ignore leading whitespace.

4.8.2 String length

We use the `length()` function to return the string's length. In C++, the `length` function is nested in `std::string`, it is written as `std::string::length()` in documentation. Thus, `length()` is not normal standalone function, it is called *member function*.



Key insight

With normal function, we call `function(object)`. With member function, we call `object.function()`.

`length()` return unsigned integral value, to assign it to integer value, we have to use `static_cast<type><name.length()>`. We can also use `ssize(name)` to get the length as a signed integer.²

²this function does not exist in our Visual Studio yet.



Best practice

Do not pass `std::string` by value, as making copies of `std::string` is expensive. Prefer `std::string_view` parameters.

4.8.3 Literals for `std::string`

We can create string literals with type `std::string` by using `s` suffix after the double-quote string literal.

Here is the exercise from this session:

```
#include <iostream>
#include <string>
#include <string_view>

int main()
{
    std::cout << "Enter your full name: ";
    std::string name{};
    std::getline(std::cin >> std::ws, name);

    std::cout << "Enter your age: ";
    int age{};
    std::cin >> age;

    //std::cout << "Your age + length of your name is: " << age + static_cast<int>(name.length());

    int letters{ static_cast<int>(name.length()) };
    std::cout << "Your age + length of your name is: " << age + letters << '\n';

    return 0;
}
```

4.9 Introduction to `std::string_view`

When use `std::string`, it creates a copy of the string, this is inefficient. Thus, from C++ 17, there is `std::string_view` from `string_view` header that provide **read-only access** to an existing string, not a copy of a string.

Here is an example:

```

#include <iostream>
#include <string>
#include <string_view>

void printSV(std::string_view str)
{
    std::cout << str << '\n';
}

int main()
{
    std::string_view s{ "Hello world!" };
    printSV(s);

    return 0;
}

```

Unlike `std::string`, `std::string_view` support `constexpr`:



- Returning a `std::string_view` from a function is usually a bad idea. We will explore why in lesson 11.7 – `std::string_view` (part 2). For now, avoid doing so.
- The chapter summary can be found at <https://www.learncpp.com/cpp-tutorial/chapter-4-summary-and-quiz/>

5 Debugging C++ Programs

5.1 Syntax and semantic errors

Software errors are prevalent. It's easy to make them, and it's hard to find them

Two categories of errors:

- **Syntax error:** write a statement that is not valid according to the grammar of C++ (missing semicolons, using undeclared variables, missing braces, etc...). A compiler will catch these errors, so we identify and fix them. For example:

```
#include <iostream>

int main()
{
    std::cout < "Hi there"; << x << '\n'; // invalid operator (<), extraneous semicolon, undeclared x
    return 0 // missing semicolon at end of statement
}
```

- **Semantic error** occurs when a statement is syntactically valid, does not do what we want.
 - Sometimes it causes the program to crash

```
#include <iostream>

int main()
{
    int a { 10 };
    int b { 0 };
    std::cout << a << " / " << b << " = " << a / b << '\n'; // division by 0 is undefined
    return 0;
}
```

- Or wrong value

```
#include <iostream>

int main()
{
    int x;
    std::cout << x << '\n'; // Use of uninitialized variable leads to undefined result

    return 0;
}
```

• Or

```
#include <iostream>

int main()
{
    return 0; // function returns here

    std::cout << "Hello, world!\n"; // so this never executes
}
```

5.2 The debugging process

Generally, there are 5 steps:

1. Find the root cause of the problem
2. Ensure you understand why the issue is occurring
3. Determine how you'll fix the issue
4. Repair the issue causing the problem
5. Retest to ensure the problem has been fixed and no new problems have emerged

5.2.1 Debugger

A **debugger** is a computer program that allows the programmer to control how another program executes and examine the program state while that program is **running**. For example, a programmer can use debugger to execute a program line by line, examining the value of variables along the way.

The advantage:

- The ability to precisely control execution of program
- To view the program state



Do not neglect learning to use a debugger. As your programs get more complicated, the amount of time you spend learning to use the integrated debugger effectively will pale in comparison to amount of time you save finding and fixing issues.

5.2.2 The call stack

The **call stack** is a list of all active functions that have been called to get to the current point of execution. It includes an entry for each function called, as well as which line code will be returned to when the function returns. Whenever a new functions is called, that function is added to the top of the call stack.

6 Operators

6.1 Introduction

An **operation** is an mathematical calculation involving zero or more input values (called **operand**) to produce a new value. The specific operation to be performed is denoted by a construct (symbol or pair of symbols: +, -, "*", "/",) called an **operator**.

Operator precedence:

- In the compound expression (including more than 1 operators), C++ use the rule of **operator precedence**. Each operator is assigned one level, the higher level operator will be evaluate first.
- When the operators have the same level, C++ uses **operator associativity** that indicates whether the it analyzes from left-to-right or right-to-left.
- We can use the pasteurization for explicitly say the order of the operator



Best practice

Use parentheses to make it clear how a non-trivial expression should evaluate (even if they are technically unnecessary).

For example

```
x = (y + z + w);    // instead of this
x = y + z + w;      // it's okay to do this

x = ((y || z) && w); // instead of this
x = (y || z) && w;   // it's okay to do this

x = (y *= z); // expressions with multiple assignments still benefit from parenthesis
```



In many cases, the operands in a compound expression may evaluate **in any order**. This includes function calls and the arguments to those function calls.

We can have the mistakes that the compiler chooses the order differs from what we want:


```
#include <iostream>

int getValue()
{
    std::cout << "Enter an integer: ";
    int x{};
    std::cin >> x;

    return x;
}

int main()
{
    std::cout << getValue() + (getValue() * getValue());

    return 0;
}
```



Best practice

Outside of the operator precedence and associativity rules, assume that the parts of an expression could evaluate in any order. Ensure that the expressions you write are not dependent on the order of evaluation of those parts.

Example for best practice:

```
#include <iostream>

int getValue()
{
    std::cout << "Enter an integer: ";
    int x{};
    std::cin >> x;

    return x;
}

int main()
{
    int a{ getValue() };
    int b{ getValue() };
    int c{ getValue() };
}
```

```

    std::cout << a + (b * c);

    return 0;
}

```

6.2 Arithmetic operators

There are two unary arithmetic operators:

Operator	Symbol	Form	Operation
Unary plus	+	+x	Value of x
Unary minus	-	-x	Negation of x

Figure 6.1: Unary Arithmetic

The binary arithmetic operators:

Operator	Symbol	Form	Operation
Addition	+	x + y	x plus y
Subtraction	-	x - y	x minus y
Multiplication	*	x * y	x multiplied by y
Division	/	x / y	x divided by y
Modulus (Remainder)	%	x % y	The remainder of x divided by y

Figure 6.2: Binary Arithmetic

Arithmetic assignment operators

6.3 Increment/decrements operators, and side effects

Let have an example:

Operator	Symbol	Form	Operation
Assignment	=	x = y	Assign value y to x
Addition assignment	+=	x += y	Add y to x
Subtraction assignment	-=	x -= y	Subtract y from x
Multiplication assignment	*=	x *= y	Multiply x by y
Division assignment	/=	x /= y	Divide x by y
Modulus assignment	%=	x %= y	Put the remainder of x / y in x

Figure 6.3: Arithmetic Assignment operators

Operator	Symbol	Form	Operation
Prefix increment (pre-increment)	++	++x	Increment x, then return x
Prefix decrement (pre-decrement)	--	--x	Decrement x, then return x
Postfix increment (post-increment)	++	x++	Copy x, then increment x, then return the copy
Postfix decrement (post-decrement)	--	x--	Copy x, then decrement x, then return the copy

Figure 6.4: Increment/decrement operators

```

#include <iostream>

int main()
{
    int x{ 5 };
    int y{ 5 };

    std::cout << x << ' ' << y << '\n';
    std::cout << ++x << ' ' << --y << '\n'; // prefix make the calculation
                                           // and assign directly

    std::cout << x << ' ' << y << '\n';

    std::cout << x++ << ' ' << y-- << '\n'; // postfix creates a copy, calcul,
                                           // and return the copy

    std::cout << x << ' ' << y << '\n';

    return 0;
}

```



Best practice

Strongly favor the prefix version.

6.3.1 Side effects can cause undefined behavior

A function or expression is said to have a **side effect** if has some observable effect beyond producing a return value.

Common examples are assignment value of objects, doing input, output,... Most of the time, side effects are useful:

```

x = 5;
++x;
std::cout << x

```

However, side effects can also lead to unexpected results:

```

int add(int x, int y)
{
    return x + y;
}

```

```
int main()
{
    int x{ 5 };

    int value{ add(x, ++x) };
    std::cout << value << '\n'; // value depends on how the above line evaluates

    return 0;
}
```



- C++ does not define the order of evaluation for function arguments or the operands of operators.
- Don't use a variable that has a side effect applied to it more than once in a given statement. If you do, the result may be undefined.

6.4 Comma and conditional operators

Operator	Symbol	Form	Operation
Comma	,	x, y	Evaluate x then y, returns value of y

Figure 6.5: Comma operator

The comma operator allows us to evaluate multiple expressions wherever a single expression is allowed.

In almost every case, a statement written using the comma operator would be better written as a separate statements.



Best practice

Avoid using the comma operator, except within for loops

6.4.1 Comma as a separator

In C++, comma symbol is often used as a separator, and does not invoke the comma operator:

```
void foo(int x, int y) // Comma used to separate parameters in function definition
{
    add(x, y); // Comma used to separate arguments in function call
    constexpr int z{ 3 }, w{ 5 }; // Comma used to separate multiple variables being defined on
}
```

6.4.2 Conditional operator


Operator	Symbol	Form	Operation
Conditional	<code>?:</code>	<code>c ? x : y</code>	If <code>c</code> is nonzero (true) then evaluate <code>x</code> , otherwise evaluate <code>y</code>

Figure 6.6: Conditional operator

Always parenthesize the conditional part of the conditional operator, and consider parenthesizing the whole thing as well.

The conditional operator evaluates as an expression

Because the conditional operator operands are expressions rather than statements, the conditional operator can be used in some places where `if/else` cannot.

 The type of expressions in conditional operator must match or be convertible.

The conditional operator gives us a convenient way to compact some `if/else` statements. It's most useful when we need a conditional initializer (or assignment) for a variable, or to pass a conditional value to a function.

6.5 Relational operators and floating point comparisons

Comparison of **calculated** floating point values can be problematic.

```
#include <iostream>

int main()
{
    double d1{ 100.0 - 99.99 }; // should equal 0.01 mathematically
    double d2{ 10.0 - 9.99 }; // should equal 0.01 mathematically
}
```

Operator	Symbol	Form	Operation
Greater than	>	$x > y$	true if x is greater than y, false otherwise
Less than	<	$x < y$	true if x is less than y, false otherwise
Greater than or equals	>=	$x \geq y$	true if x is greater than or equal to y, false otherwise
Less than or equals	<=	$x \leq y$	true if x is less than or equal to y, false otherwise
Equality	==	$x == y$	true if x equals y, false otherwise
Inequality	!=	$x != y$	true if x does not equal y, false otherwise

Figure 6.7: Relational Operators

```

if (d1 == d2)
    std::cout << "d1 == d2" << '\n';
else if (d1 > d2)
    std::cout << "d1 > d2" << '\n';
else if (d1 < d2)
    std::cout << "d1 < d2" << '\n';

return 0;
}
#> d1 > d2 : error

```

Avoid using operator== and operator!= to compare floating point values if there is any chance those values have been calculated.

7 Scope, duration and linkage

7.1 Compound statements

Compound statement (block):

- Zero, one, or more statement that compiler treats as simple statement
- Between “{”, not need semicolon at the end of “}”
- Be anywhere when the single statement is valid

Function cannot be nested inside other function, but *block* can. Then we have

- Outer block: enclosing one
- Inner block: nested one

Block is very useful with `if` statements: replace single statement with a block. For example,

```
#include <iostream>

int main()
{
    std::cout << "Enter an integer: ";
    int value{};
    std::cin >> value;

    if (value >= 0)
    {
        std::cout << value << " is a positive integer (or zero)\n";
        std::cout << "Double this number is " << value * 2 << '\n';
    }
    else
    {
        std::cout << value << " is a negative integer\n";
        std::cout << "The positive of " << value << " is " << -value << '\n';
        std::cout << "Then, its double is " << (-value) * 2 << '\n';
    }
}
```



```
    return 0;
}
```

Nesting level or **nesting depth** is the maximum numbers of nested blocks inside at any point in the function. For example,

```
#include <iostream>

int main()
{ // block 1, nesting level 1
    std::cout << "Enter an integer: ";
    int value {};
    std::cin >> value;

    if (value > 0)
    { // block 2, nesting level 2
        if ((value % 2) == 0)
        { // block 3, nesting level 3
            std::cout << value << " is positive and even\n";
        }
        else
        { // block 4, also nesting level 3
            std::cout << value << " is positive and odd\n";
        }
    }

    return 0;
}
```

There are 4 blocks, and nesting level is 3.



Best practice

Keep the nesting level of our functions to **3 or less**. If greater, think to re-factor the function into sub-ones.

7.2 User-defined namespaces and the scope resolution operator

The better way to solve the name collisions is to put our functions in our **own namespaces** eg, standard library moved into std namespace.

7.2.1 Define our own namespaces

We define namespace by using statement namespace

in foo.cpp file:

```
namespace foo
{
    int doSomething(int x, int y)
    {
        return x - y;
    }
}
```

in goo.cpp file:

```
namespace goo
{
    int doSomething(int x, int y)
    {
        return x - y;
    }
}
```

in main.cpp:

```
#include <iostream>

int doSomething(int x, int y);

int main()
{
    std::cout << "Hello World!\n";

    std::cout << doSomething(4, 3) << '\n';

    return 0;
}
```

Running the program, there is “LINK ERROR”: cannot find the doSomething in global namespace because it is in either foo or goo namespace.

=> Solve this problem by: scope resolution or using statements

Accessing namespace with the scope resolution operator (::)

We use the operator (::) to tell exactly whose namespace of the called function. For example in main.cpp

```
#include <iostream>

int doSomething(int x, int y);

int main()
{
    std::cout << "Hello World!\n";

    std::cout << foo::doSomething(4, 3) << '\n';

    return 0;
}
```

If an identifier inside a namespace is used, without scope resolution, the compiler finds inside the namespace, if it cannot find, it goes to each containing namespace in sequence to find, the global namespace being checked last.

```
#include <iostream>

void print() // this print lives in the global namespace
{
    std::cout << " there\n";
}

namespace foo
{
    void print() // this print lives in the foo namespace
    {
        std::cout << "Hello";
    }

    void printHelloThere()
    {
        print(); // calls print() in foo namespace
        ::print(); // calls print() in global namespace
    }
}
```

```
int main()
{
    foo::printHelloThere();

    return 0;
}
```

If we define namespace in different files to main.cpp:

- Create header file for the namespace. For example, namespace foo:

```
#ifndef FOO_H
#define FOO_H

namespace foo
{
    int add(int x, int y);

    int subtract(int x, int y);
}

#endif
```

- Include the header into the namespace file foo.cpp:

```
#include "foo.h"

namespace foo
{
    int add(int x, int y)
    {
        return x + y;
    }

    int subtract(int x, int y)
    {
        return x - y;
    }
}
```

- Add header to main.cpp:

```

#include "foo.h"
#include <iostream>

int main()
{
    std::cout << "Hello World!\n";

    std::cout << foo::add(4, 3) << '\n';
    std::cout << foo::subtract(10, 7) << '\n';

    return 0;
}

```

Namespace can be nested in other namespace by declaration namespace foo::goo.

Namespace aliases

For nested namespace, C++ allow to name the namespace for shortenning:

```

#include <iostream>

namespace foo::goo
{
    int add(int x, int y)
    {
        return x + y;
    }
}

int main()
{
    namespace active = foo::goo; // active refers to foo::go

    std::cout << active::add(1, 2) << '\n'; // This is really foo::goo::add()

    return 0;
}

```



- In general, we should avoid deeply nested namespaces.
- We can separate application-specific code from code that might be reusable later. For example, one namespace for math: `math::`, language and localization: `lang::`
- When sharing the code, should put it in a namespace.

7.3 Local variables

Local variables have a **block scope**: *in scope* from point of definition to the end of the block they are defined within.

```
#|label: block-scope

#include <iostream>

int main()
{
    int i{ 5 };           // i enters scope
    double d{ 4.0 };     // d enters scope

    return 0;
} // d and i go out of scope here.
```

All variable names within a scope must be unique.

A local variable has a lifetime, called storage **duration**. This determine when and how a variable will be created and destroyed.

Local variable has **automatic storage duration**: created and destroyed within the block

=> local variable == **automatic variable**.

We can define variables inside nested block:

- They enters the scope at time of definition and go out at the end of the nested block. So do their lifetimes.
- Thus, it cannot be found from outer block
- But, variable in outer block can be used inside nested block.

```

#include <iostream>

int main()
{ // outer block

    int x { 5 }; // x enters scope and is created here

    { // nested block
        int y { 7 }; // y enters scope and is created here

        // x and y are both in scope here
        std::cout << x << " + " << y << " = " << x + y << '\n';
    } // y goes out of scope and is destroyed here

    // y can not be used here because it is out of scope in this block

    return 0;
} // x goes out of scope and is destroyed here

```

Variable should be defined in the most limited scope: inside the nested block that uses it.

💡 Define variables in the most limited existing scope. Avoid creating new blocks whose only purpose is to limit the scope of variables.

7.4 Introduction to global variables

We define a variable outside of a function, this one is **global variable**. By convention:

- It is under the `#include` and above any function.
- Consider using prefix 'g' or 'g_' for global variables.

Its properties:


- Has global scope (global namespace scope)
- Uses anywhere in the file
- Created when the program starts and destroyed when it ends => **static duration** So, it is **static variable**.
- Zero-initialized by default¹.

¹local variables have uninitialized default

- Global variable can be defined as constant, but we have to initialize it:

```
const int g_x; //error: no default initialized


const int g_w{1}; // ok
```

 We should avoid to use non-constant global variables.

7.5 Variable shadowing (name hiding)

When outer block and inner block have a variable with same name, the nested variable “hides” the outer variable when they are both in scope. This is called **name hiding** or **shadowing**.

The same applies to global variable, but global variable is a part of namespace so we can use `::` prefix to indicate using global variable.

 Avoid variable shadowing

7.6 Internal linkage

An identifier’s linkage determines whether the other declarations of that name refer to the same object or not.

- Local variable does not have linkage.
- Global variable has internal and external linkage

An **internal linkage** is seen and used within a single file.

A global variable has external linkage as default, so we define internal linkage by use `static`

But `const` global variable or `constexpr` global variable are internal linkage by default.

```
static int g_x{}; // internal linkage
const int g_y{ 2 }; // internal linkage by default
constexpr int g_z{ 3 }; //internal linkage by default
```


The one-definition rule and internal linkage

One-definition rule says that an object or function cannot have more than one definition, either within a file or a program. But, internal object are defined in different files are considered to be independent entities.

We can apply the same to function by using static:

```
# This function is declared as static, and can now be used only within this file
# Attempts to access it from another file via a function forward declaration will
# fail

[[maybe_unused]] static int add(int x, int y)
{
    return x + y;
}
```

7.7 External linkage and variable forward declarations

Identifier with **external linkage** can be used and seen from its file and other code files (via a forward declaration).

- Function have *external linkage by default* so we can use a forward declaration
- Non-const global variable are external by default
- Const global variable defined through extern keyword
- To use global variable in other file, we also forward declaration like in function,

The following code are example of using external linkage for global variable (const and non-const one). First, we have global variable in one source file a.cpp:

```
int g_x{ 3 };           // internal by default
extern const int g_y { 4 }; // okie
extern constexpr int g_z{3}; // can, but useless
```

We use them in the main.cpp by their declarations:

```
#include <iostream>

int g_x;           // internal by default
extern const int g_y; // okie
extern constexpr int g_z; // can, but useless
```

```
int main()
{
    std::cout << g_x << '\n'; // prints 2

    return 0;
}
```

⚠ Do not use `extern` keyword if we define an uninitialized non-const global variable. C++ will think it is a declaration.

We should be clear about **file scope** and **global scope**: all global variable can be used within the file that defines them; if it can be seen by other files too, it has global scope (with proper declarations).

7.8 Why (non-const) global variables are evil

- Their values can be changed by any function that is called => Make program state is *unpredictable*.
- A function that utilizes nothing but its parameters and has no side effects is *perfectly modular*.
- Modularity helps both in understanding what a program does, as well as reusability.
- Global variables make the program less modular.

💡 Use local variables instead of global variables whenever possible.

7.9 Sharing global constants across multiple files

1. Create a .cpp file for a constants namespace includes all the const global variables.
2. Create a header file for that namespace
3. Include the header file in other files using the constants

This solution has downside. The compiler does not evaluate the const global variables at compile-time because it only recognizes the declarations. There may impact the performance maximization. To have a better solution, C++17 introduce a new keyword `inline` that allow an object can be defined in different files without violating the *one definition rule*.

```
#> In constant.cpp

#include "constant.h"
```

```

namespace constants
{
    extern const double pi{ 3.14159 };
    extern const double avogadro{ 6.0221413e23 };
    extern const double myGravity{ 9.2 };
}

#> In constant.h

#ifndef CONSTANT_H
#define CONSTANT_H

namespace constants
{
    extern const double pi;
    extern const double avogadro;
    extern const double myGravity;
}

#endif // !CONSTANT_H

#> In main.Cpp

#include "constant.h"
#include <iostream>

int main()
{
    std::cout << "Enter a radius: ";
    int radius{};
    std::cin >> radius;

    std::cout << "The circumference is " << 2.0 * radius * constants::pi << '\n';

    return 0;
}

```

7.9.1 Global constants as inline variables

In C++, the term `inline` has evolved to mean “multiple definitions are allowed”. An **inline variable** is one that is allowed to be defined in multiple files without violating one definition rule.

Inline variables have **two primary restrictions** that must be obeyed:

1. All definitions of the inline variable must be identical.
2. The inline variable definitions (not a forward declaration) must be present **in any file** that uses the variable.

We see the differences with the sharing constant global variables mentioned at the session's beginning:

- For inline const var, we do not need to have separate .cpp file for header because we define (not only declare) all in the header file.

```
# In constants.h

#ifndef CONSTANTS_H
#define CONSTANTS_H
namespace constants
{
    inline constexpr double pi{3.14159};
    inline constexpr double avogadro{6.0221413e23};
}

#endif

# In main.cpp

#include "constants.h"

#include <iostream>

int main()
{
    std::cout << "Enter a radius: ";
    int radius{};
    std::cin >> radius;

    std::cout << "The circumference is: " << 2.0 * radius * constants::pi << '\n';

    return 0;
}
```

💡 If you need global constants and your compiler is C++17 capable, prefer defining inline constexpr global variables in a header file.

7.10 Static local variables

Local variable, by its definition, has the scope within its block or *automatic duration*. When we want a local variable exists in whole the program, we use **static local variables** by adding `static` before local variable's definition.

Let have the examples:

```
#include <iostream>

void incrementAndPrint()
{
    int value{ 1 };
    ++value;
    std::cout << value << '\n';
}

int main()
{
    incrementAndPrint();
    incrementAndPrint();
    incrementAndPrint();

    return 0;
}

#> 2
#> 2
#> 2
```

Now, we modify for static local variable:

```
#label: static-local-var

#include <iostream>

void incrementAndPrint()
```

```

{
    static int value{ 1 };
    ++value;
    std::cout << value << '\n';
}


int main()
{
    incrementAndPrint();
    incrementAndPrint();
    incrementAndPrint();

    return 0;
}
#> 2
#> 3
#> 4

```

We use static local variable to keep track of variable like IDs for object.

7.10.1 Static local constants

 Avoid static local variables unless the variable never needs to be reset

7.11 Using declarations and using directives

7.12 Inline function

Codes of called function are expanded in caller. Modern compiler decides this.

7.13 Constexpr and consteval functions

A **constexpr function**: can be evaluated at compile-time. Simply by supplying constexpr keyword in front of the return type.

```
#include <iostream>
```

```
constexpr int greater(int x, int y)
{
    return (x > y ? x : y);
}

int main()
{
    constexpr int x{ 5 };
    constexpr int y{ 6 };

    constexpr int g{ greater(x, y) };

    std::cout << g << " is greater!\n";

    return 0;
}
```

To be a constexpr function, a function has to:

- Have a constexpr return type
- Do not have any non-constexpr functions

Constexpr function can be evaluated at *runtime*.

7.14 Chapter summary

A **compound statement** or **block** is a group of statements is treated as a single statement by compiler.

8 Control flow and Error Handling

In C++, CPU run the program from the beginning of `main()` function to its end. This running follows a sequence called **execution path**. There are types of path:

- **Straight-line program**: take tge same path every time they are run
- C++ provides different **control flow statements**: allow us to change the normal execution path through the program. One example of the control flow is `if` statements.
- **Branching**: when control flow statement causes execution point to change to a non-sequential statement.

Figure Figure. 8.1 below show the categories of **flow control statement**:

Category	Meaning
Conditional statements	Conditional statements cause a sequence of code to execute only if some condition
Jumps	Jumps tell the CPU to start executing the statements at some other location.
Function calls	Function calls are jumps to some other location and back.
Loops	Loops tell the program to repeatedly execute some sequence of code zero or more condition is met.
Halts	Halts tell the program to quit running.
Exceptions	Exceptions are a special kind of flow control structure designed for error handling.

Figure 8.1: Categories of flow control statements

8.0.1 If statements and blocks

C++ have two kinds of conditional statement: `if` statement and `switch` statement.

We already worked with `if` statement in previous chapters, one thing to be noted here is that the author recommends to use the `{}` for the statements, even if we just have single statement after `if` and/or `else`.

8.0.2 Common `if` statement problem

1. `if` statements within other `if` statements:

```
#include <iostream>

int main()
{
    std::cout << "Enter an integer: ";

    int x{};
    std::cin >> x;

    if (x >= 0)
        if (x <= 20)
            std::cout << x << " is between 0 and 20\n";

    return 0;
}
```

The ambiguity arises when there is an `else` statement:

```
#include <iostream>

int main()
{
    std::cout << "Enter an integer: ";

    int x{};
    std::cin >> x;

    if (x >= 0)
        if (x <= 20)
            std::cout << x << " is between 0 and 20\n";

    else std::cout << x << " is negative\n"; // else for which if?
```

```

    return 0;
}

```

C++ understands else for the last unmatched if statement. We should use the **block** for being clear:

```

#include <iostream>

int main()
{
    std::cout << "Enter an integer: ";

    int x{};
    std::cin >> x;

    if (x >= 0)
    {
        if (x <= 20)
            std::cout << x << " is between 0 and 20\n";
        else std::cout << x << " is greater than 20\n"; #// else for which if?
    }
    else
        std::cout << x << " is negative\n";

    return 0;
}

```

Nested if statement can be flattened by either restructuring the logic or by using logical operators:

```

#include <iostream>

int main()
{
    std::cout << "Enter an integer: ";
    int x{};
    std::cin >> x;

    if (x < 0)
        std::cout << x << " is negative\n";
    else if (x <= 20)
        std::cout << x << " is between 0 and 20\n";
    else

```

```
        std::cout << x << " is greater than 20";

    return 0;
}
```

8.0.3 Switch statement basics

The idea behind a **switch statement** is that: an expression (condition) is evaluated to produce a value. If the expression's value is equal to the value after any of case labels, the statements after the matching case label are executed. If there does not have any match, default label are executed instead.



- One restriction: condition has to be in **integral type**.
- The case follows by a constant expression.
- If matching, first statements after case label start and then continues sequentially.

Here is an example:

```
#include <iostream>

void printDigitName(int x)
{
    switch (x)
    {
        case 1:
            std::cout << "One";
            return;
        case 2:
            std::cout << "Two";
            return;
        case 3:
            std::cout << "Three";
            return;
        default:
            std::cout << "Unknown";
            break;
    }
}
```

```

int main()
{
    printDigitName(2);
    std::cout << '\n';

    return 0;
}

```

Using the return in switch, the program ends when there is a match or at default. We can use break to tell the program stop at the point matching or default and jump to the next statement next to switch.

```

#include <iostream>

void printDigitName(int x)
{
    switch (x)
    {
        case 1:
            std::cout << "One";
            break;
        case 2:
            std::cout << "Two";
            break;
        case 3:
            std::cout << "Three";
            break;
        default:
            std::cout << "Unknown";
            break;
    }

    std::cout << "\nOut of switch!\n";
}

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

    return 0;
}

```

What does the execution of switch when we miss return or break statement? It will continue to the next case and have unexpected result.

If we intentionally omit the end statement after the case, we can use the attribute `[[fallthrough]]`:

```
#include <iostream>

void printDigitName(int x)
{
    switch (x)
    {
        case 1:
            std::cout << "One";
            break;
        case 2:
            std::cout << "Two";
            [[fallthrough]];
        case 3:
            std::cout << "Three";
            break;
        default:
            std::cout << "Unknown";
            break;
    }

    std::cout << "\nOut of switch!\n";
}

int main()
{
    printDigitName(2);
    std::cout << '\n';

    return 0;
}
```

Sequential case labels: we can use switch statements by placing multiple case labels in sequence:

```
bool isVowel(char c)
{
    switch (c)
    {
```

```

        case 'a': // if c is 'a'
        case 'e': // or if c is 'e'
        case 'i': // or if c is 'i'
        case 'o': // or if c is 'o'
        case 'u': // or if c is 'u'
        case 'A': // or if c is 'A'
        case 'E': // or if c is 'E'
        case 'I': // or if c is 'I'
        case 'O': // or if c is 'O'
        case 'U': // or if c is 'U'
            return true;
        default:
            return false;
    }
}

```

If defining variables used in a case statement, do so in a block inside the case.

8.1 Goto statements

This is a type of unconditional jump. In C++, we use **goto statement** and the spot to jump by using **statement label**.

Here is an example:

```

#include <iostream>
#include <cmath>

int main()
{
    double x{};
    int tryTime{ 0 };

tryHere:
    std::cout << "Enter a non-negative number: ";
    std::cin >> x;

    if (x < 0.0 && tryTime < 5)
    {
        ++tryTime;
        std::cout << "You entered a negative number.\n";
    }
}

```


```

        goto tryHere;
    }

    if (tryTime < 5)
        std::cout << "The square root of " << x << " is " << std::sqrt(x) << '\n';
    else
        std::cout << "You cannot enter more than 5 times. Closing program.\n";

    return 0;
}

```

 Avoid using goto statement

8.2 Introduction to loops and while statements

Loops are control flow statements that allow a group of code execute repeatedly *until meeting defined condition*.

Here is an program that print integer number to 10:

```

#include <iostream>

int main()
{
    int count{1};
    while (count <= 100)
    {
        std::cout << count << ' ';
        ++count;
    }

    std::cout << "done!\n";

    return 0;
}

```

If the condition in while is always true, the program run infinitely or we have **infinite loop**.

Intentional infinite loop: we can add the break, return, or goto statement to end the infinite loop.

Loop variable:: to count how many times a loop has executed. This is also called **counter**

Iteration: each time loop executes. We write a program to count from 1 to 50, each line have 10 numbers:

```
#include <iostream>

int main()
{
    int count{1};
    while(count <= 50)
    {
        if (count < 10)
        {
            std::cout << '0';
        }

        std::cout << count << ' ';

        if (count % 10 == 0)
        {
            std::cout << '\n';
        }

        ++count;
    }

    return 0;
}
```

8.2.1 Nested loop

A loop can be inside other loop. The inner loop can use outer loop's counter. Here is an example:

```
#include <iostream>

int main()
{
    int outer{ 1 };
    while (outer <= 10)
    {
        int inner{ 1 };
        while (inner <= outer)
```



```

        {
            std::cout << inner << ' ';
            ++inner;
        }
        std::cout << '\n';
        ++outer;
    }

    return 0;
}

```

In the exercise, we write a small program to print the characters from 'a' to 'z' and its ASCII code. From this, we need to recall the function `static_cast<type>(variable)` to transform characters to its integer:

```

#include<iostream>

int main()
{
    char myChar{ 'a' };
    while (myChar <= 'z')
    {
        std::cout << myChar << ' ' << static_cast<int>(myChar) << '\n';
        ++myChar;
    }

    return 0;
}

```

The next exercise ask us to print an inverse triangle:

```

#|label: while-inverse-triangle

#include <iostream>

int main()
{
    int outer{ 5 };
    while (outer >= 1)
    {
        int inner{ outer };
        while ((inner >= 1))

```

```

    {
        std::cout << inner-- << ' ';
        #/--inner;
    }

    std::cout << '\n';
    --outer;
}

return 0;
}

```

The most exercise is to create a right-handside triangle:

```

#include <iostream>

int main()
{
    int outer{ 1 };
    while (outer <= 5)
    {
        int inner{ 5 };

        while (inner >= 1)
        {
            if (inner <= outer)
                std::cout << inner << ' ';
            else
                std::cout << " ";
            --inner;
        }

        std::cout << '\n';
        outer++;
    }

    return 0;
}

```

8.3 Do while statement

This is just like while loop but the statement always executes at least once. After the statement has been executed, the do-while loop checks the condition.

```
#include <iostream>

int main()
{
    // selection must be declared outside of the do-while so we can use it later
    int selection{};

    do
    {
        std::cout << "Please make a selection: \n";
        std::cout << "1) Addition\n";
        std::cout << "2) Subtraction\n";
        std::cout << "3) Multiplication\n";
        std::cout << "4) Division\n";
        std::cin >> selection;
    }
    while (selection != 1 && selection != 2 &&
           selection != 3 && selection != 4);

    // do something with selection here
    // such as a switch statement

    std::cout << "You selected option #" << selection << '\n';

    return 0;
}
```

The author recommend favor while loops over do-while when given equal choice.

8.4 For statements

In this session, we learn about the classic for statements with the form: `for (init-statement; condition; end-expression) statements`.

We can easily link to the while statement:

```

init_statement;
while(condition)
{
    statement;
    end_expression;
}

```

We can write a very simple for loop:

```

int main()
{
    for (int count{ 1 }; count <= 10; ++count)
        std::cout << count << ' ';

    std::cout << '\n';

    return 0;
}

```

We can easily change to while statement:

```

int main()
{
    {
        int count{1};
        while (count <= 10)
        {
            std::cout << count << ' ';
            ++count;
        }
    }

    std::cout << '\n';
}

```

We can define multiple counters:

```

#include "io.h"
#include <iostream>

int main()
{

```

```

    for (int x{ 0 }, y{ 9 }; x < 10; ++x, --y)
        std::cout << x << ' ' << y << '\n';

    return 0;
}

```

Loop can nest in other loop:

```

void charNumber()
{
    for (char c{ 'a' }; c <= 'h'; ++c)
    {
        std::cout << c;

        for (int i{ 0 }; i < 4; i++)
        {
            std::cout << i;
        }

        std::cout << '\n';
    }
}

```

i Best practice

- For statements are the mostly important used loop in C++.
- Prefer for loop over while loop when there is an obvious loop variable.
- Prefer while loop over for loops when there is no obvious loop variable.

8.5 Break and continue

break statement terminates the switch or loop, and execution continue at the *first statement* beyond the switch or loop.

return statement *terminate the entire function that the loop is within*, and execution continues at point where the function was called.

8.5.1 Continue

Continue statement tells the program end current iteration, and continue next iteration.

The author advise to use the break, continue statement or *early return* when they simplify the loop logic.

8.6 Halts

The last category of flow control statement is **halt**: terminates the program.

8.6.1 The std::exit() function

It terminates the program normally. The term **normal termination** means that the program exits in an expected way; it does not imply about whether the program was successful or not.

std::exit() cleans up some static storage objects, returned back to OS.

The program does not execute any statements after std::exit().

std::exit() does not clean up local variables in the current function or up in the call stack.

Function std::atexit() take the arguments as functions to remember to do the clean up when we call std::exit().

8.6.2 std::abort()

This function ends the program *abnormally*. It means that the program has some kind of unusual runtime error and could not continue to run.

```
#include <cstdlib> // for std::abort()
#include <iostream>

int main()
{
    std::cout << 1 << '\n';
    std::abort();

    // The following statements never execute
    std::cout << 2 << '\n';

    return 0;
}
```

```
}
```

It does not clean up anything.

💡 Only use a halt if there is no safe way to return normally from the main function.

8.7 Introduction to testing the code

8.7.1 Test your programs in small pieces

We should write small functions (or classes), and then compile and test immediately. This test is called **unit testing**; each unit test assures a correct-particular behavior of that unit.

These are some methods to test your unit code:

Informal testing

You write your function and run it with some inputs to see whether it runs as expected or not.

We have a function that checks whether a character is a lower Vowel, then we add some character to see that it works well

```
#include "io.h"
#include <iomanip>
#include <iostream>

int main()
{
    std::cout << std::boolalpha;
    std::cout << isVowel('0') << '\n';

    return 0;
}
```

Preserving your tests

Instead of testing informally, we can write a program to test the unit function.

```

void testVowel()
{
    std::cout << "Choose 'a': " << isVowel('a') << '\n';
    std::cout << "Choose 'b': " << isVowel('b') << '\n';
}

```

8.8 Common semantic errors in C++

Semantic errors means that the codes do not do what we intended, and it leads to undefined behavior. There are some common semantic errors:

- Conditional logic errors: errors occurs when the programmer incorrectly codes the logic of a conditional statement or loop condition.
- Infinite loop
- Off-by-one errors: counter are not correctly condition
- Incorrect operator precedence
- Integer division
- Accidental null statements
- Not using compound statement when one is required

8.9 Detecting and handling errors

Most errors occur due to faulty assumptions made by the programmer and/or lack of proper error detection/handling:

- Assume about return value
- Assume that user give correct input
- Assume that function is called correctly

Thus, we need to learn about error handling strategies (what to do when things go wrong) inside a function.

Functions may fail for many reasons and there is no best way to handle an error. There are 4 general strategies that can be used:

8.9.1 Handling the error within the function

We correct the error in the same function which the error occurred so that the error can be contained and corrected without impacting any code outside function:


```

void printDivision(int numerator, int demonator)
{
    if (demonator != 0)
        std::cout << static_cast<double>(numerator) / demonator << '\n';
    else
        std::cerr << "Error: Could not divide by zero\n";
}

```

i The handling print the error message out, and then the program continue to run the next statements in the caller.
When we added a halt, it stops entire program.

8.9.2 Passing error back to the caller

As the above handling, the called function does not return anything to the calling function. If we want to pass the error to caller function, we should create a handling that return a value so that the caller function knows whether the called function runs correctly or not.

8.9.3 Fatal errors

In some case, the error is severe that we have to stop the program (*non-recoverable*), we can use the halt statement such as `std::exit()`:

```

void printDivision(int numerator, int demonator)
{
    if (demonator != 0)
        std::cout << static_cast<double>(numerator) / demonator << '\n';
    else
    {
        std::cerr << "Error: Could not divide by zero\n";
        std::exit(1);
    }
}

```

8.10 std::cin and handling invalid input

A **robust** program anticipate the misuse from users and provide way to handle those program at the first place.

`std::cin`, buffers, and extraction

`std::cin` let user enter a value. This value is then stored temporarily inside of `std::cin` and waits for extracting to a variable. They call it **buffer**. The operator `>>` is an extraction to get the value in buffer into the variable. Normally, a value or a character is taken out, if there is still information in the buffer, that information is still there for the next extraction.

For example, if there is "5a" in buffer and we extract to variable `x`, then `x = '5'` and 'a' is in the buffer for the next extraction.

We write a program to illustrate the following error cases:

```
#include <iostream>

double getDouble()
{
    std::cout << "Enter a double value: ";
    double x{};
    std::cin >> x;
    return x;
}

char getOperator()
{
    std::cout << "Enter one of the following: +, -, *, or /: ";
    char op{};
    std::cin >> op;
    return op;
}

void printResult(double x, char operation, double y)
{
    switch (operation)
    {
        case '+':
            std::cout << x << " + " << y << " is " << x + y << '\n';
            break;
        case '-':
            std::cout << x << " - " << y << " is " << x - y << '\n';
            break;
        case '*':
            std::cout << x << " * " << y << " is " << x * y << '\n';
            break;
    }
}
```

```

        case '/':
            std::cout << x << " / " << y << " is " << x / y << '\n';
            break;
    }
}

int main()
{
    double x{ getDouble() };
    char operation{ getOperator() };
    double y{ getDouble() };

    printResult(x, operation, y);

    return 0;
}

```

Error case 1: Input extraction succeeds but the input is meaningless to the program

For example, users enter $x = 5$, $y = 2$ and operator = 'k'. These inputs are attracted successfully to variable but we cannot operate the function.

We solve this case by doing input validation:

```

char getOperator()
{
    while (true)
    {
        std::cout << "Enter one of the following: +, -, *, or /: ";
        char op{};
        std::cin >> op;

        switch (op)
        {
            case '+':
            case '-':
            case '*':
            case '/':
                return op;
            default:
                std::cerr << "Error: Operator is not valid. Please try again.\n";
        }
    }
}

```

```

    }
}
}

```

Error case 2: Extraction succeeds but with extraneous input

For example, user enters 5*7, the programs run and return 35 but does not lead user continue enter the input because *7\n is stored in buffer for the following operator>>.

To solve this error, we have to tell the program **ignore** all the following character until the next \n. In C++, we use the function `std::cin.ignore(num_char_to_ignore, \n)` or better is `std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n')`

```

void ignoreLine()
{
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
}

```

Then call the function in get input function:

```

double getDouble()
{
    std::cout << "Enter a double: ";
    double x{};
    std::cin >> x;
    ignoreLine();

    return x;
}

```

Error case 3: Extraction fails

This case happens when we enter a double value by a value with different type eg.: a. The operator>> cannot cover it to a double, so it let a in the buffer and goes into *fail mode*. Then, further requests for input extraction will silently fail. Thus, the output prompt still prints, but the input prompt is skipped, and we get stuck in infinite loop.

We solve the problem by:

- Test whether the `std::cin` is fail: `!std::cin`
- If it is fail, put it back to *normal* state: `std::cin.clear()`

- Flush the buffer so we can enter the new value: `ignoreLine()`

```
if (!std::cin)
{
    std::cin.clear();    ///put back to normal mode
    ignoreLine();        ///and remove bad input
}
```

Error case 4: Extraction succeeds but the user overflows a numeric value

8.11 Assert and `static_assert`

Return to the program:

```
void printDivision(int numerator, int demonator)
{
    if (demonator != 0)
        std::cout << static_cast<double>(numerator) / demonator << '\n';
    else
    {
        std::cerr << "Error: Could not divide by zero\n";
    }
}
```

The program checks whether we divide a number by a zero. This is a semantic error and will cause the program to crash down.

We link to the session [Section 8.6](#) for ideas on `std::exit()` and `std::abort()`. If we skip the offending statements,

9 Summary

In summary, this book has no content whatsoever.

Markdown allows you to write using an easy-to-read, easy-to-write plain text format.

1 + 1

References

Knuth, Donald E. 1984. “Literate Programming.” *Comput. J.* 27 (2): 97–111. <https://doi.org/10.1093/comjnl/27.2.97>.

Index

Markdown, [107](#)