Chapter 1: Basic Ideas 1

- C++ is one of the fastest languages and can be used to do anything, almost.

Modern C++ 1

Just about any kind of program can be written in C++, from device drivers to operating systems and from payroll and administrative programs to games. Major operating systems, browsers, office suites, email clients, multimedia players, database systems

Above all else, C++ is perhaps best suited for applications where performance matters, such as applications that have to process large amounts of data, modern games with high-end graphics, or apps that target embedded or mobile devices.

- C++11, C++14, C++17 are just C++ with standardization. applying an implicit set of guidelines and best practices, all designed to make C++ programming easier, less error-prone, and more productive.

Standard Libraries 2 (*Using Pre-written Code*)

-a huge collection of routines and definitions that provide functionality that is required by many programs. Examples are numerical calculations, string processing, sorting and searching, organizing and managing data, and input and output. Also uses StandardTemplateLibrary, a subset of C++ Standard Library.

C++ Program Concepts 3

Source Files and Header Files 3

The file extension, .cpp, indicates that this is a C++ source file.
.cc, .cxx, or .c++ are sometimes used to identify a C++ source file.

Comments and Whitespace 4

// Comments & /* Comments */ & WHITESPACE is any sequence of spaces, tabs, newlines, or form feed characters.

Preprocessing Directives and Standard Library Header 4

`#include <iostream>` <-**Preprocessing Directive** = cause the source code to be modified in some way before it is compiled to executable form. This preprocessing directive adds the contents of the Standard Library header file with the name iostream to this source file, Ex1_01.cpp. The header file contents are inserted in place of the #include directive.

headers, contain definitions to be used in a source file. iostream contains definitions that are needed to perform input from the keyboard and text output to the screen using Standard Library routines. In particular, it defines std::cout and std::endl among many other things, OTHERWISE the compiler would not know what std::cout or std::endl is.

Functions 5

- **function** is a named block of code that carries out a well-defined operation such as “read the input data” or “calculate the average value” or “output the results.”
- You execute, or **call**, a function in a program using its name.
- All the executable code in a program appears within functions.
- There must be one function with the name **main**, and execution always starts automatically with this function. The main() function usually calls other functions, which in turn can call other functions, and so on

ADVANTAGES

✓ A program that is broken down into discrete functions is easier to develop and test.

✓ You can reuse a function in several different places in a program, which makes the program smaller than if you coded the operation in each place that it is needed.

✓ You can often reuse a function in many different programs, thus saving time and effort.

✓ Large programs are typically developed by a team of programmers. Each team member is responsible for programming a set of functions that are a well-defined subset of the whole program. Without a functional structure, this would be impractical.

- int main(); is the **Function Header**.

Statements 5

-statement is a basic unit in a C++ program. You can enclose several statements between a pair of curly braces, { }, in which case they're referred to as a statement block. statement block is also referred to as a compound statement.

nesting-blocks can be placed inside other blocks. Blocks can be nested, one within another, to any depth.

Data Input and Output 6

- Input and output are performed using **streams** in C++.
- A stream is an abstract representation of a source of data or a data sink.
- **cout** and **cin** -The standard output and input streams in C++.
- Begins output-- **std::cout** and Ends Output -- **std::endl** ----- << **Insertion operator**(Writes) ---- >> **Extractor operator**(Reads)
- Whatever appears to the right of each << is transferred to cout.
- Inserting endl to std::cout causes a new line to be written to the stream and the output buffer to be flushed.
- Flushing the output buffer ensures that the output appears immediately.

return Statements 7

- last statement in main() is a **return** statement.
- A return statement ends a function and returns control to where the function was called.
- This particular return statement returns 0 to the operating system.
- Returning 0 to the operating system indicates that the program ended normally.
- You can return nonzero values such as 1, 2, etc., to indicate different abnormal end conditions.
- main() is the only function for which omitting return is equivalent to returning zero.
- Any other function with return type int always has to end with an explicit return statement—the compiler shall never presume to know which value an arbitrary function should return by default.

Namespaces 7

- **namespace** is a sort of family name that prefixes all the names declared within the namespace.
- Almost all names from the Standard Library are prefixed with **std**

Names and Keywords 8

Lots of things need names in a program, and there are **PRECISE RULES** for defining names:

- A name can be any sequence of upper or lowercase letters A to Z or a to z, the digits 0 to 9, and the underscore character, _.
- A name must begin with either a letter or an underscore.
- Names are case sensitive.

Other names that you are **NOT** supposed to use include the following:

- Names that begin with two consecutive underscores
- Names that begin with an underscore followed by an uppercase letter
- Within the global namespace: all names that begin with an underscore

-the common denominator with these reserved names is that they all start with an underscore. Thus, our advice is this:

■ Tip *Do not use names that start with an underscore.*

Classes and Objects 9

-class is a block of code that defines a data type. A class has a name that is the name for the type.

-An item of data of a class type is referred to as an **object**.

-EX: School software. Class = Students / Object = age, gender, etc.

- Objects are mostly intuitive in use because they're mostly designed to behave like real-life entities (although some do model more abstract concepts, such as input or output streams, or low-level C++ constructs, such as data arrays and character sequences).

Templates 9

template a recipe you created, used by compiler, generates code automatically for a class / function customized for a particular type or types.

The compiler uses a **class template** to generate one or more of a family of classes.

It uses a **function template** to generate functions. Each template has a name that you use when you want the compiler to create an instance of it. The Standard Library uses templates extensively.

Code Appearance and Programming Style 9

- Use tabs and/or spaces to indent program statements in a manner that provides visual cues to their logic
- Arrange matching braces that define program blocks in a consistent way so the relationships between the blocks are apparent
- Spread a single statement over two or more lines when that will improve the readability of your program.

--- choose clear, descriptive names for all your variables, functions, and types.

Creating an Executable 10

- executable module. three-step process.
- the **preprocessor** processes all preprocessing directives. One of its key tasks is to, at least in principle, copy the entire contents of all #included headers into your .cpp files.
- second, **compiler** processes each .cpp file to produce an **object file** that contains the machine code equivalent of the source file.
- third step, the **linker** combines the object files for a program into a file containing the complete executable program.

FIXING ERRORS / PERFECTING CODE

1. Look for Typographical / Syntax Errors = No Answer Production
2. Look for Logical Errors = Wrong Answer Production
3. Repeat
4. Test / Try to create errors

Procedural and Object-Oriented Programming (Methodology) 12

Procedural programming (Methodology), focus on the process that your program must implement to solve the problem.

- You create a clear, high-level definition of the overall process that your program will implement.
- Segment the process into workable units of computation that are, sorta, self-contained. These will usually correspond to functions.
- You code the functions in terms of processing basic types of data: numerical data, single characters, and character strings.

Object-Oriented Programming (Methodology)- ***SUPERIOR METHODOLOGY:*** Easy to Read / Maintain. Longer writing process. Solving problems with entities that relates to a problem, Composed of: numbers and characters.

- From the problem specification, you determine what types of **objects** the problem is concerned with.

If your program deals with baseball players, you're likely to identify BaseballPlayer as one of the types of data your program will work with. If your program is an accounting package, you may want to define objects of type Account and type Transaction.

- You also identify the set of **operations** that the program will need to carry out on each type of object. This will result in a set of application-specific data types that you will use in writing your program.
- You produce a detailed design for each of the new data types that your problem requires, including the operations that can be carried out with each object type.
- You express the logic of the program in terms of the new data types you've defined and the kinds of operations they allow.

Representing Numbers 13

Binary / Hexadecimal / Negative Binary / Octal Values / Floating-Point Number Representation
Base 2 / Base 16 / Sign Magnitude /Base 8-Obsolete /

Binary Numbers 13- Binary #'s = Bits

Hexadecimal Numbers 14- Base 16- 0-9/A-F = 8 binary digits(Byte) = 2 Hex Digits 0011 1101 = 3D

Negative Binary Numbers 16

Integers that can be both positive and negative are referred to as **signed integers**

computer's memory is generally composed of 8-bit bytes

SIGN MAGNITUDE= 6 represented as 00000110, -6 represented as 10000110. Changing +6 to –6 = Flipping the sign bit from 0 to 1. Process is called **2's Complement.**

Octal Values 17- Base 8 – 0-7 = Obsolete method

Bi-Endian and Little-Endian Systems 18

--Integers are stored in memory as binary values in a contiguous sequence of bytes, commonly groups of 2, 4, 8, or 16 bytes.

---**Little-Endian**= Largest to Left(Highest Address) / Smallest -> Right (Rightmost Byte)

Byte address: 00 01 02 03

Data bits: 00000001 00000010 00000100 00000000

--More Efficient calculations / Simpler Hardware

--ONLY IMPORTANT WHEN: processing binary data that comes from another machine. You need to know the endianness. Binary data is written to a file or transmitted over a network as a sequence of bytes.

Floating-Point Numbers 19

- **Floating Point Numbers** = mechanism nearly all computers support for handling fractional numbers. EX: 3.14 or -0.05
- Fractional Numbers and Very Large Numbers (20+ Digits). The Decimal Point "Floats" EX. 3,654,800 = 3.6548E02 in Floating-Point
- A Close approximation of a number (Give or take a couple digits)
- Mind the Digit Value when adding/subtracting EX: 3.650000E+06 + 1.230000E-04 = 3.650000E+06(Again)
- Almost equal numbers are also chancey and can cause-- **catastrophic cancellation**.
- floating-point numbers is limited and that the order and nature of arithmetic operations you perform with them can have a significant impact on the accuracy of your results.
- most common floating-point numbers representations are the so-called **single precision(**7 Digit Accuracy**)** (1 sign bit, 8 bits for the exponent, and 23 for the mantissa, adding up to 32 bits in total) and **double precision(**16 Digit Accuracy**)** (1 + 11 + 52 = 64 bits) floating-point numbers.

Representing Characters 20

Each character is assigned a unique integer value called its **code** or **code point**. Numbers/Chars are all Binary.

ASCII Codes 21 (American Standard Code for Information Interchange)

FOR KEYBOARD: 7-bit code, so there are 128 different code values. ASCII values 0 to 31 represent various nonprinting control characters such as carriage return (code 15) and line feed (code 12). Code values 65 to 90 inclusive are the uppercase letters A to Z, and 97 to 122 correspond to lowercase a to z. Values from 128 to 255 are variable.

Universal Character Set (UCS) emerged with codes with up to 32 bits. Potential for hundreds of millions of unique code values.

UCS and Unicode 21

A **code point** is an integer; an **encoding** specifies a way of representing a given code point as a series of bytes or words.

C++ Source Characters 22

You write C++ statements using a **basic source character set**. The basic source character set consists of the following characters:

- The letters a to z and A to Z
- The digits 0 to 9
- The whitespace characters space, horizontal tab, vertical tab, form feed, and newline
- The characters _ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = , \ " '

Escape Sequences 22

Using **character constants** such as a single character or a character string in a program, certain characters are problematic.

Escape sequence is an indirect way of specifying a Special character, and it always begins with a backslash.

Escape Sequence	Control Character
\n	New Line
\t	New Tab
\v	Vertical Tab
\b	Backspace
\r	Carriage Return
\f	Form Feed
\a	Alert/Bell
	PROBLEM CHARACTER
\\	Backslash
\’	Single Quote
\”	Double Quote

```
std::cout << "\"Least \'said\' \\\n\t\tsoonest \'mended\'.\"" << std::endl;
std::cout << "" << std::endl; ------IS THE MAIN COMMAND
```

Double quote characters are **delimiters** that identify the beginning and end of the string literal; they aren’t part of the string.

A backslash in a string literal always indicates the start of an escape sequence

If you’re no fan of escape sequences, Chapter 7 will introduce a possible alternative to them called **raw string literals**.

Summary 24

- A C++ program consists of one or more functions, one of which is called **main().** Execution always starts with main().
- The executable part of a function is made up of statements contained between braces. **“{}”**
- A pair of **curly braces** is used to enclose a statement block.
- A statement is terminated by a **semicolon.**
- **Keywords**- reserved words with specific meanings in C++. No entity in a program can have a name that is a keyword.
- A C++ program will be contained in one or more files.
  Source files contain the executable code, and **header files** contain definitions used by the executable code.
- The source files that contain the code defining functions typically have the extension **.cpp**
- Header files that contain definitions that are used by a source file typically have the extension **.h**
- **Preprocessor directives** specify operations to be performed on the code in a file. All preprocessor directives execute before the code in a file is **compiled**.
- The contents of a header file are added into a source file by an “#include” preprocessor directive.
- The **Standard Library** provides an extensive range of capabilities that supports and extends the C++ language.
- Access to Standard Library functions / definitions is enabled by including Standard Library header files in a source file.
- **Input** and **output** are performed using **streams** and involve the use of the **insertion** and **extraction operators**, **<<** and **>>** . **std::cin** is a standard input stream that corresponds to the keyboard. **std::cout** is a standard output stream for writing text to the screen. Both are defined in the iostream Standard Library header.
- **Object-oriented programming** involves defining new data types that are specific to your problem. Once you’ve defined the data types that you need, a program can be written in terms of the new data types.
- **Unicode** defines unique **integer code** values that represent characters for virtually all of the languages in the world as well as many specialized character sets. Code values are referred to as code points. Unicode also defines how these code points may be encoded as byte sequences.

■Chapter 2: Introducing Fundamental Types of Data 27

Variables, Data, and Data Types 27

- **Variable** is a named piece of memory that you define. Each variable stores data only of a particular type.
- Every variable has a **type** that defines the kind of data it can store.
- Each fundamental type is identified by a unique type name that consists of one or more **keywords**.
- Keywords are reserved words in C++ that you cannot use for anything else.
- Numerical values fall into two categories: **integers**, which are whole numbers, and **floating-point values**, which can be nonintegral.

Defining Integer Variables 28

- The braces enclosing the initial value are called a **braced initializer,** can have Several Values.
- A conversion to a type with a more limited range of values is called a **narrowing conversion**.
- **Functional notation** looks like this: int orange_count(5); int total_fruit(apple_count + orange_count);
- alternative is the **so-called assignment notation**: int orange_count = 5; int total_fruit = apple_count + orange_count;
- **braced initializer syntax** was introduced in C++11 specifically to standardize initialization. Its main advantage is that it enables you to initialize just about everything in the same way—which is why it is also commonly referred to as **uniform initialization**.

```
int banana_count(7.5);        // May compile without warning
int coconut_count = 5.3;      // May compile without warning
int papaya_count{0.3};        // At least a compiler warning, often an error
```

SIGNED INTEGER TYPES		
Type Name	**Typical Size (Bytes)**	**Range of Values**
(LETTERS) signed char	1	-128 to +127
(WHOLE NUMBERS) short short int signed short signed short int	2	-256 to +255
(WHOLE NUMBERS) int signed signed int	4	-2,147,483,648 to +2,147,483,647
(WHOLE NUMBERS) long long int signed long signed long int	4 or 8	Same as int or long long
(WHOLE NUMBERS) long long long long int signed long long signed long long int	8	9,223,372,036,854,775,808 to +9,223,372,036,854,775,807

Zero Initialization 31

```
int counter {0};        // counter starts at zero   =   int counter {}; // counter starts at zero
```

Zero initialization works for any fundamental type. For all fundamental numeric types, for instance, an empty braced initializer is always assumed to contain the number zero.

Defining Variables with Fixed Values 32

const keyword in the definition of a variable that must not be changed. Such variables are often referred to as **constants**.

example: *const unsigned toe_count {10};* // An unsigned integer with fixed value 10

■ Tip If nothing else, knowing which variables can and cannot change their values along the way makes your code easier to follow. So, we recommend you add the const specifier whenever applicable.

Integer Literals 32

Constant values of any kind, such as 42, 2.71828, 'Z', or "Mark Twain", are referred to as **literals**. These examples are, in sequence, an **integer literal**, a **floating-point literal**, a **character literal**, and a **string literal**.

Decimal Integer Literals 32

examples of decimal integers: -123L +123 123 22333 98U -1234LL 12345ULL

Unsigned integer literals have **u** or **U** appended. Literals of types long and type long long have L or LL appended, respectively, and if they are unsigned, they also have u or U appended

---

unsigned long age {99UL}; // 99ul or 99LU would be OK too
unsigned short price {10u}; // There is no specific literal type for short
long long distance {15'000'000LL}; // Common digit grouping of the number 15 million

---

■ Note While mostly optional, there are situations where you do need to add the correct literal suffixes, such as when you initialize a variable with type auto (as explained near the end of this chapter) or when calling overloaded functions with literal arguments

Hexadecimal Literals 33

unsigned int color {0x0f0d0e}; // Unsigned int hexadecimal constant - decimal 986,382 int mask {0XFF00FF00}; // Four bytes specified as FF, 00, FF, 00 unsigned long value {0xDEADlu}; // Unsigned long hexadecimal literal - decimal 57,005

Octal Literals 34 Out-of-date

Binary Literals 34

- binary integer literal as a sequence of binary digits (0 or 1) prefixed by either 0b or 0B.
- binary literal can have L or LL as a suffix to indicate it is type long or long long, and u or U if it is an unsigned literal.

Binary literals: 0B110101111 0b100100011U 0b1010L 0B110010101101 0b11111111
Decimal literals: 431 291U 10L 3245 255

Calculations with Integers 35

- An operation, addition or multiplication is defined by an **operator**—for addition and multiplication are + and *, respectively.
- The values that an operator acts upon are called **operands**, so in an expression such as 2*3, the operands are 2 and 3.
- Need 2 Operators - **binary operators**. Need 1 Operators - **unary operators**.

OPERATOR OPERATION

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (the remainder after division)

The modulus operator, %, complements the division operator in that it produces the remainder after integer division. It is defined such that, for all integers x and y, **(x / y) * y + (x % y) == x**

Compound Arithmetic Expressions 36

Parenthesis arithmetic comes first.

multiplication / division / modulus operations / addition and subtraction.
Nested Parenthesis: 2*(a + 3*(b + 4*(c + 5*d)))

Assignment Operations 37

This last line is an **assignment statement**, and the = is the **assignment operator**.

```
int a {}, b {}, c {5}, d{4};
a = b = c*c - d*d;
```

Variables can be overwritten as many times as you want, except constants, of course.

```
int y {5};
y = y + 1;
```

The op= Assignment Operators 40

```
feet = feet % feet_per_yard;
```

This statement could be written using an op= **assignment operator**. The op= assignment operators, or also **compound assignment operators**, are so called because they're composed of an operator and an assignment operator =. You could use one to write the previous statement as follows:

```
feet %= feet_per_yard;
```

```
x *= y + 1;      ----------      This is equivalent to the following:
x = x * (y + 1);
```

op= Assignment Operators			
Operation	**Operator**	**Operation**	**Operator**
Add	+=	Bitwise AND	&=
Subtract	-=	Bitwise OR	|=
Multiply	*=	Bitwise exclusive OR	^=
Division	/=	Shift Left	<<=
Modulus	%=	Shift Right	>>=

DECLARATIONS AND DIRECTIVES

You can eliminate the need to qualify a name with the namespace name in a source file with a ***using*** *declaration*.
With this declaration before the main() function definition, you can write cout instead of std::cout
You could include two using declarations at the beginning of Ex2_02.cpp and avoid the need to qualify cin and cout:

```
using std::cin;
using std::cout;
using std::endl;
```

The sizeof Operator 42

use the **sizeof operator** to obtain the number of bytes occupied by a type, by a variable, or by the result of an expression. Here are some examples of its use:

```
int height {74};
std::cout << "The height variable occupies " << sizeof height << " bytes." << std::endl;
std::cout << "Type \"long long\" occupies " << sizeof(long long) << " bytes." << std::endl;
std::cout << "The result of the expression height * height/2 occupies "
 << sizeof(height * height/2) << " bytes." << std::endl;
```

Incrementing and Decrementing Integers 42

- **increment operator** and the **decrement operator**, **++** and **--**

- The following three statements that modify **count** have exactly the same effect:

```
int count {5};
count = count + 1;
count += 1;
++count;
```

- ++ or -- operator before the variable is called the **prefix form.** Incrementation happens first.
- You can also place them after a variable, which is called the **postfix form**. Incrementation happens last.

Postfix Increment and Decrement Operations 43

Defining Floating-Point Variables 44

Data Type	Description	Example
float	Single precision floating-point values	2.00000000
double	Double precision floating-point values	2.0000235
long double	Double-extended precision floating-point values	

precision refers to the number of significant digits in the mantissa.

Type	Precision (Decimal Digits)	Range (+ or –)
float	7	$\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$
double	15 (nearly 16)	$\pm 2.22 \times 10^{-308}$ to $\pm 1.8 \times 10^{308}$
long double	18-19	$\pm 3.65 \times 10^{-4932}$ to $\pm 1.18 \times 10^{4932}$

```
float pi {3.1415926f}; // Ratio of circle circumference to diameter
double inches_to_mm {25.4};
long double root2 {1.4142135623730950488L}; // Square root of 2
```

Floating-Point Literals 45

- You can see from the code fragment in the previous section that float literals have f (or F) appended and long double literals have L (or l) appended. Floating-point literals without a suffix are of type double. A floating-point literal includes either a decimal point or an exponent, or both; a numeric literal with neither is an integer.
- An exponent is optional in a floating-point literal and represents a power of 10 that multiplies the value. An exponent must be prefixed with **e or E** and follows the value. Here are some floating-point literals that include an exponent:
- 5E3 (5000.0) 100.5E2 (10050.0) 2.5e-3 (0.0025) -0.1E-3L (-0.0001L) .345e1F (3.45F)

Floating-Point Calculations 46

```
const double pi {3.141592653589793};          // Circumference of a circle divided by its diameter
double a {0.2};          // Thickness of proper New York-style pizza (in inches)
double z {9};            // Radius of large New York-style pizza (in inches)
double volume {};        // Volume of pizza - to be calculated
volume = pi*z*z*a;
```

- The modulus operator, %, can't be used with floating-point operands, but all the other binary arithmetic
- operators that you have seen, +, -, *, and /, prefix and postfix increment and decrement operators, ++ and -- can be used

Pitfalls (Of Floating-Point Values)46

- Many decimal values don't convert exactly to binary floating-point values. The small errors that occur can easily be amplified in your calculations to produce large errors.

- Taking the difference between two nearly identical values will lose precision. If you take the difference between two values of type float that differ in the sixth significant digit, you'll produce a result that will have only one or two digits of accuracy. Resulting in catastrophic cancellation.
- Working with values that differ by several orders of magnitude can lead to errors. An elementary example of this is adding two values stored as type float with 7 digits of precision where one value is 108 times larger than the other. You can add the smaller value to the larger as many times as you like, and the larger value will be unchanged.

Invalid Floating-Point Results 47

-the result of division by zero is undefined.

-Binary mantissa of all zeroes and an exponent of all ones to represent +infinity or -infinity, depending on the sign. When you divide a positive nonzero value by zero, the result will be **+infinity**, and dividing a negative value by zero will result in **-infinity**.

-Another special floating-point value defined by this standard is called **not-a-number**, or **NaN**. This represents a result that isn't mathematically defined, such as when you divide zero by zero or infinity by infinity.

Operation	Result	Operation	Result
±value / 0	±infinity	0 / 0	NaN
±infinity ± value	±infinity	±infinity / ±infinity	NaN
±infinity * value	±infinity	infinity - infinity	NaN
±infinity / value	±infinity	infinity * 0	NaN

To check whether a given number is either infinity or NaN, you should use the **std::isinf()** and **std::isnan()** functions provided by the **#include<cmath>** header

Mathematical Functions 48

#include<cmath> ---→ std::FUNCTION

Standard Library header file defines a large selection of **trigonometric** & **numerical functions** that you can use in your programs.

FUNCTION	DESCRIPTION
abs(arg)	computes **Absolute Value** of an Argument. Unlike most cmath functions, abs() returns an integer type if the arg is integer.
ceil(arg)	(Searches for ceiling) computes from the **smallest floating-point value**. ceil <= arg EXAMPLE: std::ceil(2.5) produces 3.0 and std::ceil(-2.5) produces -2.0.
floor(arg)	(Searches for floor) computes from the **largest floating-point value**. floor >= arg EXAMPLE: std::floor(2.5) results in 2.0 and std::floor(-2.5) results in -3.0
exp(arg)	computes the value of **EArg**
log(arg)	Computes the **Natural Logarithm (to base e)** of arg.
log10(arg)	computes the **Logarithm to base 10** of arg
pow(arg1, arg2)	**Arg1^{Arg2}** . can be integer or floating-point types. std::pow(2, 3) is 8.0 , std::pow(1.5f, 3) equals 3.375f , and std::pow(4, 0.5) is equal to 2.
sqrt(arg)	**Square Root** of Arg
round(arg)	**Round** Arg to nearest int(floating-point[Whole] number). std::round(-1.5f) gives -2.0f
lround() & llround()	**Round** to int of type long & long long. std::lround(0.5) gives 1L
cos(), sin(), and tan()) inverse functions-> **(std::acos(), asin(), and atan())**	**Angles** are always expressed in **radians**

calculate the **Cosine of an angle in radians:**

double angle {1.5}; // In radians

```
double cosine_value {std::cos(angle)};
```

If the angle is in degrees, you can calculate the **Tangent by using a value for π to convert to radians:**

```
float angle_deg {60.0f};        // Angle in degrees
const float pi { 3.14159265f };
const float pi_degrees {180.0f};
float tangent {std::tan(pi * angle_deg/pi_degrees)};
```

If you know the height of a church steeple is 100 feet and you're standing 50 feet from its base, you can **calculate Tangent the angle in radians of the top of the steeple like this:**

```
double height {100.0};                  // Steeple height- feet
double distance {50.0};         // Distance from base
double angle {std::atan(distance / height)};            // Result in radians
```

You can use this value in angle and the value of distance to **calculate the distance from your toe to the top of the steeple:**

```
double toe_to_tip {distance / std::sin(angle)};
```

Formatting Stream Output 51

You can change how data is formatted when it is written to an output stream using **stream manipulators**, which are declared in the **#include<iomanip>** and **#include<ios>** Standard Library headers.

You apply a stream manipulator to an output stream with the insert operator, <<. We'll just introduce the most useful manipulators.

Consult a Standard Library reference if you want to get to know the others.

All manipulators declared by ios are automatically available if you include the familiar **#include<iostream>** header.

Unlike those of the **#include<iomanip>** header, these stream manipulators do not require an argument:

Stream Manipulator	Description
std::fixed	Output: Floating-point data in a **fixed-point notation**
std::scientific	Output: All Subsequent Floating-point data in **Scientific Notation**, which always includes an exponent and one digit before the decimal point.
std::defaultfloat	Revert to **default floating-point** data presentation.
std::dec	All subsequent int outputs are **Decimal**.
std::hex	All subsequent int outputs are **Hexadecimal**.
std::oct	All subsequent int outputs are **Octal**.
std::showbase	Outputs the **base prefix for hexadecimal and octal integer values**. Inserting **std::noshowbase** in a stream will switch this off.
std::left	Output **justified to the left**.
std::right	Default. Output **justified to the right**.
#include<iomanip> (header)->Stream Manipulators	
std::setprecision(n)	Sets the floating-point precision or the number of decimal places to n digits. **OR** If the default floating-point output presentation is in effect, n specifies the number of digits in the output value. **OR** If fixed or scientific format has been set, n is the number of digits following the decimal point. The default precision is 6.
std::setw(n)	Sets the output field width to n characters, but only for the next output data item. Subsequent output reverts to the default where the field width is set to the number of output character needed to accommodate the data.
std::setfill(ch)	When the field width has more characters than the output value, excess characters in the field will be the default fill character, which is a space. This sets the fill character to be ch for all subsequent output.

Mixed Expressions and Type Conversion 53

implicit conversions -- compiler will arrange to convert one of the operand values to the same (Data) type as the other.

compiler chooses the operand with the type that has the more limited range of values as the one to be converted to the type of the other. In effect, it ranks the types in the following sequence, from high to low:

1. long double 2. double 3. float
4. unsigned long long 5. long long 6. unsigned long
7. long 8. unsigned int 9. int

subtraction of unsigned integers wraps around to very large positive numbers is sometimes called **underflow**.

addition is called **overflow**.

Explicit Type Conversion 54

static_cast<type_to_convert_to>(expression)

The **static_cast** keyword reflects the fact that the cast is checked statically, that is, when the code is compiled. With classes, the conversion is checked **dynamically**, that is, when the program is executing.

By adding an explicit cast, you signal the compiler that a narrowing conversion is intentional.

With integer division, casting from a floating-point type to an integral type uses **truncation**.

The std::round(), lround(), and llround() functions from the cmath header allow you to round floating-point numbers to the nearest integer. In many cases, this is better than (implicit or explicit) casting, where truncation is used instead.

Old-Style Casts 56

Don't Use. Outdated C command for C++ (type_to_convert_to)expression

Finding the Limits 57

To display the maximum value you can store in a variable of type double, you could write this:

std::cout << "Maximum value of type double is " << std::numeric_limits<double>::max();

To get the lowest negative value a type can represent, you should use lowest() instead:

std::numeric_ limits::lowest()

Finding Other Properties of Fundamental Types 58

You can retrieve many other items of information about various types. The number of binary digits, or bits for example, is returned by this expression: **std::numeric_limits<type_name>::digits**

To obtain the special floating-point values for infinity and not-anumber (NaN), you should use expressions of the following form:

float positive_infinity = std::numeric_limits<float>::infinity();
double negative_infinity = -std::numeric_limits<double>::infinity();
long double not_a_number = std::numeric_limits<long double>::quiet_NaN();

Besides quiet_NaN(), there's a function called signaling_NaN()

Working with Character Variables 59

Variables of type **char** are used primarily to store a code for a single character and occupy 1 byte.

```
char letter; // Uninitialized - so junk value
char yes {'Y'},no {'N'}; // Initialized with character literals
char ch {33}; // Integer initializer equivalent to '!'
```

You can initialize a variable of type char with a character literal between single quotes or by an integer.

```
char ch {'A'};
char letter {ch + 5};   // letter is 'F'
++ch;                   // ch is now 'B'
```

ch += 3; // ch is now 'E'

When you write a char variable to cout, it is output as a character, not as an integer.
If you want to see it as a numerical value, you can cast it to another integer type. Here's an example:

```
std::cout << "ch is '" << ch
   << "' which is code " << std::hex << std::showbase
   << static_cast<int>(ch) << std::endl;
```

This produces the following output:
ch is 'E' which is code 0x45

Working with Unicode Characters 60

4 integer types that store Unicode characters: **char, wchar_t, char16_t,** and **char32_t**

Unicode –Gives Ability to work with characters for multiple Languages simultaneously or if you want to handle character sets

for many non-English languages, 256 character codes doesn't go nearly far enough. Int Code for Letters.

Type **wchar_t** is a fundamental type that can store all members of the largest extended character set that's supported by an implementation. The type name derives from wide characters because the character is "wider" than the usual single-byte character. By contrast, type char is referred to as "narrow" because of the limited range of character codes that are available.

You define wide-character literals in a similar way to literals of type char, but you prefix them with L. Here's an example:

wchar_t wch {L'Z'};

This defines wch as type wchar_t and initializes it to the wide-character representation for Z. Your keyboard may not have keys for representing other national language characters, but you can still create them using hexadecimal notation. Here's an example:

wchar_t wch {L'\x0438'}; // Cyrillic и

single quotes is an escape sequence that specifies the hexadecimal representation of the character code.

Type wchar_t does not handle international character sets very well. It's much better to use type char16_t, which stores characters encoded as UTF-16, or char32_t, which stores UTF-32 encoded characters. Here's an example of defining a variable of type char16_t:

char16_t letter {u'B'}; // Initialized with UTF-16 code for B
char16_t cyr {u'\x0438'}; // Initialized with UTF-16 code for cyrillic и

The lowercase u prefix to the literals indicates that they are UTF-16. You prefix UTF-32 literals with **uppercase U**. Here's an example:

char32_t letter {U'B'}; // Initialized with UTF-32 code for B
char32_t cyr {U'\x044f'}; // Initialized with UTF-32 code for cyrillic я

Of course, if your editor and compiler have the capability to accept and display the characters, you can define cyr like this:
char32_t cyr {U'я'};

You should not mix output operations on wcout with output operations on cout.

The auto Keyword 61

Compiler Automatically assigns Data Types to a Variable.
auto m {10}; // m has type int
auto n {200UL}; // n has type unsigned long
auto pi {3.14159}; // pi has type double

-The compiler will deduce the types for m, n, and pi from the initial values you supply.
-You can use functional or assignment notation with auto for the initial value as well:
auto m = 10; // m has type int
auto n = 200UL; // n has type unsigned long
auto pi(3.14159); // pi has type double

you should simply never use braced initializers with auto.

Instead, either explicitly state the type or use assignment or functional notation.

Summary 62

- **Constants** of any kind are called **literals**. All literals have a type.
- You can define integer literals as decimal, hexadecimal, octal, or binary values.
- A **floating-point** literal must contain a decimal point or an exponent or both. If there is neither, it's an integer.
- The fundamental types that store **integers** are **short, int, long,** and **long long**. These store **signed** integers, but you can also use the type modifier unsigned preceding any of these type names to produce a type that occupies the same number of bytes but stores unsigned integers.
- The **floating-point** data types are **float**, **double**, and **long double**.
- Uninitilized variables generally contain garbage values. Variables may be given initial values when they're defined, and it's good programming practice to do so. A braced initializer is the preferred way of specifying initial values.
- A variable of type **char** can store a single character and occupies one byte. Type char may be signed or unsigned, depending on your compiler. You can also use variables of the types signed char and unsigned char to store integers. Types char, signed char, and unsigned char are different types.
- Type **wchar_t** stores a wide character and occupies either two or four bytes, depending on your compiler. Types **char16_t** and **char32_t** may be better for handling Unicode characters in a cross-platform manner.
- You can fix the value of a variable by using the **const** modifier. The compiler will check for any attempts within the program source file to modify a variable defined as const.
- The four main mathematic operations correspond to the binary +, -, *, and / operators.

For integers, the modulus operator **%** gives you the remainder after integer division.

- **++** and **--** **operators** are special shorthand for adding or subtracting one in numeric variables. Both exist in postfix / prefix forms.
- You can mix different types of variables and constants in an expression. The compiler will arrange for one operand in a binary operation to be automatically converted to the type of the other operand when they differ.
- The compiler will automatically convert the type of the result of an expression on the right of an assignment to the type of the variable on the left where these are different. This can cause loss of information when the left-side type isn't able to contain the same information as the right-side type—double converted to int, for example, or long converted to short.
- You can explicitly convert a value of one type to another using the **static_cast<>() operator**.

■Chapter 3: Working with Fundamental Data Types 65

Operator	Conditions / Functions
x & y	AND Condition Produces 1 bit when **both** are 1. FINDS SIMILARITIES
x | y (a OR b OR Both)	OR Condition Produces 1 bit when **either** is 1 FINDS ALL BITS.
x ^ y (Either a OR b, NOT Both)	Exclusive OR Condition contains a 1 if and only if one of the corresponding input bits is equal to 1, while the other equals 0. Whenever both input bits are equal, even if both are 1, the resulting bit is 0. FINDS DIFFERENCES
~x	Complement Turn off condition. INVERT BITS.
x << y	(Multiply by Powers of 2) Shift x's Bits Left by y (2^y)
x >> y	(Divide by Powers of 2) Shift x's Bits Right by y (2^y)

Operator Precedence and Associativity 65

- You don't need to memorize the operator precedence and associativity for all operators, but you need to be conscious of it when writing code. Always use parentheses if you are unsure about precedence.
- The type-safe **enumerations type** are useful for representing fixed sets of values, especially w/ names, such as days of the week or suits in a pack of playing cards.
- **Bitwise operators** are necessary when working with **flags**—single bits that signify a state. These arise surprisingly often—when dealing with file input and output, for example. The bitwise operators are also essential when you are working with values packed into a single variable. One extremely common example thereof is RGB-like encodings, where three to four components of a given color are packed into one 32-bit integer value.
- **using** keyword allows you to define aliases for other types. In legacy code, may still encounter typedef being used.
- By default, a variable defined within a block is **automatic**, which means that it exists only from the point at which it is defined to the end of the block in which its definition appears, as indicated by the closing brace of the block that encloses its definition.
- Variables can be defined outside of all the blocks in a program, in which case they have global namespace scope and static storage duration by default. Variables with global scope are accessible from anywhere within the program file that contains them, following the point at which they're defined, except where a local variable exists with the same name as the global variable. Even then, they can still be reachedby using the **scope resolution operator** (**::**).

Chapter 4: Making Decisions 89

Computer Decision Making -- Altering the sequence of execution depending on the result of a comparison of True/False.

Comparing Data Values 89

Relational Operators	
Operator	**Meaning**
<	Less Than
<=	Less Than OR Equal to
>	Greater Than
>=	Greater Than OR Equal to
==	Equal to
!=	Not Equal to

Make sure to use == when comparing Variables, opposed to =

bool isValid {true}; // Define and initialize a logical variable

Defines variable isValid as type bool with value of true. If you initialize a bool variable using empty braces, {}, its initial value is false:

bool correct {}; // Define and initialize a logical variable to false

Applying the Comparison Operators 90

std::cout << std::boolalpha; // Shows True or False ----rather than 1 or 0

std::cout << std::noboolalpha; // Shows 1/0 ---rather than T/F

Comparing Floating-Point Values 92

The comparison operators are all of lower precedence than the arithmetic operators, so none of the parentheses are strictly necessary, but they do help make the expressions clearer

The if Statement 92

The **if statement** enables you to choose to execute a single statement, or a block of statements, when a given condition is true.

if (condition)
statement;
Next statement;
	if (condition)
	{
	statement;
	...
	}
	Next statement;

NEVER put Semicolon after an if statement: if(condition);

Without the braces, only the first statement would be the subject of the if

When you have a numerical value where a bool value is expected, the compiler will insert an implicit conversion to convert the numerical value to type bool. This is useful in decision-making code.

Nested if Statements 96

The condition of the inner if is tested only if the condition for the outer if is true.

You can nest ifs to whatever depth you require.

Character Classification and Conversion 97

- The letters A to Z are represented by a set of codes where the code for 'A' is the minimum and the code for 'Z' is the maximum.
- The codes for the uppercase letters are contiguous, so no nonalphabetic characters lie between the codes for 'A' and 'Z'.
- All uppercase letters in the alphabet fall within the range A to Z.

The first two assumptions are true for any character encoding used in practice today, the third is not true for many languages.

The Greek alphabet, for instance, knows uppercase letters such as Δ, Θ, and Π; the Russian one contains Ж, Ф, and Щ; and even Latin-based languages such as French often use capital letters such as É and Ç whose encodings won't lie at all between 'A' and 'Z'.

- **locale:** a set of parameters that defines the user's language and regional preferences, including the national or cultural character set and the formatting rules for currency and dates.

character classification functions provided by the **#include<cctype>** header

Character Classification Functions with cctype Header	
Function(std::)	**Operation ----Tests where (c) =**
isupper(c)	Uppercase letter, by default 'A' to 'Z'.
islower(c)	Lowercase letter, by default 'a' to 'z'.
isalpha(c)	Uppercase or Lowercase letter (or any alphabetic character, neither uppercase nor lowercase, should the locale's alphabet contain such characters).
isdigit(c)	Digit, '0' to '9'.
isxdigit(c)	Hexadecimal digit, either '0' to '9', 'a' to 'f', or 'A' to 'F'.
isalnum(c)	Alphanumeric character; same as isalpha(c) || isdigit(c).
isspace(c)	Whitespace, by default, or: a space (' '), newline ('\n'), carriage return ('\r'), form feed ('\f'), or horizontal ('\t') or vertical tab ('\v').
isblank(c)	Space character used to separate words within a line of text. By default either a space (' ') or a horizontal tab ('\t').

ispunct(c)	Punctuation character. By default, this will be either a space or one of the following: _ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = , \ " '
isprint(c)	Printable character ::: Includes: uppercase or lowercase letters, digits, punctuation characters, and spaces.
iscntrl(c)	Control character, the opposite of a printable character.
isgraph(c)	Has a graphical representation, which is true for any printable character other than a space.

These functions returns a value of type int. Value is nonzero (true) if the character is of the type being tested for, and 0 (false) if it isn't.

```
if (std::isupper(letter))
{ std::cout << "You entered an uppercase letter." << std::endl;
 return 0;}
if (std::islower(letter))
{ std::cout << "You entered a lowercase letter." << std::endl;
 return 0;}
```

Converting Character's with cctype (Upper/Lower)	
Function	**Operation**
tolower(c)	If c is uppercase, the lowercase equivalent is returned; otherwise, c is returned.
toupper(c)	If c is lowercase, the uppercase equivalent is returned; otherwise, c is returned.

Result = type int, so you need to explicitly cast it if you want to store it as type char.

The if-else Statement 99

you may want to execute one block of statements when the condition is true and another set when the condition is false.

```
if (condition)
{   // Statements when condition is true     }
else
{   // Statements when condition is false    }
// Next statement…
```

```
if (std::isalnum(letter))
{   std::cout << "It is a letter or a digit." << std::endl;          }
else
{   std::cout << "It is neither a letter nor a digit." << std::endl;          }
```

Nested if-else Statements 101

```
if (coffee == 'y')
 if (donuts == 'y')
 std::cout << "We have coffee and donuts." << std::endl;
 else
 std::cout << "We have coffee, but not donuts." << std::endl;
```

■ Caution An else always belongs to the nearest preceding if that's not already spoken for by another else. The potential for confusion here is known as the dangling else problem. ALWAYS USE BRACES FOR IF STATEMENTS TO AVOID CONFUSION.

Understanding Nested ifs 102

When an else block is another if, writing else if on one line is an accepted convention.

Using **Else If** as If for more answer clarification:

```
if (coffee == 'y')
   if (donuts == 'y')
      std::cout << "We have coffee and donuts." << std::endl;
```

```
  else
    std::cout << "We have coffee, but not donuts." << std::endl;
else if (tea == 'y')
  std::cout << "We have no coffee, but we have tea, and maybe donuts..." << std::endl;
else
  std::cout << "No tea or coffee, but maybe donuts..." << std::endl;
```

Logical Operators 103

logical operators provide a neat and simple solution. Using logical operators, you can combine a series of comparisons into a single expression so that you need just one if, almost regardless of the complexity of the set of conditions.

Logical Operators	Description
&&	Logical AND
||	Logical OR
!	Logical NOT (negation)

Logical AND 104 &&

Logical OR 104 ||

Logical Negation 105 !

■ Caution Let foo, bar, and xyzzy be variables (or any expressions if you will) of type bool.

if (foo == true) ...
if (bar == false) ...
if (xyzzy != true) ...

While technically correct, it is generally accepted that you should favor the following equivalent yet shorter if statements instead:

if (foo) ...
if (!bar) ...
if (!xyzzy) ...

Combining Logical Operators 105

■ Tip When combining logical operators, it is recommended to always add parentheses to clarify the code.

Logical Operators on Integer Operands 107

logical operators can be—and actually fairly often are—applied to integer operands instead of Boolean operands.

Logical Operators vs. Bitwise Operators 108

logical operators always evaluate to a value of type bool, even if their operands are integers. The converse is true for bitwise operators: they always evaluate to an integer number, even if both operands are of type bool.

the integer result of a bitwise operator always converts back to a bool, it may often seem that logical and bitwise operators can be used interchangeably.

bitwise operators, the binary logical operators are so-called **short-circuit operators**.

short-circuit operators: if the first operand to a binary logical expression already determines the outcome, the compiler will make sure no time is wasted evaluating the second operand. Properties of: && and ||. NOT Bitwise operators & and |.

Short-circuiting semantics of logical operators, often exploited by C++ programmers:

- If you need to test for multiple conditions that are glued together with logical operators, then you should put the cheapest ones to compute first. This technique only really pays off if one of the operands is truly expensive to calculate.
- Short-circuiting is more commonly utilized to prevent the evaluation of right-hand operands that would otherwise fail to evaluate—as in cause a fatal crash. This is done by putting other conditions first that short-circuit whenever the other

operands would fail. As we will see later in this book, a popular application of this technique is to check that a pointer is not null before dereferencing it.

Logical XOR 132

1, but not the other

```
if ((age < 20) ^ (balance >= 1'000'000))
{
 ...
}
```

In other words, this test is equivalent to either one of the following combinations of logical operators:

```
if ((age < 20 || balance >= 1'000'000) && !(age < 20 && balance >= 1'000'000))
   { ...}
```

Both Statements are Equal.

```
if ((age < 20 && balance < 1'000'000) || (age >= 20 && balance >= 1'000'000))
   { ...}
```

The '?'/':' Conditional Operator(s) 110 c = a > b? a : b; // Set c to the higher of a and b

conditional operator ,or, **ternary operator** because it involves three operands—the only operator to do so.

It parallels the if-else statement, in that instead of selecting one of two statement blocks to execute depending on a condition, it selects the value of one of two expressions. Thus, the conditional operator enables you to choose between two values.

Suppose you have two variables, a and b, and you want to assign the value of the greater of the two to a third variable, c. The following statement will do this:

c = a > b? a : b; // Set c to the higher of a and b

Which is the Procedural equivalent of:

```
if (a > b)
   { c = a; }
else
   { c = b; }
```

Using Strings:

```
std::cout << "You have " << mice
          << (mice == 1? " mouse" : " mice")
          << " in total." << std::endl;
```

---

2 Conditional Operators to choose between 3 options:

```
std::cout << (a < b ? "a is less than b." :
 (a == b ? "a is equal to b." : "a is greater than b."));
```

The switch Statement 112

The **switch statement** enables you to select from multiple choices.

The choices are identified by a set of fixed integer or enumeration values, and the selection of a particular choice is determined by the value of a given integer or enumeration constant.

The choices in a switch statement are called **cases**.

For Winning Lottery Numbers:

```
switch (ticket_number)    //Entered Your Ticket Number
{
case 147:
   std::cout << "You win first prize!";
   break;
case 387:
   std::cout << "You win second prize!";
```

```
    break;
case 29:
    std::cout << "You win third prize!";
    break;
default:
    std::cout << "Sorry, you lose.";
    break;
}
```

- ■ **Note** You can use switch on values of integral (int, long, unsigned short, etc.), character (char, etc.), and enumeration types Switching on Boolean values is allowed as well, but instead of a switch on Booleans, you should just use if/else statements.
- Unlike some other programming languages, however, C++ does not allow you to create switch() statements with conditions and labels that contain expressions of any other type. A switch that branches on different string values, for instance, is not allowed.
- switch statement appear in a block, and each choice is identified by a casevalue. A **case value** appears in a **case label**:

case case_value: **EX: case 5: doSomething(); break;**

- called a case label because it labels the statements or block of statements that it precedes.
- Each case value must be a **constant expression**, which is an expression that the compiler can evaluate at compile time.
- Any case label must either be the same type as the condition expression inside the preceding switch() or be convertible to that type.
- **Default label** identifies the **default case**, which is a catchall that is selected if none of the other cases is selected.
- Executing a break statement breaks out of the switch & causes execution to continue with the statement following the closing brace.
- If you omit the break statement for a case, the statements for the following case will execute. No need for a break after the final case (usually the default case) because execution leaves the switch at this point anyway.
- It's good programming style to include it, it safeguards against falling through to another case that you might add to a switch later.
- **switch, case, default,** and **break** are all keywords

Multiple Cases:

```
case 'a': case 'e': case 'i': case 'o': case 'u':
        std::cout << "You entered a vowel." << std::endl;
        break;
default:
        std::cout << "You entered a consonant." << std::endl;
        break;
```

```
switch (std::tolower(letter))
 {
 case 'a': case 'e': case 'i': case 'o': case 'u':
 std::cout << "You entered a vowel." << std::endl;
 return 0; // Ends the program
 }
 // We did not exit main() in the above switch, so letter is not a vowel:
 std::cout << "You entered a consonant." << std::endl;
```

Note that after a return statement you should never put a break statement anymore.

Fallthrough 116

- **fallthrough:** Remove break statements from a switch statement, the code beneath the case label directly following the case without a break statement then gets executed as well. in a way we "fall through" into the next case.

- more often than not fallthrough signals a bug, many compilers issue a warning if a nonempty switch case is not followed by either a break or a return statement.
- Sometimes it's a good thing, EX: if a few different lottery numbers = 2nd Prize.

```
switch (ticket_number)
{
...
case 929:
 std::cout << "You win a special bonus prize!" << std::endl;
 [[fallthrough]];
case 29:
case 78:
 std::cout << "You win third prize!" << std::endl;
 break;
...
}
```

Statement Blocks and Variable Scope 118

Initialization Statements 119

if (initialization; condition) ...

The additional initialization statement is executed prior to evaluating the condition expression, the usual Boolean expression of the if statement. You will use such initialization statements mainly to declare variables local to the if statement. Like so:

```
if (auto lower{ static_cast<char>(std::tolower(input)) }; lower >= 'a' && lower <= 'z') {
   std::cout << "You've entered the letter '" << lower << '\'' << std::endl;
}
// ... more code (lower does not exist here)
```

switch (initialization; condition) { ... }

Summary 120

- comparison operators- can compare two values. This will result in a value of type bool, which can be either true or false.
- You can convert a bool value to an integer type—true will convert to 1, and false will convert to 0.
- Numerical values can be converted to type bool—a zero value converts to false, and any nonzero value converts to true. When a numerical value appears where a bool value is expected—such as in an if condition—the compiler will insert an implicit conversion of the numerical value to type bool.
- The **if statement** executes a statement or a block of statements depending on the value of a condition expression. If the condition is true, the statement or block executes. If the condition is false, it doesn't.
- The **if-else statement** executes a statement or block of statements when the condition is true and executes another statement or block when the condition is false.
- if and if-else statements can be nested.
- The **logical operators** **&&**, **||**, and **!** are used to string together more complex logical expressions. The arguments to these operators must either be Booleans or values that are convertible to Booleans (such as integral values).
- The conditional operator selects between two values depending on the value of an expression.
- The switch statement provides a way to select one from a fixed set of options, depending on the value of an expression of integral or enumeration type.

■Chapter 5: Arrays and Loops 123

An array enables you to work with several data items of the same type using a single name, the array name.
. A loop is a mechanism for repeating one or more statements as many times as your application requires.

Arrays 123 EX: int arrayName[366]

- **Array** stores several data items of the same type, an array of integers or an array of characters (or any type of data)
- array is a variable that represents a sequence of memory locations, each storing an item of data of the same data type.

Using an Array 123

- The data values are called **elements**. The number of elements specified between the brackets is the **size** of the array.
- The size of an array must always be specified using a **constant integer expression**. Any integer expression that the compiler can evaluate at compile time may be used, though mostly this will be either an integer literal or a const integer variable that itself was initialized using a literal.
- You refer to an array element using an integer called an **index**. The index of a particular array element is its offset from the first element. The first element has an offset of 0 and therefore an index of 0; an index value of 3 refers to the fourth array element.
- **unsigned int height[6] {26, 37, 47, 55, 62, 75};** // Define & initialize array of 6 heights
- ■ Note The type of the array will determine the amount of memory required for each element. The elements of an array are stored in one contiguous block of memory. So if the unsigned int type is 4 bytes on your computer, the height array will occupy 24 bytes.

To Make a Constant Array:

unsigned int sum {};

sum = height[0] + height[1] + height[2]; // The sum of three elements

---

Copy Array Values

height[3] = height[2]; // Copy 3rd element value to 4th element

Understanding Loops 125 For, For(Range),

Loop: Mechanism to execute a statement or block of statements repeatedly until a particular condition is met.

2 things make a loop: the statement or block of statements to be executed repeatedly forms the **body** of the loop, and a **loop condition** that determines when to stop repeating the loop. A single execution of a loop's body is called an **iteration**.

A loop condition can take different forms to provide different ways of controlling the loop. loop conditions can do the following:

- Execute a loop a given number of times
- Execute a loop until a given value exceeds another value
- Execute the loop until a particular character is entered from the keyboard
- Execute a loop for each element in a collection of elements

You choose the loop condition to suit the circumstances. You have the following varieties of loops:

- **for loop** primarily provides for executing the loop a prescribed number of times, but there is considerable flexibility beyond that.
- The **range-based for loop** executes one iteration for each element in a collection of elements.
- The **while loop** continues executing as long as a specified condition is true. The condition is checked at the beginning of an iteration, so if the condition starts out as false, no loop iterations are executed.
- The **do-while loop** continues to execute as long as a given condition is true. This differs from the while loop in that the do-while loop checks the condition at the end of an iteration. *This implies that at least one loop iteration always executes.*

The for Loop 126

```
for (initialization; condition; iteration)
{
 // Loop statements
}
// Next statement
```

Initialization expression: Evaluated only once, at the beginning of the loop. Loop condition: checked next, and if it is true, the loop statement or statement block executes. If the condition is false, the loop ends, and execution continues with the statement after the loop. After each execution of the loop statement or block, the iteration expression is evaluated, and the condition is checked to decide whether the loop should continue.

typical usage of the for loop, the first expression initializes a counter, the second expression checks whether the counter has reached a given limit, and the third expression increments the counter

```
double rainfall[12] {1.1, 2.8, 3.4, 3.7, 2.1, 2.3, 1.8, 0.0, 0.3, 0.9, 0.7, 0.5};
double copy[12] {};
for (size_t i {}; i < 12; ++i) // i varies from 0 to 11
{
 copy[i] = rainfall[i]; // Copy ith element of rainfall to ith element of copy
}
```

- The first expression defines ***i*** as type ***size_t,*** an initial value of 0. The size_t type is returned by the **sizeof operator**, ***unsigned integer type***, generally used for sizes and counts of things. As ***i*** will be used to ***index*** the arrays, using size_t makes sense.
- The second expression, the loop condition, is true as long as i is less than 12, so the loop continues while i is less than 12.
- When i reaches 12, the expression will be false, so the loop ends.
- The third expression increments i at the end of each loop iteration, so the loop block that copies the ith element from rainfall to copy will execute with values of i from 0 to 11.
- **size_t** : Alias for one of the unsigned integer types, sufficiently large to contain the size of any type the compiler supports (including any array). The alias is defined in the **#include<cstddef>** header, as well as in a number of other headers
- ■ **Caution** Array index values are not checked to verify that they are valid. Important to not reference elements outside the bounds of the array. Storing data using an index value that's outside the valid range for an array will inadvertently overwrite something in memory or cause a **segmentation fault** or **access violation** (both are synonymous / denote an error raised by the operating system when it detects unauthorized memory access). Either way, your program will almost certainly come to a sticky end.
- A loop defines a scope. The loop statement or block, including any expressions that control the loop, falls within the scope of a loop. Any automatic variables declared within the scope of a loop do not exist outside it.

Avoiding Magic Numbers 128

To Prevent Bugs: Create const Values for arrays, for adding array values & changing it only in one place, a **Magic Number**.

A safer solution already is to define a const variable for the array size and use that instead of the explicit value:

```
const size_t size {12};
double rainfall[size] {1.1, 2.8, 3.4, 3.7, 2.1, 2.3, 1.8, 0.0, 0.3, 0.9, 0.7, 0.5};
double copy[size] {};
for (size_t i {}; i < size; ++i) // i varies from 0 to size-1
{
 copy[i] = rainfall[i]; // Copy ith element of rainfall to ith element of copy
}
```

Defining the Array Size with the Braced Initializer 130

You can omit the size of the array when you supply one or more initial values in its definition.

```
int values[] {2, 3, 4};
```

```
int values[3] {2, 3, 4};  //Same as ^
```

The advantage of omitting the size is that you can't get the array size wrong; the compiler determines it for you.

Determining the Size of an Array 130

You don't want to specify a magic number for the array size when you let the compiler decide the number of elements from the braced initializer list. You need a fool-proof way of determining the size when necessary.

int values[] {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};

Can use the expression **std::size(values)** to obtain the array's size, 10. Also **std::vector<>** and **std::array<>** containers

The expression **sizeof(values) / sizeof(values[0])** divides the number of bytes occupied by the whole array by the number of bytes for one element, so this evaluates to the number of elements in the array.

std::size() is easier to use and understand than the old sizeof-based expression. So, if possible, you should always use std::size().

```
int sum {};
for (size_t i {}; i < std::size(values); sum += values[i++]) {}
```

Controlling a for Loop with Floating-Point Values 132

■**Note** Any number that is a fraction with an odd denominator cannot be represented exactly as a binary floating-point value.

■**Caution** Comparing floating-point numbers can be tricky. You should always be cautious when comparing the result of floating-point computations directly using operators such as ==, <=, or >=. Rounding errors almost always prevent the floating-point value from ever becoming exactly equal to the mathematical precise value.

More Complex for Loop Control Expressions 135

■ **Note** The optional initialization statement of if and switch statements is completely equivalent to that of for loops. So there too you can define multiple variables of the same type at once if you want.

The Comma Operator 136

It combines two expressions into a single expression, where the value of the operation is the value of its right operand. This means that anywhere you can put an expression, you can also put a series of expressions separated by commas.

The Range-Based for Loop 137

range-based for loop: iterates over all the values in a range of values.

range: Array is a range of elements, and a string is a range of characters. The **containers** provided by the Standard Library are all ranges as well. This is the general form of the range-based for loop:

for (range_declaration : range_expression)
loop statement or block;

range_declaration identifies a variable that will be assigned each of the values in the range in turn, with a new value being assigned on each iteration. The **range_expression** identifies the range that is the source of the data.

```
int values [] {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
int total {};
for (int x : values)
 total += x;
```

Variable x will be assigned a value from the values array on each iteration. It will be assigned values 2, 3, 5, and so on, in succession.

The while Loop 138

while loop uses a logical expression to control execution of the loop body.

```
while ( condition )
{
 // Loop statements ...
}
// Next statement
```

You can use any expression to control the loop, as long as it evaluates to a value of type bool or can be implicitly converted to type bool. If the loop condition expression evaluates to a numerical value, for example, the loop continues as long as the value is nonzero. A zero value ends the loop.

■ Note Any for loop can be written as an equivalent while loop, and vice versa. For instance, a for loop has the following generic form:

```
for (initialization; condition; iteration)
 body
```

This can typically be written using a while loop as follows:

```
{
initialization;
    while (condition)
    {
    body
    iteration
    }
}
```

The while loop needs to be surrounded by an extra pair of curly braces to emulate the way the variables declared in the initialization code are scoped by the original for loop.

The do-while Loop 140

do-while loop is similar to the while loop in that the loop continues for as long as the specified loop condition remains true. The only difference is that the loop condition is checked at the end of the do-while loop, rather than at the beginning, **so the loop statement is always executed at least once**.

```
do
{
 // Loop statements...
} while ( condition );
// Next statement
```

the semicolon that comes after the condition between the parentheses is absolutely necessary.

ideal for situations where you have a block of code that you always want to execute once and may want to execute more than once.

■ Caution While the semicolon after a do-while statement is required by the language, you should normally never add one after the while() of a regular while loop:

Nested Loops 142

You can nest loops within loops to whatever depth you require to solve your problem. You can nest a for loop inside a while loop inside a do-while loop inside a range-based for loop, if you have the need. They can be mixed in any way you want.

Skipping Loop Iterations 145

continue statement: Use to skip one loop iteration and press on with the next. **continue;** // Go to the next iteration

Breaking Out of a Loop 146

use the **break statement**.

executing a break statement within a loop ends the loop immediately, and execution continues with the statement following the loop.

often used with an **indefinite loop**

Indefinite Loops 146

- **indefinite loop** can potentially run forever.
- Omitting the second control expression in a for loop results in a loop that potentially executes an unlimited number of iterations.
- Indefinite loops have many practical uses, such as programs that monitor some kind of alarm indicator, or that collect data from sensors in an industrial plant, or when reading a variable quantity of input data. An indefinite loop can be useful when you don't know in advance how many loop iterations will be required, such as In these circumstances, you code the exit from the loop within the loop block, not within the loop control expression.

INDEFINITE FOR:

```
for (;;)
{
 // Statements that do something...
 // ... and include some way of ending the loop
}
```

INDEFINITE WHILE:

```
while (true)
{
 // Statements that do something...
 // ... and include some way of ending the loop
}
```

- do-while Indefinite Loop isn't advantageous.
- The obvious way to end an indefinite loop is to use the break statement.
- **bubble sort**: elements gradually "bubble up" to their correct position in the array. Not the most efficient sorting method, but it has the merit that it is very easy to understand, and it's a good demonstration of yet another use for an indefinite loop.

■**Tip** Indefinite loops, or break statements, should be used judiciously. sometimes considered bad coding style. Put the conditions that determine when a loop terminates between the round parentheses of the for or while statement. Doing so increases code readability because this is where every C++ programmer will be looking for such conditions. Any (additional) break statements inside the loop's body are much easier to miss and can therefore make code harder to understand.

Controlling a for Loop with Unsigned Integers 150

■ **Caution** Take care when subtracting from unsigned integers. Any value that mathematically speaking should be negative then wraps around to become a huge positive number. These types of errors can have catastrophic results in loop control expressions.

Arrays of Characters 152

an array of characters, can store one character, or a string.

Characters in the string are stored in successive array elements, followed by a special string termination character called the **null character** that you write as '\0'. The null character marks the end of the string. referred to as a C-style string.

Objects of type string are much more flexible and convenient for **string** manipulation than using arrays of type **char**.

You can also declare an array of type char and initialize it with a **string literal**, as follows:

```
char name[10] {"Mae West"};
std::cout << name << std::endl;
```

// Read a line of characters including spaces

```
std::cin.getline(text, max_length);
```

■ Caution You can't output the contents of an array of a numeric type by just using the array name. This works only for char arrays. And even a char array passed to an output stream must be terminated with a null character, or the program will likely crash.

getline() function expects two arguments between the parentheses. The first argument specifies where the input is to be stored, which in this case is the text array. The second argument specifies the maximum number of characters that you want to store. This includes the string termination character, '\0'

The period between the name of the cin object and that of its so-called member function getline() is called the **direct member selection operator**. This operator is used to access members of a class object.

you can optionally supply a third argument to the getline() function. This specifies an alternative to '\n' to indicate the end of the input. If you want the end of the input string to be indicated by an asterisk, use this statement to read the input:

```
std::cin.getline(text, maxlength, '*');
```

Multidimensional Arrays 155

One-dimensional arrays have one index can reference all the elements. **Multidimensional arrays**: arrays that require two or more index values to access an element. An array that requires two index values to reference an element is called a **two-dimensional array**. And so on, for as many dimensions as you think you can handle.

Suppose, as an avid gardener, that you want to record the weights of the carrots you grow in your small vegetable garden. To store the weight of each carrot, which you planted in three rows of four, you could define a two-dimensional array:

```
double carrots[3][4] {};
```

To store the weight of the third carrot in the second row, you could write the following:

```
carrots[1][2] = 1.5;
```

---

arrays two+ dimensions, the rightmost index value varies most rapidly, and the leftmost index varies least rapidly.

To Print Multidimensional Arrays, Row by Row.

```
for (size_t i {}; i < 3; ++i) // Iterate over rows
{
 for (size_t j {}; j < 4; ++j) // Iterate over elements within the row
 {
 std::cout << std::setw(12) << carrots[i][j];
 }
 std::cout << std::endl; // A new line for a new row
}
```

This uses magic numbers, 3 and 4, which you can avoid by using std::size():

```
for (size_t i {}; i < std::size(carrots); ++i)
{
 for (size_t j {}; j < std::size(carrots[i]); ++j)
 {
 std::cout << std::setw(12) << carrots[i][j];
```

```
 }
 std::cout << std::endl;
}
```

■ **Note** range-based loops should be used. To write the outer loop as a range-based for, you'll first need to learn about references.

Best not to use magic numbers for the array dimension sizes in the first place, so you could define the array as follows:

```
const size_t nrows {3}; // Number of rows in the array
const size_t ncols {4}; // Number of columns, or number of elements per row
double carrots[nrows][ncols] {};
```

to record 3 temperatures per day, 7 days a week, for 52 weeks of the year. Declare the following array to store such data as type int:

```
int temperatures[52][7][3] {};
```

Initializing Multidimensional Arrays 158

initializing values for a one-dimensional array are written between braces and separated by commas. Following on from that, you could declare and initialize the two-dimensional carrots array, with this statement:

```
double carrots[3][4]                    {
 {2.5, 3.2, 3.7, 4.1}, // First row
{4.1, 3.9, 1.6, 3.5}, // Second row
{2.8, 2.3, 0.9, 1.1} // Third row                  };
```

double carrots[3][4] {1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7}; The first four values in the list will initialize elements in row 0. The last three values in the list will initialize the first three elements in row 1. The remaining elements will be initialized with zero.

Multidimensional Character Arrays 160

The compiler, however, can determine only one of the dimensions in a multidimensional array, and it has to be the first

```
double carrots[][4] {
 {2.5, 3.2 }, // First row
 {4.1 }, // Second row
 {2.8, 2.3, 0.9 } // Third row  };
```

```
char stars[][80] {
 "Robert Redford",
 "Hopalong Cassidy",
 "Lassie",
 "Slim Pickens",
 "Boris Karloff",
 "Oliver Hardy"
 };
```

Allocating an Array at Runtime 161

The C++17 standard does not permit an array dimension to be specified at runtime.

That is, the array dimension must be a constant expression that can be evaluated by the compiler.

So-called variable-length arrays can be a useful feature, in case your compiler supports this, this is how it works:

```
size_t count {};
std::cout << "How many heights will you enter? ";
std::cin >> count;
unsigned int height[count]; // Create the array of count elements
```

Because the array size is not known at compile time, you cannot specify any initial values for the array.

Alternatives to Using an Array 164

The Standard Library defines a rich collection of data structures called **containers**: organize and access your data.

two most elemental containers: **std::array<>** and **std::vector<>**. A direct alternative to the plain arrays and much easier to work with, safer to use, and provide significantly more flexibility than the more low-level, built-in arrays.

Like all containers, std::array<> and std::vector<> are defined as **class templates**

The compiler uses the **std::array<T,N>** and **std::vector<T>** templates to create a concrete type based on what you specify for the template parameters, T and N.

Using array<Type,Number> Containers 164

The **array<T,N> template** is defined in the **#include<array>** header, so you must include this in a source file to use the container type. An array<T,N> container is a fixed sequence of N elements of type T, so it's just like a regular array except that you specify the type and size a little differently. Here's how you create an array<> of 100 elements of type double:

std::array<double, 100> values; ========= **double values[100];**

set all the elements to a given value, using the **fill() function** for the array<> object:

values.fill(3.14159265358979323846); // Set all elements to pi

size() function for an array<> object returns the number of elements as type size_t. It outputs 100:

std::cout << values.size() << std::endl;

size() function: first real advantage over a standard array because an array<> object always knows how many elements there are.

Used for passing arguments to functions.

Accessing Individual Elements

Can access/use elements using an index the same as for a standard array. Here's an example: **values[4] = values[3] + 2.0*values[1];**

at() function for an array<> object does and will detect attempts to use an index value outside the legitimate range. The argument to the at() function is an index, same as using square brackets, so you could write the for loop that totals the elements like this:

```
double total {};
for (size_t i {}; i < values.size(); ++i)
{ total += values.at(i); }
```

The at() function provides a further advantage over standard arrays.

Advantage: To access the first and last array elements. **values.front()** = values[0] and **values.back()** = values[values.size() - 1].

Containers are compared element by element. For a true result for ==, all pairs of corresponding elements must be equal. For inequality, at least one pair of corresponding elements must be different for a true result.

You can also assign one array<> container to another, as long as they both store the same number of elements of the same type.

them = those; // Copy all elements of those to them

vector<> container that can hold array<> objects as elements, each containing three int values:

std::vector<std::array> triplets;

access the built-in array that is encapsulated within the array<> object using its data() member.

Using std::vector<Type> Containers 169

size of a vector<> can grow automatically to accommodate any number of elements.

std::vector values;

You can add an element using the **push_back()** function for the container object. Here's an example:

values.push_back(3.1415); // Add an element to the end of the vector

You can initialize a vector<> with a predefined number of elements, like this:

std::vector<double> values(20); // Vector contains 20 double values - all zero

a vector<> container always initializes its elements. In this case, our container starts out with 20 elements that are initialized with zero. If you don't like zero as the default value for your elements, you can specify another value explicitly:

std::vector<long> numbers(20, 99L); // Vector contains 20 long values - all 99

The **first argument** that specifies the number of elements—20 in our example—does not need to be a constant expression. It could be the result of an expression executed at runtime or read in from the keyboard.

A further option for creating a vector<> is to use a braced list to specify initial values:

std::vector<unsigned int> primes { 2, 3, 5, 7, 11, 13, 17, 19 };

The primes vector container will be created with eight elements with the given initial values.

■ **Caution** Don't initialize the values and numbers vector<> objects using the Curly braces, instead using round parentheses:

std::vector<double> values(20); // Vector contains 20 double values - all zero

std::vector<long> numbers(20, 99L); // Vector contains 20 long values - all 99

This is because using braced initializers here has a significantly different effect, as the comments next to the statements explain:

std::vector<double> values{20}; // Vector contains 1 single double value: 20

std::vector<long> numbers{20, 99L}; // Vector contains 2 long values: 20 and 99

Initialize a vector<> with a given number of identical values/without repeating that same value over and over, cannot use curly braces.

You can use an index between square brackets to set a value for an existing element or just to use its current value in an expression.

values[0] = 3.14159265358979323846; // Pi

values[1] = 5.0; // Radius of a circle

values[2] = 2.0*values[0]*values[1]; // Circumference of a circle

Index values for a vector<> start from 0, just like a standard array.

You can always reference existing elements using an index between square brackets, but you cannot create new elements this way. For that use the push_back() function.

Consider using the at() function to refer to elements whenever there is the potential for the index to be outside the legal range.

Besides the at() function, **nearly all other advantages of array<> containers directly transfer to vector<>** as well:

- Each vector<> knows its size and has a **std::size()** member to query it.
- Passing a vector<> to a function is straightforward (see Chapter 8).
- Each vector<> has the convenience functions **front()** and **back()** to facilitate accessing the first and last elements of the vector<>.
- Two vector<> containers can be compared using <, >, <=, >=, ==, and != operators. Even works for vectors that do not contain the same number of elements. Same as when you alphabetically compare words of different length like Aardvark precedes zombie in the dictionary, even with more letters. The comparison of vector<> containers is similar. Any values the compiler knows how to compare can be by using <, >, <=, >=, ==, and !=. In technical speak, this principle is called a lexicographical comparison.
- Assigning a vector<> to another vector<> variable copies all elements, overwriting any elements that may have been there before, even if the new vector<> is shorter. If need be, additional memory will be allocated to accommodate more elements as well.
- A vector<> can be stored inside other containers, so you can, for instance, create a vector of vectors of integers.

A vector<> does not have a fill() member, though. Instead, it offers **assign()** functions that can be used to reinitialize the contents of a vector<>, much like you would when initializing it for the first time:

std::vector<long> numbers(20, 99L); // Vector contains 20 long values - all 99

numbers.assign(99, 20L); // Vector contains 99 long values - all 20

numbers.assign({99L, 20L}); // Vector contains 2 long values - 99 and 20

Deleting Elements from Vector 195

You can remove all the elements from a vector<> by calling the clear() function for the vector object.

std::vector<int> data(100, 99); // Contains 100 elements initialized to 99

data.clear(); // Remove all elements

---

empty() does not empty a vector<>; clear() does. Instead, the empty() member checks whether a given container is empty. Evaluates to the Boolean value of true if and only if the container contains no elements and does not modify the container at all in the process

You can remove the last element from a vector object by calling its pop_back() function. Here's an example:

std::vector<int> data(100, 99); // Contains 100 elements initialized to 99

data.pop_back(); // Remove the last element

---

std::vector<> will be the container you use most frequently.

■ **Tip** If you know the exact number of elements at compile time, use std::array<>. Otherwise, use std::vector<>.

Summary 174

- An array stores a fixed number of values of a given type.
- You access elements in a one-dimensional array using an index value between square brackets. Index values start at 0, so in a one-dimensional array, an index is the offset from the first element.
- An array can have more than one dimension. Each dimension requires a separate index value to reference an element. Accessing elements in an array with two or more dimensions requires an index between square brackets for each array dimension.
- A loop is a mechanism for repeating a block of statements.
- There are four kinds of loop that you can use: the while loop, the do-while loop, the for loop, and the range-based for loop.
- The while loop repeats for as long as a specified condition is true.
- The do-while loop always performs at least one iteration and continues for as long as a specified condition is true.
- for loop is typically used to repeat a given number of times and has three control expressions. The first is an initialization expression, executed once at the beginning of the loop. The second is a loop condition, executed before each iteration, which must evaluate to true for the loop to continue. The third is executed at the end of each iteration and is usually used to increment a loop counter.
- range-based for loop iterates over all elements within a range. An array is a range of elements, and a string is a range of characters. The array and vector containers define a range so you can use the range-based for loop to iterate over the elements they contain.
- Any kind of loop may be nested within any other kind of loop to any depth.
- Executing a continue statement within a loop skips the remainder of the current iteration and goes straight to the next iteration, as long as the loop control condition allows it.
- Executing a break statement within a loop causes an immediate exit from the loop.
- A loop defines a scope so that variables declared within a loop are not accessible outside the loop. In particular, variables declared in the initialization expression of a for loop are not accessible outside the loop.
- The array<T,N> container stores a sequence of N elements of type T. An array<> container provides an excellent alternative to using the arrays that are built in to the C++ language.
- The vector<T> container stores a sequence of elements of type T that increases dynamically in size as required when you add elements. You use a vector<> container instead of a standard array when the number of elements cannot be determined in advance.

■Chapter 6: Pointers and References 177

Pointers are important because they provide the foundation for allocating memory dynamically. Pointers can also make your programs more effective and efficient in other ways. Both references and pointers are fundamental to object-oriented programming.

What Is a Pointer? 177 '*' / '&'

Pointer is a variable that can store, point to, an address of another variable, of some piece of data elsewhere in memory.
To use a data item stored at the address contained in a pointer, you need to know the type of the data.
ordinary variable except that the type name has an asterisk following it to indicate that it's a pointer and not a variable of that type.
Define a pointer called pnumber that can store the address of a variable of type long:

long* pnumber {}; // A pointer to type long

This pointer can only store an address of a variable of type long. the statement initializes pnumber with the pointer equivalent of zero, which is a special address that doesn't point to anything. A special pointer value: **nullptr** to specify as the initial value:

long* pnumber {nullptr};

■ **Tip** Should always initialize a pointer when defined. If you cannot give it its intended value yet, initialize the pointer to nullptr.
You can position the asterisk adjacent to the variable name, like this:

long *pnumber {};

Defines the same variable as before. Either notation. Former is common because it expresses the type, "pointer to long," more clearly.

---

all pointer variables for a given platform will have the same size.

The Address-Of Operator 180 '&'

& : unary operator that obtains the address of a variable. Could define: variable, number, and a pointer.
pnumber, initialized with the address of number with these statements:

long number {12345L};
long* pnumber {&number};

&number produces the address of number, pnumber has this address as its initial value.

---

pnumber can store the address of any variable of type long, so you can write the following assignment:
long height {1454L}; // Stores the height of a building
pnumber = &height; // Store the address of height in pnumber
The result of the statement is that pnumber contains the address of height.

Using auto*, you define a variable of a compiler-deduced pointer type:
auto* mynumber {&height}; // mynumber = height = 1454L

Variable declared auto* can be initialized only with a pointer value. Initializing a value of any other type will result in a compiler error.
Accessing the data in the memory location to which the pointer points is fundamental, and you do this using the indirection operator.

The Indirection Operator 181 '*'

Applying the indirection operator, *, to a pointer accesses the contents of the memory location to which it points.
The name indirection operator means the data is accessed "indirectly." Called the **dereference operator** as well.
The process of accessing the data in the memory location pointed to by a pointer is termed ***dereferencing the pointer***.
To access the data at the address contained in the pointer pnumber, you use the expression *pnumber.

Why Use Pointers? 183

•Allocate memory for new variables dynamically—that is, during program execution. This allows a program to adjust its use of memory depending on the input. You can create new variables while your program is executing, as and when you need them. When you allocate new memory, the memory is identified by its address, so you need a pointer to record it.

•You can also use pointer notation to operate on data stored in an array. Equal to array notation, so pick the notation best suited for the occasion. Array notation is more convenient when it comes to manipulating arrays, but pointer notation has its merits as well.

•When Defining your own functions,Ch8, pointers are used extensively to enable a function to access large blocks of data that are defined outside the function.

•Pointers are fundamental to enabling polymorphism to work. Polymorphism is perhaps the most important capability provided by the object-oriented approach to programming. More Ch14.

■ **Note** Last two items in this list apply equally well to reference, a language construct of C++ that is similar to pointers in many ways.

Pointers to Type char 183

const char* pproverb {"A miss is as good as a mile."}; // Do this instead! Always a 'const'

a pointer to a numeric type must be dereferenced: *value -- Wrong: *charType // 1 char of the array. Correct: cout << charType

This approach works nicely as long as you know the exact number of strings at compile time and provided all of them are defined by literals. In real applications, however, you're much more likely to gather a variable number of strings, either from user input or from files. Working with plain character arrays then rapidly becomes cumbersome and very unsafe

Arrays of Pointers 186

```
const char* heros[]{"Iron Man", "Iron Man 2" , "Iron Man 3" };
```

IMPORTANT:Using pointers eliminates waste of memory because each string now occupies only the number of bytes necessary.

Constant Pointers and Pointers to Constants 188

Pointer to Constant / a Constant Pointer / A Constant Pointer to a Constant

using a basic nonarray variable, pointing to just one celebrity.

const char* my_favorite_star{ "Lassie" };

This defines an array that contains const char elements. Can't rename Lassie to Lossie: since a const

my_favorite_star[1] = 'o'; // Error: my_favorite_star[1] is const!

---

overwrite it with a pointer that refers to const char elements:

my_favorite_star = "Mae West"; // my_favorite_star now points to "Mae West"

my_favorite_star = pstars[1]; // my_favorite_star now points to "Clara Bow"

add a second const to protect the content of the my_favorite_star variable:

const char* const forever_my_favorite{ "Oliver Hardy" };

---

constant pointer: address stored in the pointer can't be changed. The contents of that address aren't constant and can be changed.

int data {20};

int* const pdata {&data};

***pdata = 25;** // Allowed, as pdata points to a non-const int

---

const float value {3.1415f};

value is a constant, can initialize a pointer with the address of value:

const float* const pvalue {&value};

■ **Tip** more complex types arises, such as pointers to pointers. **A practical tip** you can read all type names right to left.

float const * const pvalue {&value}; // Read From Right to Left == pvalue is a const pointer to const float(s)

const float* const pvalue {&value}; // This is How it's actually Wrote

So when reading types right to left, you often still have to swap around this const with the element type.

Pointers and Arrays 190

double values[10];
double* pvalue {values};
This will store the address of the values array in the pointer pvalue. Although an array name represents an address, it is not a pointer. You can modify the address stored in a pointer, whereas the address that an array name represents is fixed.

Pointer Arithmetic 191

You can perform arithmetic operations on a pointer to alter the address it contains.
You're limited to **addition** and **subtraction** for modifying the address contained in a pointer, but you can also **compare pointers to produce a logical result**. You can **add an integer** (or an expression that evaluates to an integer) **to a pointer**, and the result is an address. You can **subtract an integer from a pointer**, and that also results in an address.
Can **subtract one pointer from another**, the result is an integer, not an address. No other arithmetic operations on pointers are legal.

Incrementing a pointer by 1 means ***incrementing it by one element*** of the type to which it points. Same with Decrementing:
++pvalue; --or-- pvalue += 1; --pvalue; --or-- pvalue -= 1;

Of course, you can dereference a pointer on which you have performed arithmetic. (There wouldn't be much point to it, otherwise!)
***(pvalue + 1) = *(pvalue + 2);** // pvalue == values[3], this statement is equivalent to the following:
values[4] = values[5]; // ^ Same as

---

pvalue + 1 doesn't change the address in pvalue. ++pvalue and pvalue += n do change pvalue.
parentheses around the expression are essential because the precedence of the indirection operator is higher than + and -.
Error expression *pvalue + 1 adds 1 to the value stored at the address contained in pvalue, so it's equivalent to executing values[3] + 1.
Arithmetic expressions work on Hex, which will cause an error.

The Difference Between Pointers

Subtracting one pointer from another is meaningful only when they are of the same type and point to elements in the same array.
Suppose you define and initialize two pointers like this:
long *pnum1 {&numbers[6]}; // Points to 7th array element
long *pnum2 {&numbers[1]}; // Points to 2nd array element
You can calculate the difference between these two pointers like so:
auto difference {pnum1 - pnum2}; // Result is 5
5 because the difference is measured in elements, not bytes. Only one question remains, though: what will the type of difference be? Clearly, it should be a signed integer type to accommodate for statements such as the following:
auto difference2 {pnum2 - pnum1}; // Result is -5
std::ptrdiff_t, a platform-specific type alias for one of the signed integer types defined by the **#include<cstddef>**. So:
std::ptrdiff_t difference2 {pnum2 - pnum1}; // Result is -5

Comparing Pointers

compare with: **==, !=, <, >, <=,** and **>=**

Using Pointer Notation with an Array Name 193 & Prime Numbers

To Access elements: data[0], data[1], data[2] // Write: *data, *(data + 1), *(data + 2)

// Try dividing the candidate by all the primes we have

```
for (size_t i {}; i < count && isprime; ++i)
{ isprime = trial % *(primes + i) > 0; // False for exact division }
```

■ **Note** You only need to try dividing by primes <= square root of the number in question.

Dynamic Memory Allocation 196

Dynamic memory allocation is allocating the memory you need to store the data you're working with at runtime, Rather than having the amount of memory predefined when the program is compiled.

Dynamically allocated variables can't be defined at compile time, so they can't be named in your source program. The obvious and only place to store this address is in a pointer.

BENEFITS: With the power of pointers and the dynamic memory management tools in C++, writing this kind of flexibility into your programs is quick and easy. Adds memory to your application when it's needed and then release the memory you have acquired when you are done with it. The amount of memory dedicated to an application can increase and decrease as execution progresses.

Variables for which memory is allocated at runtime always have dynamic storage duration.

The Stack and the Free Store 196

Automatic variables are created as the definition executes. Temporary Memory area for automatic variables is allocated in the **stack**.

The stack has a fixed size, determined by your compiler. Usually a compiler option that enables changes to the stack size, it's rare tho.

At the end of the block where an automatic variable is defined, stack memory allocated for the variable is released to reuse.

When you call a function, arguments you pass to the function will be stored on the stack along with the address of the location to return to when execution of the function ends.

Memory that is not occupied by the operating system or other programs that are currently loaded is called the **free store**.

You can request that space be allocated within the free store at runtime for a new variable of any type. You do this using the **new** operator, which returns the address of the space allocated, and you store the address in a pointer. Used with **delete** operator, releases memory that previously allocated with new. Both new and delete are keywords and uses memory very efficiently / allows programs to handle larger problems involving more data than might otherwise be possible. Memory becomes available for reuse by other dynamically allocated variables later in the same program or possibly other programs that are executing concurrently.

Note: Memory is reserved regardless of whether you record its address. It's released automatically as program execution ends.

Using the new and delete Operators 197

Need space for a variable of type double. Must define a pointer of type double* / request the memory is allocated at execution time:

```
double* pvalue {}; // Pointer initialized with nullptr
pvalue = new double; // Request memory for a double variable
```

all pointers should be initialized / Using memory dynamically typically involves using a lot of pointers. It's important that they do not contain spurious values. Always ensure a pointer contains nullptr if it doesn't contain a legal address.

Under extremes: The free store could be full during the request || free store is available isn't large enough to accommodate space requested. Might happen when dealing with large entities: arrays / complicated class objects. When it does happen, the new operator throws an **exception**, which by default will end the program.

```
pvalue = new double {3.14}; // Allocate a double and initialize it
```

You can also create and initialize the variable in the free store and use its address to initialize the pointer when you create it:

```
double* pvalue {new double {3.14}}; // Pointer initialized with address in the free store
```

pvalue to zero (0.0): **double* pvalue {new double {}}; //** Pointer initialized with address in the free store
Pointer initialized with nullptr: **double* pvalue {};**
Frees memory that it occupies using the delete operator: **delete pvalue;**

A pointer that contains such a spurious address is sometimes called a **dangling pointer**. Dereferencing a dangling pointer is a sweet recipe for disaster, so Always reset a pointer when you release the memory to which it points, like this:
delete pvalue; // Release memory pointed to by pvalue
pvalue = nullptr; // Reset the pointer, Which will terminate the program immediately, which is better than the program staggering on in an unpredictable manner with data that is invalid.
■ **Tip** It is perfectly safe to apply delete on a pointer variable that holds the value nullptr. The statement then has no effect at all.

Dynamic Allocation of Arrays 199 USE VECTORS INSTEAD

This allocates space for an array of 100 values of type double and stores its address in data.
double* data {new double[100]}; // Allocate 100 double values
Memory of this array contains uninitialized garbage values. Can initialize dynamic array's elements just like a regular array:
double* data {new double[100] {}}; // All 100 values are initialized to 0.0
int* one_two_three {new int[3] {1, 2, 3}}; // 3 integers with a given initial value
float* fdata{ new float[20] { .1f, .2f }}; // All but the first 2 floats are initialized to 0.0f

To remove the array from the free store when you are done with it, use delete followed with []: no dimensions, simply [].
delete[] data; // Release array pointed to by data

Using a vector<> container, you can forget about memory allocation for elements and deleting it when you are done; it's all taken care of by the container. **Always use std::vector<> to manage dynamic memory for you**.
Multidimensional arrays with multiple dynamic dimensions are not supported by standard C++.

Member Selection Through a Pointer 203 Don't Use this

auto* pdata {new std::vector{}};
But it might just as well be the address of a local object, obtained using the address-of operator.
std::vector<int> data;
auto* pdata = &data;
use the dereference operator, so the statement to add an element looks like this:
(*pdata).push_back(66); // Add an element containing 66
an operator that combines dereferencing a pointer to an object and then selecting a member of the object.
pdata->push_back(66); // Add an element containing 66

-> operator (a minus sign & greater-than character) and is referred to as the **arrow operator** or **indirect member selection operator**. The arrow is much more expressive of what is happening here. You'll be using this operator extensively later in the book.

Hazards of Dynamic Memory Allocation 203

As a C++ developer, a lot of the more serious bugs you deal with often boil down to mismanagement of dynamic memory.
Use the std::vector<> Container!

Dangling Pointers and Multiple Deallocations 204

dangling pointer: pointer that has the address to free store memory that was deallocated by either delete or delete[].
Dereferencing dangling pointers makes you read from or write to memory that might already be allocated to and used by other parts of your program, resulting in all kinds of unpredictable and unexpected results. **Multiple deallocations**, occur when you deallocate an already deallocated (and hence dangling) pointer for a second time using either delete or delete[], is another recipe for disaster.
Always reset a pointer to nullptr after the memory it points to is released.

Allocation/Deallocation Mismatch 204

```
int* single_int{ new int{123} }; // Pointer to a single integer, initialized with 123
int* array_of_ints{ new int[123] }; // Pointer to an array of 123 uninitialized integers
delete[] single_int; // Wrong!
delete array_of_ints; // Wrong!
```

■ Caution Every new must be paired with a single delete; every new[] must be paired with a single delete[]. Any other sequence of events leads to either undefined behavior or memory leaks (discussed next).

Memory Leaks 204

A memory leak occurs when you allocate memory using new or new[] and fail to release it.

Memory leaks are difficult to spot in complex programs, memory may be allocated / should be released in a completely separate part.

Strategy for avoiding memory leaks is to immediately add the delete operation at an appropriate place each time you use the new operator. By no means fail-safe.

Fragmentation of the Free Store 205

If you create and destroy many memory blocks of different sizes, it's possible to arrive at a situation in which the allocated memory is interspersed with small blocks of free memory, none of which is large enough to accommodate a new memory allocation request by your program. The aggregate of the free memory can be quite large, but if all the individual blocks are small (smaller than a current allocation request), the allocation request will fail. Figure 6-6 illustrates the effect of memory fragmentation. The way to avoid fragmentation of the free store then is not to allocate many small blocks of memory. Allocate larger blocks and manage the use of the memory yourself. But this is an advanced topic, well outside the scope of this book.

Golden Rule of Dynamic Memory Allocation 206

■ Tip Never use the operators new, new[], delete, and delete[] directly in day-to-day coding. These operators have no place in modern C++ code. Always use either the std::vector<> container (to replace dynamic arrays) or a smart pointer (to dynamically allocate objects and manage their lifetimes). These high-level alternatives are much, much safer than the low-level memory management primitives and will help you tremendously by instantly eradicating all dangling pointers, multiple deallocations, allocation/ deallocation mismatches, and memory leaks from your programs.

Raw Pointers and Smart Pointers 206

raw pointers: variables of these types contain nothing more than an address. A raw pointer can store the address of an automatic variable or a variable allocated in the free store. A **smart pointer:** object that contains an address, just like raw pointers.

Smart pointers are normally used only to store the address of memory allocated in the free store. A smart pointer does much more than a raw pointer, though.

Smart pointer don't use the delete or delete[] to free the memory, it's released automatically. No more multiple deallocations, allocation/deallocation mismatches, and memory leaks. Using smart pointers, dangling pointers will be a thing of the past as well.

Useful for managing class objects that you create dynamically || working with objects of a class type.

Smart pointer types are defined by templates inside the **#include <memory>** header.

3 types of Smart Pointers: **unique_ptr<T>** && **shared_ptr<T>** && weak_ptr<T>

unique_ptr<T> A unique_ptr<> object owns what it points to, exclusively. Uniqueness is enforced, compiler will never allow you to copy a unique_ptr<>.

shared_ptr<T> objects allow shared ownership of an object in the free store. At any given moment, the number of shared_ptr<> objects that contain a given address in time is known by the runtime, called **reference counting**. The reference count for a shared_ptr<> containing a given free store address is incremented each time a new shared_ptr<> object is created containing that address / it's decremented when a shared_ptr<> containing the address is destroyed or assigned to point to a different address. When there's no shared_ ptr<> objects containing a given address, the reference

count will have dropped to zero, and the memory for the object at that address will be released automatically. All shared_ptr<> objects that point to the same address have access to the count of how many there are.

weak_ptr<T> Use of weak_ptr<> objects is to avoid / break so-called reference cycles with shared_ptr<> objects. Reference cycle is where a shared_ptr<> inside an object x points to some other object y that contains a shared_ptr<>, which points back to x, meaning neither x nor y can be destroyed. Another use is the implementation of object caches.

copying a unique_ptr<> is not possible, you can "move" the address stored by one unique_ptr<> object to another by using the **std::move()** function. After this move operation the original smart pointer will be empty again.

Using unique_ptr<T> Pointers 208

--When the unique_ptr is destroyed, so is the value to which it points.

--One common use for a **unique_ptr<>** is to hold something called a **polymorphic pointer**: a pointer to a dynamically allocated object, can be of any number of related class types. Useful with: class objects and polymorphism.

In the (not so) old days, you had to create and initialize a unique_ptr<T> object like this:

```
std::unique_ptr<double> pdata {new double{999.0}};
```

std::make_unique<>() function template (introduced by C++14). To define pdata, normally use the following:

```
std::unique_ptr<double> pdata { std::make_unique<double>(999.0) };
```

To save you some typing, you'll probably want to combine this syntax with the use of the auto keyword:

```
auto pdata{ std::make_unique<double>(999.0) };
```

The big difference is that you no longer have to worry about deleting the double variable from the free store.

You can access the address that a smart pointer contains by calling its get() function. Here's an example:

```
std::cout << std::hex << std::showbase << pdata.get() << std::endl;
```

unique pointer that points to an array as well. The older syntax to do this looks as follows:

```
const size_t n {100}; // Array size
std::unique_ptr<double[]> pvalues {new double[n]}; // Dynamically create array of n elements
```

As before, we recommend you always use **std::make_unique<T[]>()** instead:

```
auto pvalues{ std::make_unique<double[]>(n) }; // Dynamically create array of n elements
```

■ **Tip** Use a vector container instead of a unique_ptr because it's far more powerful / flexible than the smart pointer.

reset the pointer contained in a unique_ptr<>, or any type of smart pointer for that matter, by calling its **reset()** function:

```
pvalues.reset(); // Address is nullptr
```

Using shared_ptr<T> Pointers 211

You can define a shared_ptr<T> object in a similar way to a unique_ptr<T> object:

```
std::shared_ptr<double> pdata {new double{999.0}};
```

You can also dereference it to access what it points to or to change the value stored at the address:

```
*pdata = 8888.0;
std::cout << *pdata << std::endl; // Outputs 8888
*pdata = 8889.0;
std::cout << *pdata << std::endl; // Outputs 8889
```

More efficient: using the make_shared<T>() function, defined in memory header, to create a smart pointer of type shared_ptr<T>:

```
auto pdata{ std::make_shared<double>(999.0) }; // Points to a double variable
```

initialize a shared_ptr<T> with another when you define it:

```
std::shared_ptr<double> pdata2 {pdata};
```

Understanding References 214

Reference: Alias for another variable. Like pointer, it refers to something else in memory, but there are a few crucial differences.

Unlike a pointer, you cannot declare a reference and not initialize it. Because a reference is an alias, the variable for which it is an alias must be provided when the reference is initialized. Also, a reference cannot be modified to be an alias for something else.

Once a reference is initialized as an alias for some variable, it keeps referring to that same variable for the remainder of its lifetime.

Defining References 214

```
double data {3.5};
double& rdata {data};        // Defines a reference to the variable data
rdata += 2.5;  // This increments data by 2.5.
double* pdata1 {&rdata};    // pdata1 == pdata2
double* pdata2 {&data};
```

You must dereference the pointer to access the variable to which it points. With a reference, there is no need for dereferencing; it just

doesn't apply. Reference is like a pointer, already dereferenced, although it also can't be changed to reference something else.

```
double other_data = 5.0;      // Create a second double variable called other_data
rdata = other_data;           // Assign other_data's current value to data (through rdata)
```

```
const double& const_ref{ data };
const_ref *= 2; // Illegal attempt to modify data through a reference-to-const
```

Using a Reference Variable in a Range-Based for Loop 216

Using a reference in a range-based for loop is efficient when you are working with collections of objects. Copying objects can be expensive on time, so avoiding copying by using a reference type makes your code more efficient.

```
const double F2C {5.0/9.0}; // Fahrenheit to Celsius conversion constant
for (auto& t : temperatures) // Reference loop variable
 t = (t - 32.0) * F2C;
```

Without needing to modify 't'

```
for (const auto& t : temperatures)
 std::cout << std::setw(6) << t;
```

Summary 217

- A pointer is a variable that contains an address. A basic pointer is referred to as a **rawpointer**.
- You obtain the address of a variable using the address-of operator, **&**.
- **Dereference(indirection) operator**: To refer to the value pointed to by a pointer, you use the *****.
- You access a member of an object through a pointer or smart pointer using the **indirect member selection operator, ->**.
- Can add or subtract integer values from the address stored in a raw pointer. It's as though the pointer refers to an array & pointer is altered by the number of array elements specified by the integer value. You cannot perform arithmetic with a smart pointer.
- The **new** and **new[] operators** allocate a block of memory in the free store—holding a single variable and an array, respectively and return the address of the memory allocated.
- Use the **delete** or **delete[] operator** to release a block of memory that you've allocated previously using new or, respectively, the new[] operator. Don't need to use these operators when the address of free store memory is stored in a smart pointer.
- Low-level dynamic memory manipulation is synonymous for a wide range of serious hazards such as dangling pointers, multiple deallocations, deallocation mismatches, memory leaks, and so on. **Never use the low-level new/new[] and delete/delete[] operators directly**. Containers(std::vector<> in particular) and smart pointers are nearly always the smarter choice!

- **Smart pointer:** Object that can be used like a raw pointer. By default, used only to store free store memory addresses.
- 2 Commonly used varieties of smart pointers. There can only ever be one type **unique_ptr<T>** pointer in existence that points to a given object of type T, but there can be multiple **shared_ptr<T>** objects containing the address of a given object

of type T. The object will then be destroyed when there are no shared_ptr<T> objects containing its address.

- **Reference:** Alias for a variable that represents a permanent storage location.
- Use a reference type for the loop variable in a range-based for loop to allow the values of the elements in the range to be modified.

■Chapter 7: Working with Strings 219

A Better Class of String 219

#include<cstring> capabilities for joining strings, searching a string, and comparing strings. Using C-style strings is unsafe

#include<string> defines the std::string type, which is much easier to use than a null-terminated string. String type defined by a class (more precise, class template), !fundamental type. Type string is a compound type, which is a type that's a composite of several data items that are ultimately defined in terms of fundamental types of data. Next to the characters that make up the string it represents, a string object contains other data as well, such as number of characters in the string.

Defining string Objects 220

std::string empty; // An empty string

You can initialize a string object with a string literal when you define it:

std::string proverb {"Many a mickle makes a muckle."};

■ Note Convert a std::string object to a C-style string using 2 methods. 1st Calling its c_str() member function (short for C-string):

const char* proverb_c_str = proverb.c_str();

This conversion results in a C-string of type const char*. Because it's const, this pointer cannot be used to modify the characters of the string, only to access them. 2nd data() function, evaluates to a non-const char* pointer1.

char* proberb_data = proverb.data();

You should convert to C-style strings only when calling legacy C-style functions.

NOTE ALWAYS use std::string objects because these are far safer and more convenient than plain char arrays.

obtain the length of the string using **length() function**, takes no arguments. Length never includes the string termination character:

std::cout << proverb.length(); // Outputs 29

---

initial sequence from a string literal, for instance:

std::string part_literal { "Least said soonest mended.", 5 }; // "Least"

---

initialize a string with any number of instances of a given character:

std::string sleeping(6, 'z');

Use an existing string object to provide the initial value:

std::string sentence {proverb};

Use a pair of index values to identify part of an existing string and use that to initialize a new string object:

std::string phrase {proverb, 0, 13}; // Initialize with 13 characters starting at index 0

---

output string objects just like C-style strings. Extraction from cin is also supported for string objects:

std::string name;
std::cout << "enter your name: ";
std::cin >> name; // Pressing Enter ends input

Reads characters up to the first whitespace character, which ends the input process. The read is stored in the string object, name.

Operations with String Objects 223

std::string adjective {"hornswoggling"}; // Defines adjective
std::string word {"rubbish"}; // Defines word
word = adjective; // Modifies word
adjective = "twotiming"; // Modifies adjective

Concatenating Strings 247

join strings using the addition operator; (concatenation). Concatenate objects like this:
std::string description {adjective + " " + word + " whippersnapper"};

std::string compliment("~~~ What a beautiful name... ~~~");
sentence.append(compliment, 3, 22); // Appends " What a beautiful name"
sentence.append(3, '!'); // Appends "!!!"

Concatenating Strings and Characters 249

Another option, just to illustrate the possibilities, is to use the following two statements:
sentence += first + ' ' + second;
sentence += '.';

Concatenating Strings and Numbers 250

An important limitation in C++ is that you can only concatenate std::string objects with either strings or characters.
in C++, have to explicitly convert these values to strings yourself. Values of fundamental numeric types, the easiest by far is to use the std::to_ string() family of functions, defined in **#include<string>**:

const std::string result_string{ "The result equals: "};
double result = 3.1415;
std::cout << (result_string + std::to_string(result)) << std::endl;

Accessing Characters in a String 228

Access Characters the same as in an array. Starts with sentence[0].
changes all the characters in sentence to uppercase:

for (size_t i {}; i < sentence.length(); ++i)
sentence[i] = std::toupper(sentence[i]);

A string object is a range, so you could also do this with a range-based for loop:

for (auto& ch : sentence)
ch = std::toupper(ch);

Specify ch as a reference allows the character in the string to be modified within the loop. Both loops need cctype header.

Accessing Substrings 230

std::string phrase {"The higher the fewer."};
std::string word1 {phrase.substr(4, 6)}; // "higher"
std::string word2 {phrase.substr(4, 100)}; // "higher the fewer."

■ Caution As before, substrings are always specified using their begin index and length, not using their begin and end indexes. Keep this in mind, especially when migrating from languages such as JavaScript or Java!

Comparing Strings 230

> >= < <= == !=

Compared char by char until it contains different characters or the end of either or both operands is reached.
Technically called, **lexicographical comparison**, meaning strings are ordered in the same manner as they are in a dictionary.

This Works Well for Comparing Strings:

```
std::cout << word1 << (word1 < word2? " comes " : " does not come ")  // Using the (a<b? “c”:”d”)
 << "before " << word2 << '.' << std::endl;
```

■ **Tip** Most Standard Library types offer a swap() function. All container types (such as std::vector<> and std::array<>), std::optional<>, all smart pointer types, and many more. The std namespace defines a nonmember function template, used to the same effect:

```
std::swap(names[i], names[i-1]);
```

works for fundamental types such as int or double as well. Have to include the utility header for std::swap() function template.

compare() Function 237

expression that calls compare() for a string object, word, to compare it with a string literal:

```
word.compare("and")
```

■ Caution A common mistake is to write an if statement of the form if (word.compare("and")), assuming this condition will evaluate to true if word and "and" equal. But the result, of course, is precisely the opposite. For equal operands, compare() returns zero. And zero, as always, converts to the Boolean value false. To compare for equality, you should use the == operator instead.

■ Note You have seen that the compare() function works quite happily with different numbers of arguments of various types. The same was true for the append() function we briefly mentioned earlier. What you have here are several different functions with the same name. These are called overloaded functions, and you’ll learn how and why you create them in the next chapter.

Comparisons using substr() 237

check whether two substrings are equal, you could write a test as follows:

```
std::string text {"Peter Piper picked a peck of pickled pepper."};
std::string phrase {"Got to pick a pocket or two."};
for (size_t i{}; i < text.length() - 3; ++i)
 if (text.substr(i, 4) == phrase.substr(7, 4))
 std::cout << "text contains " << phrase.substr(7, 4)
 << " starting at index " << i << std::endl;
```

Searching Strings 237

A string object has a find() function that finds the index of a substring within it. You can also use it to find the index of a given character. The substring you are searching for can be another string object or a string literal.

```
std::string sentence {"Manners maketh man"};
 std::string word {"man"};
 std::cout << sentence.find(word) << std::endl; // Outputs 15
 std::cout << sentence.find("Ma") << std::endl; // Outputs 0
 std::cout << sentence.find('k') << std::endl; // Outputs 10
 std::cout << sentence.find('x') << std::endl; // Outputs std::string::npos
```

use npos to check for a search failure with a statement such as this:

```
if (sentence.find('x') == std::string::npos)
 std::cout << "Character not found" << std::endl;
```

Searching within Substrings 261

```
while ((index = text.find(word, index,3)) != std::string::npos)       // string.find(string, IndexStart, Char#inString)
 { ++count;      index += word.length();        }
 std::cout << "Your text contained " << count << " occurrences of \""  << word << "\"." << std::endl;
```

Searching for any Characters 243

A number of different characters such as spaces, commas, periods, colons, and so on. **find_first_of()** for a string object does:

```
std::string text {"Smith, where Jones had had \"had had\", had had \"had\"."  " \"Had had\" had had the examiners' approval."};
std::string separators {" ,.\""};
std::cout << text.find_first_of(separators) << std::endl; // Outputs 5
```

find the first vowel in text, for example, you could write this:

```
std::cout << text.find_first_of("AaEeIiOoUu") << std::endl; // Outputs 2
```

Find the last vowel in text, use: **find_last_of() function**

```
std::cout << text.find_last_of("AaEeIiOoUu") << std::endl; // Outputs 92
```

Find characters not in the given set. **find_first_not_of()** & **find_last_not_of()** Position the first character that isn't a vowel:

```
std::cout << text.find_first_not_of("AaEeIiOoUu") << std::endl; // Outputs 0
```

```
std::cout << "Enter some text terminated by *:\n";
 std::getline(std::cin, text, '*');                    //'*' || /*Variable*/Separators = {" ,.?;:!*"} //All Gramatical Separators(Delimiter)
```

Searching a String backwards 265

find() : searches forward through a string, from the beginning or index. **rfind()** : searches a string in reverse.

```
std::string sentence {"Manners maketh man"};
std::string word {"an"};
std::cout << sentence.rfind(word) << std::endl; // Outputs 16
std::cout << sentence.rfind("man") << std::endl; // Outputs 15
std::cout << sentence.rfind('e') << std::endl; // Outputs 11
```

As with find(), you can supply an extra argument to rfind() to specify the starting index for the backward search, and you can add a third argument when the first argument is a C-style string.

3RD argument specifies the number of characters the C-style string that are to be taken as the substring for which you're searching.

Modifying a String 266

You can also insert a string into a string object at a given index or replace a substring.

insert() : To insert a string, and **replace()** : to replace a substring in a string.

Inserting a String 267

```
std::string phrase {"We can insert a string."};
std::string words {"a string into "};
phrase.insert(14, words);
```

insertion of a substring of a string object into another string object. Two extra arguments to insert(): one specifies the index of the first character in the substring to be inserted, and the other specifies the number of characters in the substring. Here's an example:

```
phrase.insert(13, words, 8, 5);
```

Inserts the first five characters of " into something" into phrase preceding the character at index 13. Adds " into"(5 Chars) at Index 13

```
phrase.insert(13, " into something", 5);
```

Inserts a sequence of identical characters:Adds 7 *'s at Index 16

```
phrase.insert(16, 7, '*');
```

Replacing a Substring 248

std::string text {"Smith, where Jones had had \"had had\", had had \"had\"."};

You can replace "Jones" with a less common name with this statement:

text.replace(13, 5, "Gruntfuttock"); // String named 'text' // replace (start=13, replace 5 Chars , with string "Gruntfuttock"

Removing Characters from a string 248

erase(indexPosition, length**)** : Substring to be erased by the of the first character and the length.

text.erase(0, 6); // Remove the first 6 characters from string text

text.erase(5); // Removes all but the first 5 characters

text.erase(); // Removes all characters **CAREFUL!** **text.clear();** // Removes all characters **CAREFUL!**

■ Caution Yet another common mistake is to call erase(i) with a single argument i in an attempt to remove a single character at the given index i. The effect of this call, however, is quite different. It removes all characters starting from the one at index i all the way until the end of the string! To remove a single character at index i, you should use **erase(i,1)** instead.

std::string vs. std::vector<char> 248

- A string has a push_back() function to insert a new character at the end of the string (right before the termination character). It's not used that often, though, as std::string objects support the more convenient **+= syntax** to append characters.
- A string has an **at()** function that, unlike the [] operator, performs bounds checking for the given index.
- A string has a **size()** function, which is an alias for length(). Reference of the "length of a string" rather than the "size of a string."
- A string offers **front()** / **back()** to access its first and last characters (not counting the null termination character).
- A string supports a range of **assign()** functions to reinitialize it. s.assign(3, 'X'), for instance, reinitializes s to "XXX", and s.assign("Reinitialize", 2, 4) overwrites the contents of the string object s with "init".

std::string is so much more than a simple std::vector<char>. Has additional functions for common string manipulations such as concatenation, substring access, string searches and replacements, and so on. And of course, a std::string is aware of the null character that terminates its char array & how to take this into account in members such as **size()**, **back()**, and **push_back()**.

Converting Strings into Numbers 248 with #include<string>

std::to_string() to convert numbers into strings.

std::stoi() "string to int" converts strings such as "123" and "3.1415" into the numbers

#include<string> offers **stol(), stoll(), stoul(), stoull(), stof(), stod(), and stold()**, to convert a string into a value of, respectively, type long, long long, unsignedlong, unsigned longlong, float, double, and long double.

String Streams 249

rather than outputting characters directly to the computer screen, gathers them all into a string object. You can then retrieve this string for further processing. **std::stringstream** defined in **#include<sstream>** You use it in the same manner as std::cout

replace std::cout with a variable of type std::stringstream **str()** function. Using that function you obtain a std::string object containing all the characters the stream has accumulated up to that point.

std::stringstream ss; // Create a new string stream
std::string s{ ss.str() };

Strings of International Characters 250

- **std::wstring** objects that contain strings of characters of type wchar_t—the wide-char type, built into your C++ implementation.
- **std::u16string** objects that store strings of 16-bit Unicode characters, which are of type char16_t.
- **std::u32string** objects that contain strings of 32-bit Unicode characters, which are of type char32_t.

The string header defines all these types.

Strings of wchar_t Characters 251

string literals containing characters of type wchar_t between double quotes, but with L prefixed to distinguish them from string literals containing char characters. Thus, you can define and initialize a wstring variable like this:

std::wstring saying {L"The tigers of wrath are wiser than the horses of instruction."};

To output wide strings, you use the wcout stream. Here's an example:

std::wcout << saying << std::endl;

Other functionalities—such as the to_wstring() function and the wstringstream class—just take an extra w in their name but entirely equivalent. Just remember specify the L prefix with string and character literals when working with wstring objects.

Objects That Contain Unicode Strings 252

std::u16string question {u"Whither atrophy?"}; // char16_t characters
std::u32string sentence {U"This sentance contains three errars."}; // char32_t characters

Raw String Literals 252

The basic form of a raw string literal is thus R"(...)". The parentheses themselves are not part of the literal. Any of the types of literal you have seen can be specified as raw literals by adding the same prefix as before—L, u, U, or u8—prior to the R.

Within a raw string literal, no escaping is required. This means you can simply copy and paste, for instance, a Windows path sequence into them or even an entire play of Shakespeare complete with quote characters and line breaks.

Summary 254

--Use the string type that's defined in the Standard Library. The string type is much easier and safer to use than C-style strings, so it should be your first choice when you need to process character strings.

- The **std::string** type stores a character string.
- Like std::vector<char>, it is a dynamic array—meaning it will allocate more memory when necessary.
- Internally, the **terminating null character** is still present in the array managed by a std::string object, but only for compatibility with legacy and/or C functions. As a user of std::string, you normally do not need to know that it even exists. All string functionality transparently deals with this legacy character for you.
- You can store string objects in an array or, better still, in a sequence container such as a **vector**.
- Access/Modify individual characters in a string object using an index between square brackets., which start at 0.
- You can use the + operator to concatenate a string object with a string literal, a character, or another string object.
- To concatenate a value of one of the fundamental numeric types, such as for instance an int or a double, you must first convert these numbers into a string. Your easiest, least flexible option is **std::to_string()** function template defined in the string header.
- Objects of type string have functions to search, modify, and extract substrings.

- string header offers functions like **std::stoi()** & **std::stod()** to convert strings to values of numeric types such as int and double.
- **std::stringstream-** To write numbers to string, or read from string. Use string streams same as std::cout and std::cin.
- Objects of type wstring contain strings of characters of type **wchar_t**.
- Objects of type u16string contain strings of characters of type **char16_t**.
- Objects of type u32string contain strings of characters of type **char32_t**.

How the Return Statement Works 263

return expression; expression must evaluate to a value of the type that is specified for the return value in the function header or must be convertible to that type. It can include function calls and can even include a call of the function in which it appears

If void, no expression can appear in a return statement. It must be written simply as follows: return;

the compiler processes a source file from top to bottom. Of course, you could revert to the original version, but in some situations this won't solve the problem. There are two important issues to consider: • As you'll see later, a program can consist of several source files. The definition of a function that is called in one source file may be contained in a separate source file. • Suppose you have a function A() that calls a function B(), which in turn calls A(). If you put the definition of A() first, it won't compile because it calls B(); the same problem arises if you define B() first because it calls A(). Naturally, there is a solution to these difficulties. You can declare a function before you use or define it by means of a **function prototype**. Functions that are defined in terms of each other, such as the A() and B() functions we described just now, are called mutually recursive functions.

Function Prototypes 263

A function prototype is sometimes referred to as a function declaration. A function can be compiled only if the call is preceded by a function declaration in the source file. The definition of a function also doubles as a declaration,

function prototype for the power() function as follows:

double power(double value, int exponent); //Do this

This works just as well. The compiler only needs to know the type each parameter is, so you can omit

the parameter names from the prototype, like this:

double power(double, int); //Don't Do this

Always write prototypes for each function that is defined in a source file—with the exception of main(), of course, which never requires a prototype. Specifying prototypes near the start of the file removes the possibility of compiler errors arising from functions not being sequenced appropriately. It also allows other programmers to get an overview of the functionality of your code.

Passing Arguments to a Function 264

Function arguments should correspond in type and sequence to the Parameter List in the function definition.

2 ways arguments are passed to functions, **pass-by-value** and **pass-by-reference**.

Pass-by-Value 265

Copies of the arguments are created, and these copies are transferred to the function.

Passing a Pointer to a Function 266

Use a pointer : To modify values in the calling function.

Passing an Array to a Function 267

First, passing the address of an array is an efficient way of passing an array to a function. Passing all the array elements by value would be time-consuming because every element would be copied.

Functions deal with a copy of the original array variable, the code in the body of the function can treat a parameter that represents an array as a pointer in the fullest sense, including modifying the address that it contains. This means you can use the power of pointer notation in the body of a function for parameters that are arrays.

Const Pointer Parameters 270

Passing a Multi-Dimensional Array to a Function 271

Pass-by-Reference 273

Using &references improves performance with objects such as type string, it does not copy the object.

References Vs. Pointers 273

■ Tip Always declare variables as const whenever their values are not supposed to change anymore after initialization.
In function signatures: Always declare pointer / reference parameters with const, if function doesn't modify corresponding arguments.
Second, reference-to-const parameters allow your functions to be called with const values.
•If you want to allow nullptr arguments, you cannot use references. Conversely, pass-by-reference can be seen as a contract that a value is not allowed to be null.
Note already, instead of representing optional values as a nullable pointer, you may also want to consider the use of std::optional<>.
•Using reference parameters allows for a more elegant syntax but may mask that a function is changing a value. Never change an argument value if it is not clear from the context—from the function name, for instance—that this will happen.
•Because of the potential risks, some coding guidelines advise to **never use reference to-non-const parameters** and advocate to instead **always use pointer-to-non-const parameters**. Personally, we would not go that far. But Choosing descriptive function and parameter names is always a good start to make a function's behavior more predictable.
•Passing arguments by **reference-to-const** is generally preferred over passing a pointer-to-const value. Very, very common.

void change_it_by_pointer(double* pit){ *pit += 10.0; // This modifies the original double}
void change_it_by_reference(double& pit){ pit += 10.0; /* This modifies the original double as well! */ }

Input Vs. Output Parameters 275

In principle, a parameter can act as both an input and an output parameter. Such a parameter is called an input-output parameter. A function with such a parameter, in one way or another, first reads from this parameter, uses this input to produce some output, and then stores the result into the same parameter. It is generally better to avoid input-output parameters, though, even if that means adding an extra parameter to your function. Code tends to be much easier to follow if each parameter serves a single purpose—a parameter should be either input or output, not both.
■ **Tip** Input parameters should be references-to-const. Smaller values, notably fundamental types, are to be passed by value. Use reference-to-non-const only for output parameters, and even then you should often consider returning a value instead.

Passing Arrays by Reference 277

double average10(const std::array<double,10>& values)

References and Implicit Conversions 279

String Views: The New Reference-to-const-string 280

std::string_view , a type defined in the **#include<string_view>**, C++17. Values of this type will act analogously to values of type const std::string- mind the const! Major difference: the strings they encapsulate can't be modified through public interface.
initializing and copying a string_view is very cheap.
■ Tip Always use the type *std::string_view* instead of const std::string& for input string parameters. While there is nothing wrong with using const std::string_view&, you might as well pass std::string_view by value because copying these objects is cheap.

Using String views function parameters 304

Using **std::string_view**, HugeTextArray isn't copied when passing it to find_words(), it would be if you'd use const std::string&.

Default Argument Values 283

void show_error(string_view message) { std::cout << message << std::endl;}
You specify the default argument value like this:
void show_error(string_view message = "Program Error");
show_error(); // Outputs "Program Error"

Multiple Default Parameter Values 283

Five ways to call this function: Specify all five arguments, or you can omit the last one, the last two, the last three, or the last four.

Arguments into main() 285 int main(int argc, char* argv[]) Command Line Arguments:::

define main() so that it accepts arguments that are entered on the command line when the program executes. The parameters you can specify for main() are standardized; either you can define main() with no parameters or define main() in the following form:

int main(int argc, char* argv[])
{ // Code for main()...}

---

argc = a count of the number of string arguments that were found on the command line.
argv = array of pointers to the command-line arguments, including the program name. The array type implies that all command-line arguments are received as C-style strings.

Put anything in command-line arguments. This program shows how you access the command-line arguments:

```
// Ex8_12.cpp // Program that lists its command line arguments
#include <iostream>
int main(int argc, char* argv[])
{ for (int i {}; i < argc; ++i)
 std::cout << argv[i] << std::endl;
}
```

Including the program name, Command-line arguments can be anything at all. EXAMPLE: file names to a file copy program / the name of a person to search for in a contact file. Can be anything that is useful to have entered when program execution is initiated.

Returning Values from a Function 286

pitfalls when you are returning a pointer or a reference.

Returning a Pointer 286

the variable pointed to must still be in scope after the return to the calling function. This implies the following absolute rule:

■ Caution Never return the address of an automatic, stack-allocated local variable from a function

---

```
x int* larger(int* a, int* b)
{ if (*a > *b)
 return a; // OK
 else   return b; // OK}
```

1st Find the minimum and another to adjust the values by any given amount. Here's a definition for the first function:

```
const double* smallest(const double data[], size_t count)
{ if (!count) return nullptr; // There is no smallest in an empty array
 size_t index_min {};
 for (size_t i {1}; i < count; ++i)
 if (data[index_min] > data[i])
 index_min = i;
 return &data[index_min];
```

```
}
```

A function to adjust the values of array elements by a given amount looks like this:

```
double* shift_range(double data[], size_t count, double delta)
{ for (size_t i {}; i < count; ++i)
 data[i] += delta;
 return data; }
```

Combine using this with the previous function to adjust the values in an array, samples, so that all the elements are non-negative:

```
const size_t count {std::size(samples)}; // Element count
shift_range(samples, count, -(*smallest(samples, count))); // Subtract min from elements
```

---

function to find the maximum:

```
const double* largest(const double data[], size_t count)
{ if (!count) return nullptr; // There is no largest in an empty array
 size_t index_max {};
 for (size_t i {1}; i < count; ++i)
 if (data[index_max] < data[i])
 index_max = i;
 return &data[index_max];
}
```

function that scales the array elements by dividing by a given value:

```
double* scale_range(double data[], size_t count, double divisor)
{ if (!divisor) return data; // Do nothing for a zero divisor
for (size_t i {}; i < count; ++i)
 data[i] /= divisor;
 return data;
}
```

■ Caution Never return a reference to an automatic local variable in a function.

Returning a Reference 290

reference return types become essential when you are creating your own data types using classes.

Returning vs. Output Parameters 291

■ **Tip** Prefer returning values over output parameters. This makes function signatures and calls much easier to read.

Return Type Deduction 292

Just like using auto, the compiler deduces the return type of a function from its definition.

```
auto getAnswer() { return 42; }          // getAnswer is an int type
```

If it's clear enough from context, what's type, and the clarity of code doesn't matter, return type deduction can be practical.

■ Note Another context where return type deduction can be practical is to specify the return type of a function template.

Return Type Deduction & References 292

■ **Caution** auto never deduces a reference type--always to a value type. Even assigning a reference to auto, the value still gets copied. The copy isn't const, unless use const auto. To have the compiler deduce a reference type, you can use **auto&** or **const auto&**.

Working with Optional Values 293

You may Often encounter optional input arguments or functions that return a value if nothing went wrong.

```
int find_last_in_string(string_view string, char char_to_find, int start_index);
```

Use **std::optional<>**, a utility we believe can help make your function declarations much cleaner and easier to read.

std::optional 294

Used to predefine a variable statically or dynamically. An optional int can be explicitly declared with optional<int> as follows:

optional<int> find_last_in_string(string_view string, char to_find, optional<int> start_index);
optional<int> read_configuration_override(string_view fileName, string_view overrideName);

Static Variables 297

Static Variables- prevents the potential problem that any function in the file can modify the variable
A static variable is created the first time its definition is executed.
Specify any type of variable as static, and use a static variable for anything that you need Globally.

Inline Functions 297

Keyword **inline** copies the code of a function for runtime optimization.

inline int larger(int m, int n) { return m > n ? m : n; }

Suggests to the compiler that it can replace calls with inline code. the definition of an inline function usually appears in a header file rather than in a source file, and the header is included in each source file that uses the function.

Function Overloading 298

Function overloading allows several functions in a program with the same name as long as they all have a parameter list that is different from each other. You might want these operations to work with arrays of different types such as int[], double[], float[], or even string[]. Ideally, all such functions would have the same name, smallest() or largest().

Two functions with the same name have different signatures if at least one of the following is true:

- The functions have different numbers of parameters. || • At least one pair of corresponding parameters is of different types.

Overloading and Pointer Parameters 300

int largest(float* pValues, size_t count); // Prototype 2

A parameter type int* is treated in the same of int[]. Hence, the following prototype declares the same function as Prototype 1 earlier:

int largest(int values[], size_t count); // Identical signature to prototype 1

Overloading and Reference Parameters 301

void do_it(std::string number); // These are not distinguishable...
void do_it(std::string& number); // ...from the argument type

You can't overload a function with a parameter type data_type with a function that has a parameter type data_type&. The compiler cannot determine which function you want from the argument.

Overloading and const Parameters 302

the following prototypes are not distinguishable:

long larger(long a, long b);
long larger(const long a, const long b);

These arguments are passed by value, meaning:a copy of each argument is passed into the function, and original is protected from modification by the function.

Overloaded functions are different if one has a parameter of type type* and the other has a parameter of const type*.

The compiler won't pass a pointer-to-const value to a function in which the parameter is a pointerto-non-const.

Overloading and Default Argument Values 304

Default parameter values for overloaded functions can sometimes affect the compiler's ability to distinguish one call from another.

Recursion 305

A function can call itself,functions that contain a call to itself is referred to as a **recursive function**. EXAMPLE: a function fun1() calls another function fun2(), which in turn calls fun1(). In this case, fun1() and fun2() are also called **mutually recursive functions**.

Recursion can be used for many different problems. Compilers are sometimes implemented using recursion because language syntax is usually defined in a way that lends itself to recursive analysis. Data that is organized in a tree structure is an example.

Basic Examples 306 Factorial of Positive Numbers 'n' / recursive version of the power() function / Quicksort Algorithm / The main() Function / The extract_words() Function / swap() Function / sort() Function / max_word_length Function / show_words Function /

Factorial of Positive Numbers 'n' 306

long long factorial(int n) { if (n == 1) return 1LL; return n * factorial(n - 1); }

This is often the example given to show recursion, but it is an inefficient process. It would certainly be much faster to use a loop.

Recursive version of the power() function // Recursive version of function for x to the power n, n positive or negative

```
double power(double x, int n);                  // Function Prototype
int main()
{ for (int i {-3}; i <= 3; ++i) // Calculate powers of 8 from -3 to +3
 std::cout << std::setw(10) << power(8.0, i);
 std::cout << std::endl; }
        // Recursive function to calculate x to the power n
double power(double x, int n)
{ if (n == 0) return 1.0;
 else if (n > 0) return x * power(x, n - 1);
 else /* n < 0 */ return 1.0 / power(x, -n); }
```

Generally better to use a different approach, such as a loop. However, in spite of the overhead, using recursion can often simplify the coding considerably. Sometimes this gain in simplicity can be well worth the loss in efficiency that you get with recursion.

Recursive Algorithms 307

Recursion is often favored in sorting & merging operations.

Sorting data can be a recursive process in which the same algorithm is applied to smaller and smaller subsets of the original data.

Quicksort Algorithm is a well-known method of Recursive sorting.

Quicksort Algorithm

choosing an arbitrary word in the sequence and arranging the other words so that all those "less than" the chosen word precede it and all those "greater than" the chosen word follow it. The following type alias will help to make the code easier to read:

```
using Words = std::vector<std::shared_ptr<std::string>>;
```

The main() Function

```
int main()
```

```
{Words words;
std::string text; // The string to be sorted
const auto separators{" ,.!?\"\n"}; // Word delimiters
std::cout << "Enter a string terminated by *:" << std::endl;          // Read the string to be searched from the keyboard
 getline(std::cin, text, '*');
 extract_words(words, text, separators);
 if (words.empty()) { std::cout << "No words in text." << std::endl;
 return 0;        }
 sort(words); // Sort the words
 show_words(words); // Output the words
}
```

The extract_words() Function

```
void extract_words(Words& words, std::string_view text, std::string_view separators)
{ size_t start {text.find_first_not_of(separators)}; // Start 1st word
 size_t end {}; // Index for the end of a word
 while (start != std::string_view::npos)
 { end = text.find_first_of(separators, start + 1); // Find end separator
 if (end == std::string_view::npos) // End of text?
 end = text.length(); // Yes, so set to last+1
 words.push_back(std::make_shared<std::string>(text.substr(start, end - start)));
 start = text.find_first_not_of(separators, end + 1); // Find next word      }
}
```

swap() Function

```
void swap(Words& words, size_t first, size_t second)
{ auto temp{words[first]};
words[first] = words[second];
 words[second] = temp;       }
```

sort() Function

```
void sort(Words& words, size_t start, size_t end)
{ // start index must be less than end index for 2 or more elements
 if (!(start < end))  return;
 // Choose middle address to partition set
 swap(words, start, (start + end) / 2); // Swap middle address with start
 // Check words against chosen word
 size_t current {start};
 for (size_t i {start + 1}; i <= end; i++)
 {  if (*words[i] < *words[start]) // Is word less than chosen word?
 swap(words, ++current, i);   // Yes, so swap to the left  }
 swap(words, start, current);          // Swap chosen and last swapped words
 if (current > start) sort(words, start, current - 1); // Sort left subset if exists
 if (end > current + 1) sort(words, current + 1, end); // Sort right subset if exists
}
```

// Sort strings in ascending sequence

```
void sort(Words& words)
{   if (!words.empty())
 sort(words, 0, words.size() - 1);      }
```

max_word_length Function

This is a helper function that is used by the show_words() function:

```
size_t max_word_length(const Words& words)
{ size_t max {};
 for (auto& pword : words)
```

```
 if (max < pword->length()) max = pword->length();
 return max;
}
```

show_words Function

```
void show_words(const Words& words)
{ const size_t field_width {max_word_length(words) + 1};
 const size_t words_per_line {8};
 std::cout << std::left << std::setw(field_width) << *words[0]; // Output the first word
 size_t words_in_line {}; // Words in current line
 for (size_t i {1}; i < words.size(); ++i)
 { // Output newline when initial letter changes or after 8 per line
 if ((*words[i])[0] != (*words[i - 1])[0] || ++words_in_line == words_per_line)
 {       words_in_line = 0;
 std::cout << std::endl;  }
 std::cout << std::setw(field_width) << *words[i]; // Output a word
 }
 std::cout << std::endl;
}
```

This last line finishes the dictionary sorting program (lexigraphic order).

Summary

314 This isn't everything relating to functions, though. The next chapter covers function templates, and you'll see even more about functions in the context of user-defined types starting in Chapter 11.

- **Functions**- self-contained compact units of code with a well-defined purpose.

A well-written program consists of a large number of small functions, not a small number of large functions.

- A function definition consists of the function **header** that specifies the function name, the parameters, and the return type, followed by the function **body** containing the executable code for the function.
- **Function prototype** lets the compiler to process calls to an undefined function.
- **Pass-by-value-**for arguments to a function-passes copies of the original argument values, so the original argument values are not accessible from within the function.
- **Passing a pointer** to a function allows the function to change the pointed value that is pointed to.
- Declaring a pointer parameter as **const** prevents modification of the original value.
- You can pass the address of an array to a function as a pointer. If you do, you should generally pass the array's length along as well.
- Specifying a function parameter as a **reference** avoids the copying that is implicit in the pass-by-value mechanism.

A reference parameter that is not modified within a function should be specified as const.

- Input parameters should be **reference-to-const**, except for smaller values such as those of fundamental types.

Returning values is preferred over output parameters, except for very large values such as std::array<>.

- Specifying default values for function parameters allows arguments to be optionally omitted.
- Default values can be combined with **std::optional<>** makes signatures more readable. Can be used for optional return values.
- Returning a reference from a function allows the function to be used on the left of an assignment operator.

Specifying the return type as a reference-to-const prevents this.

- **Signature** of a function is- the function name together with the number and types of its parameters.
- **Overloaded functions**- functions with same name but with different signatures (different parameter lists).

Overloaded functions cannot be differentiated by the return type.

•**Recursive function**- function that calls itself. Implementing an algorithm recursively can result in elegant and concise code. Not usually, this is at the expense of execution time when compared to other methods of implementing the same algorithm.

■Chapter 9: Function Templates319

you can write the code just once, as a **function template**.

Function Templates 319

function template is a blueprint or a recipe for defining an entire family of functions, not a definition of a function; A function template is a parametric function definition, where a particular function instance is created by one or more parameter values

A function definition that is generated from a template is an instance or an instantiation of the template.

Parameters can be any data type.

The **typename** keyword identifies T as a type. You put the template parameters between angled brackets after the template keyword. They are separated by commas if there is more than one.

template <typename T> T larger(T a, T b) { return a>b ? a : b; } //Function Template Definition

template T larger(T a, T b); // Prototype for function template

template parameters. The parameter T is commonly used as a name for a parameter because most parameters are types, if there are multiple parameters, using more descriptive names may be recommended.

You can position the template in a source file in the same way as a normal function definition; you can also specify a prototype for a function template. In this case, it would be as follows: template T larger(T a, T b); // Prototype for function template

Creating Instances of a Function Template 320

This mechanism is referred to as **template argument deduction**. The arguments to larger() are literals of type double, so this call causes the compiler to search for an existing definition of larger() with double parameters.

A program only ever includes 1 copy of the definition of each instance, even if the same instance is generated in different source files.

double larger(double a, double b) { return a > b ? a : b; } // If function == larger(1.5 , 2.5) because of deduction

Template Type Parameters 322

That is, suppose T is a template parameter name; then you can use T to construct derived types such as T&, const T&, T*, and T[][3]. Or you can use T as an argument to a class template, as for instance in std::vector.

■ **Note #include<algorithm>** defines a **std::max()** & **std::min()** function template that is completely analogous.

Explicit Template Arguments 323

em of using different arguments types with larger() with an explicit instantiation of the template:

```
std::cout << "Larger of " << small_int << " and 19.6 is "
<< larger<double>(small_int, 19.6) << std::endl; // Outputs 19.6
```

It will throw a Narrowing Conversion Error if there is any implicit conversion. may not be what you want. Here's an example:

```
std::cout << "Larger of " << small_int << " and 19.6 is "
<< larger<int>(small_int, 19.6) << std::endl; // Outputs 19
```

Function Template Specialization 323

Call larger() with arguments that are pointers:

```
std::cout << "Larger of " << big_int << " and " << small_int << " is "
 << *larger(&big_int, &small_int) << std::endl; // Output may be 10!
```

The compiler instantiates the template with the parameter as type int*. The prototype of this instance is as follows:

```
int* larger(int*, int*);
```

need to be particularly careful when using pointer types as template arguments.

■ Note If for your compiler the output isn't 10, you can try to rearrange the order in which big_int and small_int are declared.

Specialization of the template is possible to accommodate a pointer type template argument. A specific parameter value or set of values when a template has multiple parameters, a template specialization defines behavior different to the standard template.

The specialization must be prototyped before its first use.

a specialization of larger() for type int* is as follows:

```
template <>
int* larger<int*>(int* a, int* b)
{  return *a > *b ? a : b;      }
```

Function Templates and Overloading 324

Function Templates with Multiple Parameters 326

Allowing different types for each function argument is easy enough and can often be a good idea to keep your templates as generic as possible. However, in cases such as this, you run into trouble when specifying the return type.

A first possible solution is to add an extra template type argument to provide a way of controlling the return type. Here's an example:

```
template <typename TReturn, typename TArg1, typename TArg2>
TReturn larger(TArg1 a, TArg2 b)
{
return a > b ? a : b;
}
```

Return Type Deduction for Templates 328

An easy way to have the compiler deduce it for you after instantiating the template:

```
template <typename T1, typename T2>
auto larger(T1 a, T2 b)
{  return a > b ? a : b;        }
```

---

decltype() and Trailing Return Types 328 / 351

Expression: a > b ? a : b without return type deduction, can't derive a type from an expression, and how to use this in a function template specification? The compiler processes the template from left to right. So, when the return type specification is processed, the compiler does not know the types of a and b. To overcome this, the **trailing return type** syntax was introduced that permits the return type specification to appear after the parameter list, like this:

```
template auto larger(T1 a, T2 b) -> decltype(a > b ? a : b)  { return a > b ? a : b; }
```

`decltype(e)` is the following type: If `e` is the name, "id-expression", of a variable, the resulting type is the type of the variable. BUT, if `e` evaluates to an lvalue of type `T`, then the resulting type is `T &`, and if `e` evaluates to an rvalue of type `T`, then the resulting type is `T`.

decltype(auto) and decltype() vs. auto 330

The difference is that unlike auto, decltype() and decltype(auto) will deduce to reference types and preserve const specifiers.

auto always deduces to a value type the return type deduction may introduce unwanted copies values returning a template function.

Default Values for Template Parameters 330

you can specify default values for template arguments at the beginning of the template argument list. Like so:

template TReturn larger(const TArg&, const TArg&);

It's common practice to specify default values for template parameters at the end of the list. The Standard Library uses this extensively, often also for nontype template parameters.

Nontype Template Parameters 331

The type of a nontype template parameter can be one of the following: • An integral type, such as int or long • An enumeration type • A pointer or reference to an object type • A pointer or a reference to a function • A pointer to a class member

```
template <int lower, int upper, typename T>
bool is_in_range(const T& value)
{  return (value <= upper) && (value >= lower); }
```

If you define your template like that, the compiler is capable of deducing the type argument:

std::cout << is_in_range<0, 500>(value); // OK – checks 0 to 500

Templates for Functions with Fixed-Size Array Arguments 332

```
double average10(const double (&array)[10]) // Only arrays of length 10 can be passed!
{  double sum {}; // Accumulate total in here
 for (size_t i {}; i < 10; ++i)
 sum += array[i];       // Sum array elements
 return sum / 10;       // Return average       }
```

A Way to Turn the Array size into a variable:

```
template <typename T, size_t N>
T average(const T (&array)[N])
{  T sum {}; // Accumulate total in here
   for (size_t i {}; i < N; ++i)
        sum += array[i]; // Sum array elements
   return sum / N; // Return average          }
```

Template argument deduction deduces the nontype template argument N from the type of the arguments passed to such a template.

Common to define overloads of functions based on such templates. The Standard Library, for instance, regularly uses this technique.

Summary 334

- **Function template**- parameterized recipe, used by the compiler to generate **overloaded functions**.
- The parameters in a function template can be type parameters or nontype parameters. The compiler creates an instance of a function template for each function call that corresponds to a unique set of template parameter arguments.
- Function templates can be **overloaded** with other functions or function templates.
- **auto** & **decltype(auto)** -compiler uses to deduce the return type of a function. This is particularly powerful in the context of templates because their return type may depend on the value of one or more template type arguments.

■Chapter 10: Program Files and Preprocessing Directives 337

Understanding Translation Units 337

TRANSLATION UNIT: Each **source file**, along with the contents of all the **header files** that you include in it. EX: Classes

-Compiler processes each translation unit independently to generate an **object file**, which contains: **Machine code** & information on references to entities: functions that were not defined in the translation unit—external entities in other words.

-The Set of object files is processed by the **linker**, establishes all necessary connections of object files to produce the executable program module.

Translation: The combined process of compiling and linking translation units

The One Definition Rule 338 ODR

In a **translation unit**: no variable, function, class type, enumeration type, or template must ever be defined more than once. You can have more than one declaration for a variable or function, but there must never be more than one definition that determines what it is and causes it to be created. If there's more than one definition within the same translation unit, the code will not compile.

■ **Note** A declaration introduces a name into a scope. A definition introduces the name & defines it.

In other words, all definitions are declarations, but not all declarations are definitions

-All these definitions of a given **inline** function or variable have to be identical.

-For this reason, always define **inline functions** & **variables** in a header file that you include in a source file whenever one is required.

-Make sure there's no duplicate type definitions within a single translation unit.

-Unlike class type definitions, **class member definitions** will mostly appear in **source files** rather than in header files.

-One definition rule applies differently to function templates (or class member function templates, covered in Chapter 16)

Program Files and Linkage 339

Translation units can only handle one set of code, so the linker points to where the code should be read from.

A program either may have linkage or doesn't. A name has linkage when you can use it to access something in your program that is outside the scope in which the name is declared. If this isn't the case, it has no linkage. If a name has linkage, then it can have internal linkage or external linkage. Every name in a translation unit has **internal linkage, external linkage,** or **no linkage.**

Determining Linkage for a Name 339

Linkage for each name in a translation unit is *after* contents of header files are put in the .cpp file, the basis for the translation unit.

Internal linkage: The entity that the name represents can be accessed from anywhere within the same translation unit. For example, the names of non-inline variables defined at global scope that are specified as const have internal linkage by default.

External linkage: A name with external linkage can be accessed from another translation unit in addition to the one in which it is defined. In other words, the entity that the name represents can be shared and accessed throughout the entire program. All the functions that we have written so far have external linkage and so do both non-const and inline variables defined at global scope.

No linkage: When a name has no linkage, the entity it refers to can be accessed only from within the scope that applies to the name. All names that are defined within a block—local names, in other words—have no linkage. Normal Function Linkage.

External Functions 339

In a program made up of several files: linker establishes (or **resolves**) the connection between a function **call** in one source file and the function definition in another. The compiler only needs a function prototype to call its functions. The prototype is contained within each declaration of the function. The compiler doesn't mind whether the function's definition occurs in the same file or elsewhere. This is because function names have external linkage by default. If a function is not defined within the translation unit in which it is called, the compiler flags the call as external and leaves it for the linker to sort out.

Function prototypes typically in header files, which you can then **#include** into your translation units.

External Variables 341

To access a variable defined outside the current translation unit, then you must declare the variable name using the **extern** keyword:

extern int power_range**; //** Declaration of an externally defined variable

The type must correspond exactly to the type that appears in the definition. Can't specify initial values in extern declarations because it's a declaration of the name, not a definition of a variable.

const Variables with External Linkage 342

global constant, rather than a modifiable global variable:

```
// Range.cpp
const int power_range {3};
```

A const variable, however, has internal linkage by default, which makes it unavailable in other translation units. You can override this by using the extern keyword in the definition:

```
extern const int power_range {3}; // Definition of a global constant with external linkage
```

To access power range in *another source file*, you must declare it as const and external:

```
extern const int power_range; // Declaration of an external global constant
```

The declaration is global scope in a translation unit OR within a block in which case it is available only within that local scope.

The best place for constants is in a (custom) header file.

Internal Names 343

Recommended way to define names with internal linkage today is through **unnamed namespaces**

■ Caution **Never use static anymore** to mark names that should have internal linkage; always use unnamed namespaces instead.

```
static double compute(double x, unsigned n) // compute() now has internal linkage
{ return n == 0 ? 1.0 : x * compute(x, n - 1);          }
```

Preprocessing Your Source Code 344

Preprocessing is a process executed by the compiler before a source file is compiled into machine instructions. Preprocessing prepares and modifies the source code for the compile phase according to instructions that you specify by preprocessing directives.

DIRECTIVE	DESCRIPTION
#include	Supports header file inclusion.

#if	Enables conditional compilation.
#else	else for #if.
#elif	Equivalent to #else #if.
#endif	Marks the end of an #if directive.
#define	Defines an identifier.
#undef	Deletes an identifier previously defined using #define.
#ifdef (or #if defined)	Does something if an identifier is defined.
#ifndef (or #if !defined)	Does something if an identifier is not defined.
#line	Redefines the current line number. Optionally changes the filename as well.
#error	Outputs a compile-time error message & stops the compilation. Typically part of a conditional preprocessing directive sequence.
#pragma	Offers vendor-specific features while retaining overall C++ compatibility.

The **preprocessing phase** analyzes, executes, and then removes all preprocessing directives from a source file. This generates the translation unit that consists purely of C++ statements that is then compiled. The **linker** must then process the object file that results along with any other object files that are part of the program to produce the executable module.

Directives provide flexibility in the way you specify your programs. Preprocessing operations occur before your program is compiled.

Defining Preprocessor Macros 345

#define directive specifies a so-called macro. A **macro**, or **token**, is an symbol for what it defines. **#define PI 3.14159265**

Much better to define a constant variable, like this: **inline const double pi {3.14159265358979323846};**

Places the definition in a header file for inclusion in any source file where the value is required or define it with external linkage:

extern const double pi {3.14159265358979323846};

Now you may access pi from any translation unit just by adding an extern declaration for it wherever it is required.

■ **Caution** Using a #define directive to define an identifier that you use to specify a value in C++ code has three major disadvantages: there's no type checking support, it doesn't respect scope, and the identifier name cannot be bound within a namespace. In C++, you should always use const variables instead.

#define directive is mainly used for management of header files.

Defining Function-Like Macros 346

You can define function-like text replacement macros as well. Here's an example:

#define MAX(A, B) A >= B ? A : B

Don't use macros, USE FUNCTION TEMPLATES, in C++14, the recommended way for defining variables such as π in standard C++ is using variable templates.

PROBLEMS: If you use the ternary operator together with the streaming operator <<, the operator precedence rules tell us that the expression with the ternary operator should be between parentheses.

PROBLEMS: If a macro parameter such as A appears more than once in the replacement, the preprocessor will blindly copy the macro arguments more than once. This undesired behavior with macros is harder to avoid.

■ **Caution Never use** preprocessor macros to define operations such as min(), max(), or abs(). Instead, always use regular C++ functions OR function templates. Function templates are far superior to preprocessor macros for defining blueprints of functions that work for any argument type. In fact, the cmath header of the Standard Library already offers precisely such function templates—including std::min(), std::max(), and std::abs()—so there's often no need for you to define them yourself.

Prepocessor Operators 348

The so-called stringification operator. Turns the argument in a string literal containing its value (by surrounding it with double quotes and adding the necessary character escape sequences).

The concatenation operator. Concatenates (pastes together, similar to what the + operator does for the values of two std::strings) the values of two identifiers.

```
// Ex10_04.cpp // Working with preprocessor operators
#include <iostream>
#define DEFINE_PRINT_FUNCTION(NAME, COUNT, VALUE)
 void NAME##COUNT() { std::cout << #VALUE << std::endl; }
DEFINE_PRINT_FUNCTION(fun, 123, Test 1 "2" 3)
int main()
{  fun123();   }
```

The preprocessor therefore allows you to add line breaks, as long as they are immediately preceded with a backslash character.

All such escaped line breaks are discarded from the substitution. The preprocessor first concatenates the entire macro definition back into one single line (in fact, it does so even outside of a macro definition)

Undefining Macros 349

```
#define PI 3.14159265358979323846264338327950288 4 // All occurrences of PI in code from this point will be replaced // by 3.14159265358979323846264338327950288 4 // ...
 #undef PI // PI is no longer defined from here on so no substitutions occur. // Any references to PI will be left in the code.
```

Including Header Files 350

You include your own header files into a source file with a slightly different syntax where you enclose the header file name between double quotes. Here's an example: **#include "myheader.h"**

■ Note Some libraries use the **.hpp extension** for C++ header files and reserve the use of the .h extension for header files that contain either pure C functions or functions that are compatible with both C and C++. Mixing C and C++ code is an advanced topic.

Compiler searches default directories that contain the Standard Library headers for the file when the name is between **angled brackets**. This implies that your header files will not be found if you put the name between angled brackets. If the header name is between double quotes, the compiler searches the current directory (typically the directory containing the source file that is being compiled) followed by the directories containing the standard headers. If the header file is in some other directory, you may need to put the complete path for the header file or the path relative to the directory containing the source file between the **double quotes**.

Preventing Duplication of Header File Contents 350

You have already seen that you don't have to specify a value when you define an identifier:

```
#define MY_IDENTIFIER
#if defined MY_IDENTIFIER  // OR #ifdef MY_IDENTIFIER
 // The code here will be placed in the source file if MY_IDENTIFIER has been defined.
 // Otherwise it will be omitted.
#endif
```

You can use the #if !defined or its equivalent, #ifndef, to test for an identifier not having been defined:

```
#if !defined MY_IDENTIFIER // OR #ifndef MY_IDENTIFIER
 // The code down to #endif will be placed in the source file
 // if MY_IDENTIFIER has NOT been defined. Otherwise, the code will be omitted.
#endif
```

This pattern ensures the contents of a header file are not duplicated in a translation unit:

```
// Header file myheader.h
#ifndef MYHEADER_H
```

```
#define MYHEADER_H
 // The entire code for myheader.h is placed here.
 // This code will be placed in the source file,
 // but only if MYHEADER_H has NOT been defined previously.
#endif
```

The previous #ifndef - #define - #endif pattern is common enough to have its own name; this particular combination of preprocessor directives is called an #include guard. *All header files* should be surrounded with an **#include guard** to eliminate the potential for violations of the one definition rule.

■ **Tip** Most compilers offer a **#pragma** directive to achieve the same effect as the pattern we have described.

With nearly all compilers, placing a line containing **#pragma once** at the beginning of a header file prevents duplication of the contents. Almost all compilers support this #pragma, it is not standard C++, so for this book we'll continue to use #include guards.

Your First Header File 352

Usually, you will not create a header for each individual function. The Power.h header could be the start of a larger Math.h header that groups any number of useful, reusable mathematical functions

```
// Power.h
#ifndef POWER_H
#define POWER_H
// Function to calculate x to the power n
double power(double x, int n);
#endif
```

Namespaces 353 ::global = Global accessor.

Namespace- block that attaches an extra name—the namespace name—to every entity name that is declared or defined within it.

The full name of each entity is the namespace name followed by the **scope resolution operator, ::,** followed by the basic entity name

Different namespaces can contain entities with the same name, entities are differentiated by different namespace names.

The Global Namespace 353

With small programs, you can define your names within the global namespace without running into any problems. With larger applications, the potential for name clashes increases, so you should *use namespaces to partition your code* into logical groupings. That way, each code segment is self-contained from a naming perspective, and name clashes are prevented.

int MyFunction::myFunction(); // EXAMPLE MyFunction Namespace

Defining a Namespace 354

```
namespace myRegion
{ // Code you want to have in the namespace, including function definitions and
  declarations, // global variables, enum types, templates, etc.  }
```

Note that no semicolon is required after the closing brace in a namespace definition.

The namespace name here is myRegion. The braces enclose the scope for the namespace myRegion, and every name declared

within the namespace scope has the name myRegion attached to it.

■ Caution don't include the main() function within a namespace. Runtime environment expects main(), defined in global namespace.

```
namespace calc
{  // This defines namespace calc        // The initial code in the namespace goes here
}
namespace sort
{  // Code in a new namespace, sort    }
   namespace calc
{  /* This extends the calc namespace Code in here can refer to names in the previous calc namespace block without qualification
   */}
```

'sort' is an extension namespace definition. It extends the original namespace definition.
You can have several extension namespace definitions in a translation unit.

References to names from inside the same namespace do not need to be qualified.
Example: names, defined in namespace calc can be referenced from within calc without qualifying them with the namespace name.

Programs will consist of two files: one header file and one source file.

The header file defines a few common mathematical constants in the namespace with name constants:

```
// Constants.h          // Using a namespace
#ifndef CONSTANTS_H
#define CONSTANTS_H
namespace constants
{  inline const double pi { 3.14159265358979323846 }; // the famous pi constant
   inline const double e { 2.71828182845904523536 }; // base of the natural logarithm
   inline const double sqrt_2 { 1.41421356237309504880 }; // square root of 2  }
#endif
```

LEGACY CODE USES: extern VS inline

Using Constants from header file with the main source file:

```
// Ex10_06.cpp          // Using a namespace
#include <iostream>
#include "Constants.h"
int main()
{  std::cout << "pi has the value " << constants::pi << std::endl;
   std::cout << "This should be 2: " << constants::sqrt_2 * constants::sqrt_2 << std::endl;
}
```

Applying using Declarations 356

using declaration for a single name from a namespace: **using namespace_name::identifier;**
For instance, a set of overloaded functions defined within a namespace can be introduced with a single using declaration.
■ **Note** When you use an unqualified name, the compiler first tries to find the definition in the current scope, prior to the point at which it is used. If not, the compiler looks in the immediately enclosing scope. This continues until the global scope is reached. If a definition is not found at global scope (which could be an extern declaration), the compiler concludes that the name is not defined.

Functions and Namespaces 357

For a function to exist within a namespace, it is sufficient for the function prototype to appear in the namespace.
The function definition doesn't have to be enclosed in a namespace block (it can, but it doesn't have to be).
In general, it can sometimes be a good idea to have #include directives for every header that a file uses, even when one header may include another header that you use. This makes the file independent of potential changes to the header files.

Unnamed Namespaces 359

You don't have to assign a name to a namespace. You can declare an unnamed namespace with the following code:

namespace
{ // Code in the namespace, functions, etc. }

This creates a namespace that has a unique internal name that is generated by the compiler.

Only one "unnamed" namespace exists within each translation unit, additional namespace declarations without a name will be extensions of the first.

If a function is not supposed to be accessible from outside a particular translation unit, you should always define it in an unnamed namespace. Using a static specifier for this purpose is no longer recommended.

Nested Namespaces 382

```
// outin.cpp
#include "outin.h"    // Uses the normalize min() function from this header file
void outer::inner::normalize(std::vector<double>& data)
{  // ...
 double minValue{ min(data) }; // Calls min() in outer       // ...moreCode;
}
```

Because defining inside nested namespaces that way can become cumbersome quite fast—especially as the number of levels grows to, say, three or more—the latest C++17 version of the language has introduced a new, more convenient syntax for this. In C++17, you can write our earlier example like this:

```
namespace outer::inner
{ double average(const std::vector<double>& data) { /* body code... */ }
}
```

Namespace Aliases 362

General form of the statement you'd use to *define an alias for a namespace* name is as follows:

namespace alias_name = original_namespace_name;

to define an alias for the namespace name in the previous paragraph, you could write this:

namespace G5P3S2 = Group5::Process3::Subsection2; //Best way to Shorten Namespaces

Now you can call a function within the original namespace with a statement such as this:

int maxValue {G5P3S2::max(data)};

Logical Preprocessing Directives 362

These allow conditional inclusion of code and/or further preprocessing directives in a file, depending on whether preprocessing identifiers have been defined or based on identifiers having specific values.

The Logical #if Directive 362

logical #if directive can test whether a symbol has been previously defined.

```
#ifndef SOME_FUNCTION // Defines if NOT defined. Then ->
#def SOME_FUNCTION
#ifdef SOME_FUNCTION // Uses the Preprocessing Directive if already defined
   cout << "YAY! This Function is defined and this string was displayed!" << endl;
#endif // Ends Directive's of #ifdef
```

The code between the #if and #endif directives is compiled as part of the program only if it's defined

Testing for Specific Identifier Values 363

The general form of the #if directive is as follows:

#if constant_expression

---arithmetic operations are executed are type long or unsigned long, & Boolean operators (||, &&, and !) are supported as well.

```
#if _WIN64 || __x86_64__ || __ppc64__
 // Code taking advantage of 64-bit addressing...
#endif
```

Consult your compiler documentation for these and other predefined macro identifiers.

Multiple-Choice Code Selection 363

#else directive the same as the C++ else statement, it identifies a sequence of lines to includ in the file if the #if condition fails.

```
#if LANGUAGE == ENGLISH
 #define Greeting "Good Morning."
#elif LANGUAGE == GERMAN
 #define Greeting "Guten Tag."
#elif LANGUAGE == FRENCH
 #define Greeting "Bonjour."
#else
 #define Greeting "Ola."
#endif
 std::cout << Greeting << std::endl;
```

Standard Preprocessing Macros 365

There are several standard predefined preprocessing macros, and the most useful are listed. They all have **2 Underscores before & also usually after the name**

MACRO	DESCRIPTION
__LINE__	The line number of the current source line as a decimal integer literal.
__FILE__	The name of the source file as a character string literal.
__DATE__	The date when the source file was preprocessed as a character string literal in the form Mmm dd yyyy. Here, Mmm is the month in characters, (Jan, Feb, etc.); dd is the day in the form of a pair of characters 1 to 31, where single-digit days are preceded by a blank; and yyyy is the year as four digits (such as 2014).
__TIME__	The time at which the source file was compiled, as a character string literal in the form hh:mm:ss, which is a string containing the pairs of digits for hours, minutes, and seconds separated by colons.
__cplusplus	A number of type long that corresponds to the highest version of the C++ standard that your compiler supports. This number is of the form yyyymm, where yyyy and mm represent the year and month in which that version of the standard was approved. At the time of writing, possible values are 199711 for nonmodern C++, 201103 for C++11, 201402 for C++14, and 201703 for C++17. Compilers may use intermediate numbers to signal support for earlier drafts of the standard as well.

You can use the date and time macros to record when your program was last compiled with a statement such as this:

```
std::cout << "Program last compiled at " << __TIME__ << " on "<< __DATE__ << std::endl;
```

Testing for Available Headers 366

```
#if __has_include(<SomeStandardLibaryHeader>)
 #include <SomeStandardLibaryHeader>
 // ... Definitions that use functionality of some Standard Library header ...
```

```
#elif __has_include("SomeHeader.h")
 #include "SomeHeader.h"
 // ... Alternative definitions that use functionality of SomeHeader.h ...
#else
 #error("Need at least SomeStandardLibaryHeader or SomeHeader.h")
#endif
```

■ TIP **__has_include()** macro is new. You can check whether it is supported using the #ifdef directive, as in #ifdef __has_include.

Debugging Methods 367

Ways in which **bugs** (**errors**) can arise: Most simple typos // Logical errors // Failing to consider all variations of input data.

-You have experience with this through the regular patches and updates to the operating system and some of the applications on your computer. Most bugs in this context are relatively minor and don't limit the usability of the product greatly.

The most serious bugs in commercial products tend to be **security issues**.

PREVENTION OF BUGS-- Well-structured program consists of compact functions & well-defined purpose // Well-chosen variable & function names W/comments documenting the operations & purpose of functions // Good use of indentation and statement layout

Integrated Debuggers 367

Many C++ compilers come with a program development environment that has extensive debugging tools built in.

Main debugging tools- **Tracing Program Flow / Setting Breakpoints / Setting Watches / Inspecting Program Elements**

- **Tracing program flow**: Execute a program 1 statement at a time. May have to be compiled in Debug Mode to make this possible. The debug environment usually allows you to display information about all relevant variables at each pause. Can be tedious.
- **Setting breakpoints**: Stepping through a loop that executes 10,000 times is an unrealistic proposition. Breakpoints identify specific statements in a program at which execution pauses to allow you to check the program state and then continues to the next breakpoint when you press the specified key.
- **Setting (variable)watches**: A watch identifies a specific variable whose value you want to track as execution progresses. The values of variables identified by watches you have set are displayed at each pause point. If you step through your program statement by statement, you can see the exact point at which values are changed and sometimes when they unexpectedly don't change.
- **Inspecting program elements**: You can usually *examine a variety of program components when execution is paused*. At breakpoints Examine details of a function, such as its return type and its arguments // Information relating to a pointer, such as its location, the address it contains, and data at that address. It is sometimes possible to access the values of expressions and to modify variables. Modifying variables can often allow problem areas to be bypassed, allowing subsequent code to be executed with correct data.

Preprocessing Directives in Debugging 368

Adding your own tracing code can still be useful. You can use conditional preprocessing directives to include blocks of code to assist during testing and omit the code when testing is complete. Such as if **int Debug_Var == 1** and change it to **Debug_Var == 0** to change the custom debugging conditions.

Use this to prevent duplicate definitions: **#ifndef FUNCTIONS_H #define FUNCTIONS_H**

File Management: Put all your directives that control trace and other debug output into a separate header file. You can then include this into all your .cpp files. Then, you can alter the kind of debug output you get by making adjustments to this one header file

Random Number Generator 369

This Seeds a Random Number (Allows it to be generated)

std::srand(static_cast<unsigned>(std::time(nullptr))); // Seed random generator

This Creates a Random Number from Range: 0 to HighestNumber

```
size_t random(size_t count) {
    return static_cast(std::rand() / (RAND_MAX / HighestNumber));
}
```

■ Caution The rand() function in the stdlib header does not generate random numbers well enough. Ok for Simplest Applications.

-- more serious use of random numbers, investigate the functionality provided by the **random** header of the Standard Library.

time() function that is declared in the **ctime** header returns the number of seconds since January 1, 1970, as an integer, so using this as the argument to srand() ensures that you get a different random sequence each time the program executes

Using the assert() Macro 371

assert() preprocessor macro: To Test Logical Expressions, defined in header **cassert**

assert(expression) terminates the program (by calling std::abort()) with a diagnostic message, if expression evaluates to false.

standard error stream, **cerr**, which is always the command line.

■ Tip- Some debuggers, in particular those integrated into graphical IDEs, allow you to pause each time an assertion is triggered, right before the application terminates. This greatly increases the value of assertions during debugging sessions.

#define NDEBUG // Causes all assertions in the translation unit to be ignored.

■ Caution assert() is for detecting programming errors, not for handling errors at runtime.

Static Assertions 373

The assert() macro is for checking conditions dynamically, at runtime, whereas static assertions are for checking conditions statically, at compile time. A static assertion is a statement of either of the following forms:

static_assert(constant_expression);

static_assert(constant_expression, error_message);

■ Note The type_traits header contains a large number of type testing templates including is_integral_ v, is_signed_v, is_unsigned_v, is_floating_point_v, and is_enum_v

Summary 375

- Each **Entity** in a **Translation Unit** has only **one definition**. Multiple definitions are allowed throughout a program, if they're identical.
- A name can have **internal linkage**, meaning that the name is accessible throughout a translation unit; **external linkage**, meaning that the name is accessible from any translation unit; or **No linkage**, the name is accessible in the block where it's defined.

- You use **Header Files** to contain **Definitions** & **Declarations** required by your **Source Files**. A header file can contain template and type definitions, enumerations, constants, function declarations, inline function and variable definitions, and named namespaces. By convention, header file names use the **extension: .h**
- Your **source files** will typically contain the **definitions for all non-inline functions and variables** declared in the corresponding header. A C++ source file usually has the file name **extension: .cpp**
- You **insert** the contents of a **header file** into a .cpp file by using an **#include directive**.
- A **.cpp file** is the **basis** for a **translation unit** that is processed by the **compiler to generate** an **object file**.
- A **namespace** defines a **scope**; all names declared within this scope have the namespace name attached to them. All declarations of names that are not in an explicit namespace scope are in the global namespace.
- A single namespace can be made up of several separate namespace declarations with the same name.
- Identical names that are declared within different namespaces are distinct.
- To refer to an **identifier** that is **declared** within a **namespace** from outside the namespace, you need to specify the namespace name and the identifier, separated by the **scope resolution operator**, **::** .
- Names declared within a namespace can be used without qualification from inside the namespace.
- The **preprocessing phase** executes **directives** to transform the **source code** in a **translation unit** prior to compilation. When all directives have been processed, the translation unit will only contain C++ code, with no preprocessing directives remaining.
- Use **conditional preprocessing directives** to **ensure** that the **contents** of a **header file** are **never duplicated** within a translation unit.
- Use **conditional preprocessing directives** to **control** whether **trace** or other **diagnostic debug code** is included in your program.
- **assert() macro** enables you to **test logical conditions** during execution and **issue a message** and **abort** the program if the logical condition is false.
- Use **static_assert** to **check type arguments** for template parameters in a template instance to ensure that a type argument is consistent with the template definition.

Chapter 11: Defining Your Own Data Types 379

Classes and Object-Oriented Programming 379

Class: Programmer-defined Data Type, Variables & Functions are called **Members**. can be a composite of **Member Variables**(**Fields**)(**Data Members**) & **Member Functions**(**Methods**)(Behavior) of other types.
Public Functions Get or Set the Private Variables.

Object or **Instance** - Each of these variables or array elements. **Instantiation** – Defining an Instance.

object-oriented programming (OOP) using objects for the problem. Involves designing a set of Variables for the problem. Consists of: encapsulation and data hiding, inheritance, and polymorphism. Data values that define an object are needed "sufficient for your needs," not "sufficient to define the object in general." OOP is comprehensive and flexible.

Encapsulation 380

Encapsulation - Packaging Values and functions in an object, allows for extra, directed, code for functions.
2 Encapsulation Methodologies: Bundling data & Hiding Data.

Data Hiding – Protecting Integrity of & Restricting Access to object members. Often the Synonym for Encapsulation. Also Called **Information Hiding**. Only provide Specific access through Member Functions to alter a Member Variable to alter or obtain a value. Not mandatory, but it's generally a good idea.

Data Hiding: Protects Object Integrity & a Good Interface reduces **Coupling** – How Independent data is. Allows you to Change the State or Interface without rewriting the whole Program. Loose Coupling is the goal. Interface stability is the Most Important.

Member Variables represent the **state** of the object. **Member Functions** that manipulate them is the object's **interface**. The Internal States can change without affecting the Interface.

Class interface is dependent on function names, parameter types, and return types specified for the interface.
Direct access to the values that define an object undermines the whole idea of object-oriented programming.
Some environments have advanced functionality to put breakpoints if a particular member variable changes, putting breakpoints on function calls or specific lines of code inside functions is much easier.

Inheritance 383

Inheritance - to define an Class in terms of another Class, can also have new Characteristics or Change Characteristics (**Redefinition**). Basis of Polymorphism.

Derived Class inherits from the **Base Class**.

overriding - Redefining a base class's function in a derived class; the latter function is to override the former. This is very powerful.

Polymorphism 384

Polymorphism: Ability to Change Objects of Inheritance-related classes & be passed around & operated on using base class pointers and references.
In C++, this always involves calling a member function of an object using a pointer or a reference.

Defining a Class 386

basic organization of a class definition looks like this:

```
class ClassName
{
 // Code that defines the members of the class...
};
```

CONVENTION - Use the Uppercase Name for userdefined classes to distinguish class types from variable names.

All the members of a class are **private :** by default - they cannot be accessed from outside the class.

public : -followed by a colon to make all subsequent members accessible from outside the class.

Access Specifiers – public, protected, private

Private Members- accessed only from functions of the same class.

Public Members- are to be accessed by a function that is not a member of the class must be specified as public.
A member function can reference any other member of the same class, with using "::", Scope Operator.

Public Constructor – Used to set the values of private member variables when an object is created.
Class Objects are only created using a constructor.

Structures (structs) - nearly completely equivalent to classes, except All Members are Public. Used to represent simple aggregates of a few variables of different types—the margin sizes and dimensions of a printed page. Created by using Keyword **struct** rather than class.

Constructors 388

Constructor – Function that creates Instances. Creates / Ensures Members are Valid. Always Same name as Class.
Box::Box(), for example, is a constructor for the Box class. Does not return a value and therefore has no return type.

Default Constructors 388

Default Constructor- Has No Parameters or All Parameters have Default Values. Sometimes either are needed.

If you don't define a constructor for a class, the compiler will supply a **default default constructor**(No Parameter constructor).

Its sole purpose is to allow an object to be created. It does nothing else. Only created when there is no other constructor.

```
class Box {
private:
 double length {1};
 double width {1};
 double height {1};
public:          // The default constructor that was supplied by the compiler...
 Box()
 {               // Empty body so it does nothing...
 }
 // Function to calculate the volume of a box
 double volume()
   { return length * width * height;
   }
};
```

Defining a Class Constructor 389

```
class Box
{
private:
 double length {1.0};
 double width {1.0};
 double height {1.0};
public:
 // Constructor
 Box(double lengthValue, double widthValue, double heightValue)
 {
std::cout << "Box constructor called." << std::endl;
   length = lengthValue;
   width = widthValue;
   height = heightValue;
 }
```

No return type is allowed, and the name of the constructor must be the same as the class name
Useful for defining a box with the default values that are already specified

Box secondBox; // Causes a compiler error message ////UNLESS a Default Constructor is made

Using the default Keyword 391

-The Compiler doesn't generate a Constructor, if 1 if defined.

Box() {} // Default constructor // Good if some default values are already defined.

Box() = default; // Default constructor // Makes Compiler generated a Default Default Constructor

the use of the default keyword is preferred in modern C++ code & has a few subtle technical reasons why automated one is better.

Defining Functions and Constructors Outside the Class 391

CONVENTION for Classes: Variable/Functions/Constructor Parameters in a Header file & Constructor Defintions in a Source File.

The main() Source File Only needs to include the Header file, since the header file includes the .ccp file for the header.

Function definitions within a class definition are implicitly inline.

Box Class Header File SEPARATE FROM MAIN() --- Box class Header file. Used to instantiate member functions and variables.

```
// Box.h
#ifndef BOX_H          //Safety Convention
#define BOX_H //…….
class Box
{
private:
    double length {1.0};
    double width {1.0};
    double height {1.0};
public:
 // Constructors / Function Prototypes
 Box(double lengthValue, double widthValue, double heightValue);
 Box() = default;
 double volume(); // Function to calculate the volume of a box
};
#endif
```

Box Class Source File = Stores Constructor / Function Defintions

```
// Box.cpp
#include "Box.h"       // Connects Source file to the Header File so it can define the constructor prototypes
#include <iostream>
// Constructor definition
Box::Box(double lengthValue, double widthValue, double heightValue)
{   std::cout << "Box constructor called." << std::endl;
    length = lengthValue;
    width = widthValue;
    height = heightValue;
}
// Function to calculate the volume of a box
double Box::volume()
{ return length * width * height;
}
```

Default Constructor Parameter Values 393

default values for the parameters in the function prototype. You can do this for class member functions, including constructors.

```
public:          // Constructors
   Box(double lv = 1.0, double wv = 1.0, double hv = 1.0);          //Constructor with a default Value
   Box() = default;                                                 //Constructor with a default default Value
```

Using a Member Initializer List 394

```
// Constructor definition using a member initializer list
Box::Box(double lv, double wv, double hv)
: length {lv}, width {wv}, height {hv}          // Member Initializer List
{
   std::cout << "Box constructor called." << std::endl;
}
```

---

the parameter list by a **colon (:)**, and each initializer is separated from the next by a **comma (,)**.

When using initialization lists, List value initializes the member variable, as it is created. Especially handy when creating Class Objects.

The order that member variables are initialized is determined by the order they're declared in the class definition—NOT by the order that they appear in the member initializer list.

■ As a rule, **Initialize all member variables in the constructor's member initializer list**. More efficient. To avoid any confusion, put member variables in the initializer list in the same order as declared in the class definition. Initialize variables in the constructor body only if either more complex logic is required or the order in which they are initialized is important.

Using the explicit Keyword 394

explicit – Makes sure the value being passed to a parameter is the right data type.

By default, you should therefore declare all single-argument constructors as explicit (note that this includes constructors with multiple parameters where at least all but the first have default values); omit explicit only if implicit type conversions are truly desirable.

Delegating Constructors 397

```
Box::Box(double lv, double wv, double hv)          // Constructor that takes 3 arguments and sets the member values
: length {lv}, width {wv}, height {hv}
{ std::cout << "Box constructor 1 called." << std::endl; }
Box::Box(double side) : Box{side, side, side}      // Delegating Constructor 2nd
{                                                  // ..Delegates the construction work to the other constructor
 std::cout << "Box constructor 2 called." << std::endl;
}
```

Handy in the situation of these calls:

```
Box box1 {2.0, 3.0, 4.0}; // An arbitrary box     // Uses the 1st Constructor
Box box2 {5.0}; // A box that is a cube           // Uses the 2nd Constructor
```

The Copy Constructor 398

compiler supplied a default **copy constructor**, a constructor that creates an object by copying an existing object:

```
Box box3 {box2};
```

Can cause problems when a variable is a pointer. The copied variable is a pointer, not a value. Meaning when an object is created by the copy constructor, it is interlinked with the original object. Changing one value changes both values. Handy in some situations.

Implementing Copy Constructor 398

Copy constructor must accept a reference-to-const (Not Value) argument of the same class type & create a duplicate in an appropriate manner. Use reference-to-const because a copy constructor only creates duplicates, not modify the original.

Box::Box(**Box box**) : length {box.length}, width {box.width}, height {box.height} **// Wrong!!** (Creates a Recursive Loop)
{ //Code Body }

Box::Box(**const Box& box**) : length {box.length}, width {box.width}, height {box.height} **// Right!!**
{ //Code Body }

---

the form of the copy constructor is the same for any class:

Type::Type(const Type& object)
{
// Code to duplicate the object...
}

Delegate Copy Constructor:

Box::Box(const Box& box) : Box{box.length, box.width, box.height} {}

Accessing Private Class Members 400

ENCAPSULATION: Provide access to the values of private member variables by adding public member functions to return their values.

Accessor: Getter & Setter Functions (Retrieve/Define the values of member variables)

Accessor CONVENTION: For Variables use: getVariable() & setVariable(). For type bool: isTrue() or isValid() .

void setLength(double lv) { if (lv > 0) length = lv;} // Mutator Function

Mutator Function: allows variables to be modified.

The this Pointer 402

this-> *variable*-refers to members of the object in use. EX: getName() {return this->name;} // Each Person has their own name

When class member functions executes, it has a hidden this-> pointer. Contains the object address currently being used.

Returning this from a Function 402

Pointer Return Type for a member function, you can return this & use the pointer returned by one member function to call another.

myBox.setLength(-20.0)->setWidth(40.0)->setHeight(10.0); // Set all dimensions of myBox

---

Instead of a pointer, you can of course return a reference as well:

```
Box& Box::setLength(double lv)          // Box&   instead of Box*
{  if (lv > 0) length = lv;
   return *this;                        // Sets an Objects Value
}
```

This pattern is called **method chaining**. **return *this** allows chain calls of member function (**Uses References though**)

myBox.setLength(-20.0).setWidth(40.0).setHeight(10.0); // Set all dimensions of myBox

const Objects and const Member Functions 404

const Box myBox {3.0, 4.0, 5.0}; // Useless W/O Member Functions. Can't use Getters or Setters w/o const Member Functions

std::cout << "The length of myBox is " << myBox.length << std::endl; // ok
myBox.length = 2.0; // Error! Assignment to a member variable of a const object...
myBox.width *= 3.0; // Error! Assignment to a member variable of a const object...

const Box* boxPointer = &myBox; // A pointer-to-const-Box variable
boxPointer->length = 2; // Error!
boxPointer->width *= 3; // Error!

---

pointer-to-const or reference-to-const cannot modify the state of an object passed as an argument, even if the object was non-const.

const Member Functions 404

const Member Function- specifies which functions are allowed to called on const Box objects. Looks Like:

double volume() const; // Function to calculate the volume of a const Box. ADD const AFTER FUNCTION essetially

const Correctness 406

Use const Objects when you don't want the value of an object modified. Use const Member Functions when you want to use the values of the const Object, without modifying it. No Setters Allowed, Only Getters.

const Correctness- Compiler-Enforced restrictions that prevent const Objects from being modified (mutated).

Overloading on const 407

Add extra functions that access const variables. CONVENTION: Don't do this. Just as bad a making variables public.

Casting Away const 409 Don't do this

const_cast<>() operator- makes a const a non-const. Used in the following two forms:
const_cast<Type*>(expression)
const_cast<Type&>(expression)
CONVENTION: Don't do this to undermine const-Ness, which could cause bugs.

Using the mutable Keyword 409 Probably don't do this

only need mutable members in rare cases. Usually to modify an object from within a const function, it probably shouldn't have been const. Typical uses of mutable member variables include debugging or logging, caching, and thread synchronization members.

Friends 410 Rare to do this

friends of a class have access to select private variables of a class. Friend-ship goes one-way & doesn't extend to other classes.
They are Friend Functions of a different Class OR Friend Classes (As a whole)
CONVENTION: Use Sparingly and wisely!! Very Rare. useful in situations where a function needs access to the internals

The Friend Functions of a Class 411

public: // Constructor
Box(double lv = 1.0, double wv = 1.0, double hv = 1.0);
double volume() const; // Function to calculate the volume of a box
friend double surfaceArea(const Box& aBox); // Friend function for the surface area

Friend Classes 413

All member functions of a friend class have unrestricted access to all members of the friended class.
class Box
{ friend class Carton; //Box has no access to Carton, but Carton can access Box

};

Arrays of Class Objects 413

Box boxes[6] {box1, box2, box3, Box {2.0}}; // Copies 3 Boxes, Creates 1 with a Constructor, Creates 2 with Default Constructor

Same as making an Array of anything Else. Just Make sure it's the Right Class Type.

The Size of a Class Object 415

You obtain the size of a class object by using the **sizeof** operator.

The Size is the sum of the sizes of the member variables of the class.

boundary alignment- On most computers, for performance reasons, two-byte variables must be placed at an address that is a multiple of two, four-byte variables must be placed at an address that is a multiple of four, and so on.

Static Members of a Class 416

Static Member Variables- Member Variables declared **static**, Used in a Class so it can only accessed by that Class.

CONVENTION: Store constants that are specific to a class, or you could store Object info, like: static int howManyObjectsAreThere = 0;

static member function- Used to Access Static Member Variables.

■ Tip If member function doesn't access any nonstatic member variables, it may be a good to be declared as a static member function.

Static Member Variables 416 Using inline

Each static member variable is accessible and shared among all objects of a class.

The Instance of the Static Variable doesn't change and the value last until the end of program execution.

A use for static member variable: count how many objects of a class exist. The following statement does this:

```
class Box
{
private:
 static inline size_t objectCount {}; // Count of objects in existence. private so Can't access objectCount outside of Box class.
 double length;   // inline allows variable definition to be #included in multiple translation units w/o violating one definition rule (ODR).
 double width;    // inline CONT'D: Keeps the Value the same, eventhough the Box class will be Instantiated multiple times.
 double height;
public:       //When objectCount gets Incremented, it can have a function to check objectCount.
 size_t getObjectCount() const { return objectCount; } // Accesses Static Variable, const doesn't let value change
};
```

You do have to qualify the member name with the class name, Box, though, so that the compiler understands that you are referring to a static member of the class, unless it's a Global Variable.

Box::objectCount is CONVENTION, when accessing public static variables for readability.

Static Constants 421

Often used to define constants, global-like, the value is accessed instead of copied for every Instantiatiation.

static inline const float PI { 3.141592f };__// Creates 1 Instance of PI accessible to the Class it's inside.

■ **Tip** You typically define all member variables that are both **static** and **const** as **inline** as well. This allows you to initialize them directly inside the class definition, irrespective of their type.

Very common to define public constants containing, EX: boundary values of function parameters (like maxRadius and maxHeight) or suggested default values (defaultMaterial).

■ Note The three keywords **static**, **inline**, and **const** may appear in any consistent order.

Static Member Variables of the Class Type 422

Creating an **const Object** that has **static values**, which can be useful for creating a Object to reference, as a standard.

const Box Box::refBox {10.0, 10.0, 10.0}; // Creates the refBox Object 10x10x10 to use to reference other Box Objects

Static Member Functions 423

Static Member Function- Created at any time, has all privilages of the class.

Destructors 424

Destructor- Unique member of the class, Executed on object destruction for any cleanup that may be necessary.

If not Specified, a Default one is provided by the Compiler. Like This: **~ClassName() {}**

Always the class name prefixed with a tilde, ~ . Has no parameters or return types.

Box::~Box() = default; // Have the compiler generate a default destructor // OPTIMAL

Using Pointers as Class Members 427

Real-life programs generally consist of large collections of collaborating objects, linked together using pointers, smart pointers, references, and are deleted in a timely manner to manage memory and resources.

- **std::unique_ptr<>** - never forget to delete an object allocated from the free store.
- **std::shared_ptr<>** - invaluable if multiple objects point to & use the same object because the object will be deleted when it is NOT needed anymore.

A dynamically allocated object should always be managed by a smart pointer

The Truckload Example 427

Box->Package->Truckload Breaks Truckload down into a linked list of Boxes grouped in Packages.

```
// Package.h       // This is the Package Class
#ifndef PACKAGE_H
#define PACKAGE_H
#include <memory>
#include "Box.h"
using SharedBox = std::shared_ptr<Box>;
class Package
{
private:
 SharedBox pBox; // Pointer to the Box object contained in this Package
 Package* pNext; // Pointer to the next Package in the list
public:
 Package(SharedBox pb) : pBox{pb}, pNext{nullptr} {} // Constructor
 ~Package() { delete pNext; } // Destructor
 // Retrieve the Box pointer
 SharedBox getBox() const { return pBox; }
 //Retrieve or update the pointer to the next Package
 Package* getNext() { return pNext; }
```

```
 void setNext(Package* pPackage) { pNext = pPackage; }
};
#endif
```

This is the Truckload Class, which encapsulates Package, encapsulates Box

```
class Truckload
{
private:
 Package* pHead {}; // First in the list
 Package* pTail {}; // Last in the list
 Package* pCurrent {}; // Last retrieved from the list
public:
 Truckload() = default; // Default constructor - empty truckload
 Truckload(SharedBox pBox) // Constructor - one Box
 { pHead = pTail = new Package{pBox}; }
 Truckload(const std::vector<SharedBox>& boxes); // Constructor - vector of Boxes
 Truckload(const Truckload& src); // Copy constructor
 ~Truckload() { delete pHead; } // Destructor: clean up the list
 SharedBox getFirstBox(); // Get the first Box
 SharedBox getNextBox(); // Get the next Box
 void addBox(SharedBox pBox); // Add a new Box
 bool removeBox(SharedBox pBox); // Remove a Box from the Truckload
 void listBoxes() const; // Output the Boxes
};
```

Class invariant-an assertion that captures properties & relationships, remain stable through the life-time of instances of the class.

Components In a **Linked List**: Head, *Previous*, Current, *Next*, Last.

Singly Linked List use: *Head, *Current, *Next, *Last // **Doubly** Linked Lists use: *Previous (To Search in Reverse Order)

The previous pointer is sometimes called a **trailing pointer**

Nested Classes 440

nested class- class defined inside a Outer class & has full access to that class.

Nested Classes with Public Access 441

iterator pattern- Instead of storing pointer inside an object, use a member function, as a nested Iterator class, designed to aid with traversing an Object. Such an object is then called an **iterator**.

Iterators- allow external code to traverse the content of a container without having to know about the data structure's internals.

Summary

- A **Class** provides a way to define your own data types. Classes represent whatevertypes of objects your particular problem requires.
- Classes contain: **member variables** and **member functions**. The member functions of a class always have free access to the member variables of the same class, such as protected and private.
- **Objects**- (Instances)- created and initialized using member functions called **constructors**, called automatically when an object declaration is encountered. Can be overloaded to provide different ways of initializing an object.

- **Copy Constructor**- Constructor for an object, initialized with an existing object of the same class. The compiler generates a default copy constructor for a class if you don't define one.
- **Members** of a class can be specified as **1. public:** freely accessible in a program. **2. private:** accessed only by *member functions*, **friend functions**, or members of **nested classes**.
- Member variables of a class can be **static**. Only one instance of each static member variable of a class exists.
- **Static member variables** aren't part of the object and don't contribute to its size.
- All **non-static** member function contains the pointer **this->** ,or **(*this).** , points to the current object, which the function called.
- **Static member functions** can be called even if no objects of the class have been created. Doesn't contain the pointer *this*.
- **const member functions** can't modify, unless the member variables have been declared as **mutable**.
- Using references to class objects as arguments to function calls avoids substantial overheads in passing complex objects to a function.
- **Destructor-** member function, called when an object is destroyed. If not defined, the compiler supplies a default destructor.
- **Nested Class-** A class defined inside another class definition.

Chapter 12: Operator Overloading 449

Implementing Operators for a Class 449

Add Support for Operators, like + or -, for Classes to be able to manipulate Objects in a more natural way.

Operator Overloading 450

Operator overloading enables you to apply standard operators like: +, –, *, <, and etc, to objects of your own class types.

To define an operator for objects of your own type, all you need to do is write a function that implements the desired behavior.

Same as Function, except it's composed of the **operator** keyword followed by the operator that you are overloading.

Implementing an Overloaded Operator 450

bool operator<(const Box& aBox**)** const**;** // Overloaded 'less-than' or '<' operator. **PROTOTYPE Form=**
operator<()

bool operator<(const Box& aBox) const // OVERLOADED '<' Operator
{ return volume() < aBox.volume(); } // Operator Overload BODY

```
if (box1 < box2) std::cout << "box1 is less than box2" << std::endl;   // Reason for the Operator Overloading, Need to Compare Objects
if (box1.operator<(box2))                                              // What happens now that the Operator '<' is Overloaded
 std::cout << "box1 is less than box2" << std::endl;
```

Defining trivial operator functions inside their class definition can be a good idea. Ensures they're inline, should maximize efficiency.

Nonmember Operator Functions 453

```
inline bool operator<(const Box& box1, const Box& box2)
```

```
{
 return box1.volume() < box2.volume();
}
```
// **■ Tip Always define nonmember operators in same namespace as the class of the objects they operate on.**

Implementing Full Support for an Operator 453

When implementing overloaded operators for a class, consider the range of circumstances in which the operator might be used.

bool operator<(double value) const; // Overloading Prototype…Same As::: **Box volume < double value**

Body of Operator Overloading Prototype

```
// Compare the volume of a Box object with a constant
inline bool Box::operator<(double value) const
{
 return volume() < value;
}
```

Implementing All Comparison Operators in a Class 456

■ Note Adding using statements to a header file is generally considered bad practice. The consequence of doing so is that names become available without qualification throughout every source file that includes this header file.

Operators That Can Be Overloaded 458

Table 12-1. Operators That Can Be Overloaded

Operators	Symbols	Nonmember
Binary arithmetic operators	+ - * / %	Yes
Unary arithmetic operators	+ -	Yes
Bitwise operators	~ & | ^ << >>	Yes
Logical operators	! && ||	Yes
Assignment operator	=	No
Compound assignment operators	+= -= *= /= %= &= |= ^= <<= >>=	Yes
Increment/decrement operators	++ --	Yes
Comparison operators	== != < > <= >=	Yes
Array subscript operator	[ ]	No
Function call operator	( )	No
Conversion-to-type-T operator	T	No
Address-of and dereferencing operators	& * -> ->*	Yes
Comma operator	,	Yes
Allocation and deallocation operators	new new[] delete delete[]	Only
User-defined literal operator	""_	Only

Restrictions and Key Guideline 459

- You cannot invent new operators such as ?, ===, or <>.
- You cannot change the number of operands, associativity, or precedence of the existing operators, nor can you alter the order in which the operands to an operator are evaluated.
- As a rule, you cannot override built-in operators, and the signature of an overloaded operator will involve at least one class type. We will discuss the term overriding in more detail in the next chapter, but in this case it means you cannot

modify the way existing operators operate on fundamental types or array types. In other words, you cannot, for instance, make integer addition perform multiplication. While it would be great fun to see what would happen, we're sure you'll agree that this is a fair restriction.

■ Tip The main purpose of operator overloading is to increase both the ease of writing and the readability of code that uses your class and to decrease the likelihood of defects. The fact that overloaded operators make for more compact code should always come second. Compact yet incomprehensible or even misleading code is no good to anyone. Making sure your code is both easy to write and easy to understand is what matters.

Operator Function Idioms 461

When an operator, Op, is overloaded and the left operand is an object of the class for which Op is being overloaded, the member function defining the overload is of the following form: **Return_Type operator Op(Type right_operand);**

For comparison operators such as =, and !=, for instance, ReturnType is typically bool (although you could use int)

You can implement most binary operators as nonmember functions as well, using this form:

Return_Type operator Op(Class_Type left_operand, Type right_operand);

If the left operand for a binary operator is of class Type, and Type is not the class for which the operator function is being defined, then the function must be implemented as a global operator function of this form:

Return_Type operator Op(Type left_operand, Class_Type right_operand);

General form of a unary operator function for the operation Op as a member of the Class_Type class is as follows:

Class_Type& operator Op();

Unary operators defined as global functions have a single parameter that is the operand. Prototype for a global operator function is:

Class_Type& operator Op(Class_Type& obj);

MOST COMMON Overloading the << Operator for Output Streams 462

auto& stream1 = operator<<(std::cout, theBox); // Takes 2 arguments: Ref & Value

(stream1 << " is greater than ") << box; // Reference to a stream object (left operand) & actual value to output (right operand). It then returns a fresh reference to a stream that can be passed along to the next call of operator<<() in the chain. This is an example of what we called **method chaining**

Overloading the Arithmetic Operators 463

Box operator+(const Box& aBox) const; // Adding two Box objects -> BoxA's Length , Width, Height + BoxB's L, W, & H

It's a const reference (**const** Box**&**) to avoid unnecessary copying of the right operand.

aBox parameter is const because the function won't modify the argument, which is the right operand. Box operator() **const**; //

```
// Operator function to add two Box objects
inline Box Box::operator+(const Box& aBox) const
{ // New object has larger length and width, and sum of heights
 return Box{ std::max(length, aBox.length),
 std::max(width, aBox.width),
 height + aBox.height };
}
```

Implementing One Operator in Terms of Another 467

Convenient Way to Add a whole Package of Boxes together:

```
int main()
{
 // Generate boxCount random Box objects as before in Ex12_05...
 Box sum{0, 0, 0}; // Start from an empty Box
 for (const auto& box : boxes) // And then add all randomly generated Box objects
 sum += box;
 std::cout << "The sum of " << boxCount << " random Boxes is " << sum << std::endl;
}
```

Member vs. Nonmember Functions 469

Addition operation as a nonmember function prototype:

Box operator+(const Box& aBox, const Box& bBox);

---

friend declarations undermine data hiding and should therefore be avoided when possible.

The default option should probably be to define operator overloads as member functions.

Implement Nonmember Functions when there is no other option OR you might sometimes prefer nonmember functions over member functions is when implicit conversions are desired for a binary operator's left operand.

```
bool operator<(double value, const Box& box); // double cannot have member functions
ostream& operator<<(ostream& stream, const Box& box); // you cannot add ostream members
bool operator<(double, const Box&); // Must be non-member function
bool operator<(const Box&, double); // Symmetrical case often done for consistency
bool operator<(const Box&, const Box&); // Box-Box case sometimes as well for consistency
```

Operator Functions and Implicit Conversions 470

■ Tip Operator overloads mostly should be member functions. Only use nonmember functions if a member function cannot be used or if implicit conversions are desired for the first operand.

Overloading Unary Operators 471

Overloading the Increment and Decrement Operators 473

MyClass& operator++(); // Overloaded prefix increment operator // '++Num'

const MyClass operator++(int); // Overloaded postfix increment operator // 'Num++'

```
inline Box& Box::operator++()
{
 ++length;      // The return type for the prefix form normally needs to be
 ++width;       // a reference to the current object, *this, after
 ++height;      // the increment operation has been applied to it.
 return *this;
}
```

```
inline const Box Box::operator++(int)
{
 auto copy{*this}; // Create a copy of the current object
 ++(*this); // Increment the current object using the prefix operator...
 return copy; // Return the unincremented copy
}
```

Overloading the Subscript Operator 474

Probably never use linked lists, std::vector<> is almost always better.

Modifying the Result of an Overloaded Subscript '[]' Operator 478

Never use the ^technique^ we used here in real programs.

Function Objects 480

function object- object of a class, overloads the function call operator (). A function object is also called a **functor**.

operator()()

■ Note Unlike most operators, function call operators must be overloaded as member functions. The function call operator is the only operator that can have as many parameters as needed & that can have default arguments.

Overloading Type Conversions 482

Potential Ambiguities with Conversions 483

conversion from type Type1 to type Type2 can be implemented by including a constructor in class Type2 with this declaration:

Type2(const Type1& theObject); // Constructor converting Type1 to Type2

This can conflict with this conversion operator in the Type1 class:

operator Type2(); // Conversion from type Type1 to Type2

The compiler will not be able to decide which constructor or conversion operator function to use when an implicit conversion is required. To remove the ambiguity, declare either or both members as **explicit**.

Overloading the Assignment Operator 483

Box& operator=(const Box& right_hand_side); // Copy Assignment Operator prototype

Implementing the Copy Assignment Operator 483

Default Assignment Operator copies members of the object right of an assignment to those of the same typed object on the left.

An assignment operator should return a reference, so in the Message class it would look like this:

Message& operator=(const Message& message); // Assignment operator

Copy Assignment vs. Copy Construction 487

■ Tip Every user-defined copy assignment operator should start by checking for self-assignment. Forgetting to do so may lead to fatal errors when accidentally assigning an object itself.

■ Tip If a class manages members that are pointers to free store memory, you must never use the copy constructor and assignment operator as is; and if it has members that are raw pointers, you must always define a destructor.

Assigning Different Types 488

Just remember, by convention any assignment operator should return a reference to *this

Summary 488

Make sure that your implementation of an overloaded operator doesn't conflict with what the operator does in its standard form.

You need to decide the nature and scope of the facilities each class should provide. Always keep in mind when defining a data type—a coherent entity—and that the class needs to reflect its nature and characteristics.

• You can overload any number of operators within a class for class-specific behavior. Do so to make code easier to read and write.
•**Overloaded operators** should mimic their built-in counterparts as much as possible.
Popular exceptions to this rule are the << and >> operators for Standard Library streams and the + operator to concatenate strings.
•**Operator functions** are members of a class or as global operator functions. You should use member functions whenever possible. Only resort to global operator functions if there is no other way or if implicit conversions are desirable for the first operand.
•For a **unary operator** defined as a class member function, the operand is the class object.
For a unary operator defined as a global operator function, the operand is the function parameter.
•For **binary operator functions** as a member of a class, the left operand for class object, the right operand for function parameter.
Binary operators defined by a global operator function, first parameter specifies left operand, second parameter is right operand.
•Functions implementing overloading of **+= operator** can be used to implement the **+ function**. This is true for all op= operators.
•To overload the increment or the decrement operator, you need two functions to provide prefix and postfix form. The function to implement a postfix operator has an extra parameter of type int that serves only to distinguish the function from the prefix version.
•To support customized type conversions, you have the choice between conversion operators or a combination of conversion constructors and assignment operators.

■Chapter 13: Inheritance 491

Inheritance- the means of creating new classes by reusing/expanding on existing class definitions. Fundamental to polymorphism.

Classes and Object-Oriented Programming 491

OOP Problem-Solving Steps – 1st identify the Types & Characteristics & Operations of entities related to the problem.
2nd define Classes & their operations, provides the Objects of classes that solves the problem.
-Anything can be a class. EX: Complex Mathematical Algorithms or something to represent a physical object, like a Tree or Truck.

Hierarchies 492

Hierarachy Determination RULE OF THUMBs- Derived Relationships: Class B **is a** OR (**is apart of**) Class A.
Parent Relationships: Class A **has a** Class B.

Unified Modeling Language(UML)[notation]- The standard way to indicate relationships with a diagram by using an arrow pointing toward the more general class in the hierarchy. Used for Visualizing the design of object-oriented software programs.

Specialization- Adding extra Properties to a Derived class. Given a class A, suppose you create a new class B that is a specialized version of A. Class A is the **Base, Parent, or Superclass**. Class B is the **Derived, Child, or Subclass**.
-A derived class inherits almost all of the member variables and member functions of its base class & Needs it's own special ones.

If class B is a derived class defined directly in terms of class A, then class A is a **Direct Base Class** of B. Class B is derived from A. If Class C is derived from Class B, Class A is an **Indirect Base Class** to Class C.

Inheritance in Classes 493

Derived class objects should be sensible specializations of base class objects, which can be tested with the **"is a "** Test & double-checked by asking whether there is anything we can say about (or demand of) the base class that's inapplicable to the derived class.

Such as Ostriches are Birds, but cannot fly. So Birds can be split into FlyingBirds & NonFlyingBirds, and then categorize Ostrich.

Inheritance vs. Aggregation 493

If classes fail the is a test, then you probably shouldn't use class derivation. In this case, you could check the "**has a**" test, which passes if it contains an instance of another class.

Accomodated by including second class as a member variable of the first. Automobile object has Engine object as a member variable;

Aggregation- child object existence is independent on an parent object. EX: ENGINES don't need Automobiles.

Composition- child object existence is dependent on an parent object. EX: ROOMs need a HOUSE

Sometimes, Derived Class is used to assemble a set of capabilities, as a packaging a given set of functions.

TempFunc.Humidity();

Deriving Classes 494

Class B gets everything from class A, & specialized functions.

protected Members of a Class 497

protected- base class members protected from outside interference. -Accessed from **friend** functions & derived class, whereas the private members of the base class are not.

■ Tip Member variables should normally always be private. Protected variables has issues as public member variables, only lesser

The Access Level of Inherited Class Members 498

This specifies Box base class as public: **class Carton : public Box{}**. 3 possibilities for the base class access specifier: **public, protected,** or **private**.

class Carton : Box // defaults to the private access specifier for Box.

Base class access specifiers affects the access status of the inherited members in a derived class. There are nine possible combinations.

private base class member *always* remains private to the base class

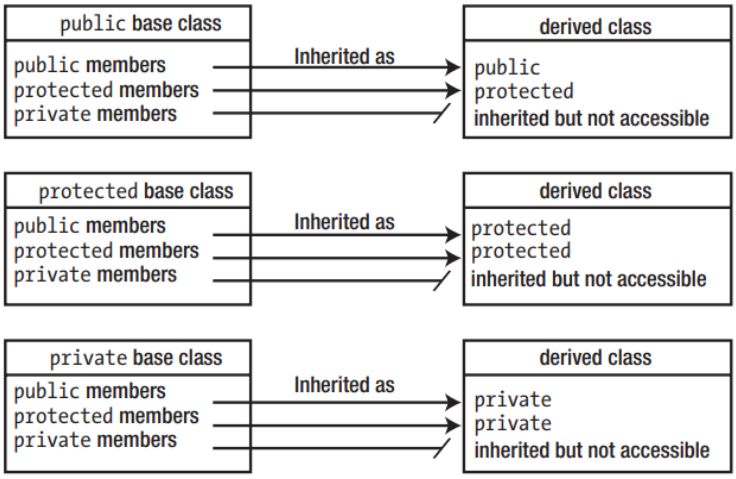


Figure 13-3. *The effect of the base class specifier on the accessibility of inherited members*

Remember that you can only make the access level more stringent; you can't relax the access level that is specified in the base class.

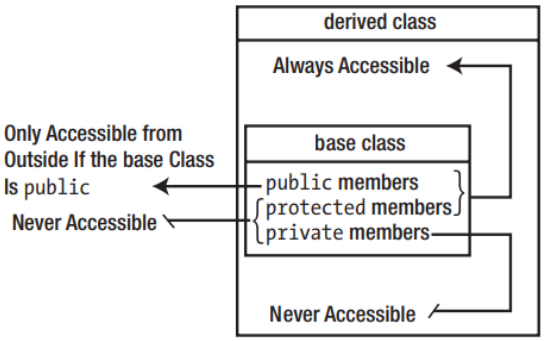


Figure 13-4. *The effect of access specifiers on base class members*

Access Specifiers and Class Hierarchies 499

By using the protected and private base class access specifiers, you are able to do two things:

- Prevent access to public base class members from outside the derived class. If the base class has public member functions, then this is a serious step because the class interface for the base class is being removed from public view in the derived class.
- Affect how the inherited members of the derived class are inherited in another class that uses the derived class as its base.

In practice, because the derived class object is a base class object, you'll want the base class interface to be inherited in the derived class, and this implies that the base class must be specified as **public**.

Choosing Access Specifiers in Class Hierarchies 500

■ **Tip** As a rule, member variables of a class should always be private. If code outside of the class requires access to member variables, you should add public or protected getter and/or setter functions. (This guideline commonly does not extend to structures. structs mostly do not encapsulate any member functions at all; the only members they typically have are public member variables.)

protected member variables have many of the same disadvantages as public ones:

- There is nothing stopping a derived class from invalidating an object's state, which may for instance invalidate so-called class invariants—properties of an object's state that should hold at all times—counted on by code in the base class.
- Once derived classes directly manipulate the member variables of a base class, changing its internal implementation becomes impossible without changing all of the derived classes as well.
- Any extra code added to public getter and setter functions in the base class becomes void if derived classes can bypass it.
- Breaking a debug session when member variables are modified becomes, at the least, more difficult if derived classes can access them directly, breaking when they're read impossible.

Therefore, always make Member Variables & Functions private, unless you have a good reason not do so.

Changing the Access Specification of Inherited Members 501

```
class Carton : private Box      // Inherits as private Box, so Carton gets Box variables as private.
```

```
{
private:
 std::string material;  // Specialized member variable
public:
 using Box::volume;  //!! Overrides Inheritance to public. Can now use volume();
 explicit Carton(std::string_view mat = "Cardboard") : material {mat} {}      // Constructor
};
```

First, when you apply a **using** declaration to a base class member, Qualify the member with the base class name, it specifies the context for the member name. Second, note that you don't supply a parameter list or a return type for a member function—just the qualified name. Implies that overloaded functions always come as a package deal. Third, the using declaration also works with inherited member variables in a derived class.

can't apply using declaration to relax a private member of a base class because private members cannot be accessed in a derived class.

Constructors in a Derived Class 502

Derived class objects are always created in the same way, even when there are several levels of derivation. The most base class constructor is called first, followed by the constructor for the class derived from that, followed by the constructor for the class derived from that, and so on, until the constructor for the most derived class is called.

- You can call a particular base class constructor in the initialization list for the derived class constructor. This enables you to use a constructor other than the default & allows you to choose a particular base class constructor, depending on the data supplied to the derived class constructor.

■ Note The notation for calling the base class constructor is the same as that used for initializing member variables in a constructor

```
// Constructor that won't compile! //Accessed from  derived class, can't be initialized in initialization list for a derived class constructor
Carton::Carton(double lv, double wv, double hv, std::string_view mat)
 : length{lv}, width{wv}, height{hv}, material{mat}
 { std::cout << "Carton(double,double,double,string_view) called.\n"; }
```

```
// Constructor that will compile!
Carton::Carton(double lv, double wv, double hv, std::string_view mat) : material{mat}
{        //Doesn't use Initializer List  //Creates Item that links to Box and then Sets Members
 length = lv;
 width = wv;
 height = hv;
 std::cout << "Carton(double,double,double,string_view) called.\n";
}
```

The Copy Constructor in a Derived Class 506

```
// Copy constructor
Box(const Box& box) : length{box.length}, width{box.width}, height{box.height}
{ std::cout << "Box copy constructor" << std::endl; }
```

```
// Copy constructor for Derived Carton Class // WRONG
Carton(const Carton& carton) : material {carton.material}
{ std::cout << "Carton copy constructor" << std::endl; }
```

```
// Copy constructor for Derived Carton Class
Carton(const Carton& carton) : Box{carton}, material{carton.material} //The Box copy constructor is called with the carton object as
 { std::cout << "Carton copy constructor" << std::endl; }   //an argument. The carton object is type Carton, but also a Box object.
```

The Default Constructor in a Derived Class 508

Carton() = default; // Tell the compiler to insert a default constructor in any event using the default keyword, even though there

definition that the compiler supplies for a derived class calls the base class constructor

Carton() : Box{} {}; // are other constructors

Every derived class constructor calls a base class constructor.

If a derived class constructor does not explicitly call a base constructor in its initialization list, the no-arg constructor will be called.

Inheriting Constructors 508

Cause Constructors to be Inherited from a direct Base class by putting a using declaration in the Derived class:

```
class Carton : public Box
  {    using Box::Box; // Inherit Box class constructors into Carton
  public:
       Carton(double lv, double wv, double hv, std::string_view mat)      //Takes 4 Params to make a Carton
       : Box{lv, wv, hv}, material{mat}                                   //Passes 3 Params to Box and 1 to Specialized Member
       { std::cout << "Carton(double,double,double,string_view) called.\n"; }
  };
```

```
Carton(double lv, double, wv, double hv) : Box {lv, wv, hv} {}        // More Examples
explicit Carton(double side) : Box{side} {}                           // More Examples
```

Each inherited constructor has the same parameter list as the base constructor and calls the base constructor in its initialization list.

Default constructors are never inherited.

Destructors Under Inheritance 509

--Destroying a derived class object involves both the derived class destructor and the base class destructor. IN CONSTRUCTION: Class A Instance creates Class B creates Class C. IN DESTRUCTION:Class C instance destroys Class B destroys Class A.

-This inherited Construction/Destruction hierarchy prevents invalid pointers/references.

The Order in Which Destructors Are Called 511

Duplicate Member Variable Names 511

It's possible that a base class and a derived class each have a member variable with the same name.

If the Base Class has a variable Value & Derived Class has a variable Value to distinguish them: **Base::Value** ||

Derived::Value

The Scope Operator can be used to define the exact space a variable resides.

Duplicate Member Function Names 512

A derived class member function will *hide* an inherited member function with the same name. Can be used like Function Overloading.

If the Base Class has a function DoIt(); & Derived Class has a function DoIt(); Same Thing: **Base::DoIt** ||

Derived::DoIt

Derived object; // Object declaration (Derived Instance)

object.Base::DoIt(3); // Derived Object calling Base DoIt

Multiple Inheritance 513

Multiple Inheritance- A derived class can have as many direct base classes as an application requires, vs

Single Inheritance- 1 direct base classes Multiple Inheritance isn't Commonly used.

Multiple Base Classes 513

Mixin programming- Multiple base classes are used together to form a composite object. Usually for convenience.

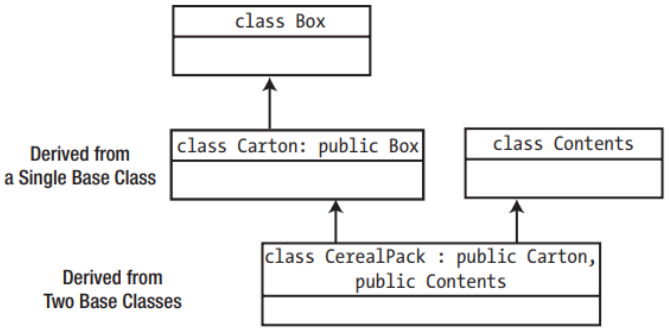


Figure 13-8. *An example of multiple inheritance*

Each base class is specified after the colon in the class header, and the base classes are separated by commas. Each base class has its own access specifier, and if you omit the access specifier, private is assumed, the same as with single inheritance.
The CerealPack class will inherit all the members of both base classes, so this will include the members of the indirect base, Box.

The definition of the CerealPack class looks like this:

```
class CerealPack : public Carton, public Contents
{
  // Details of the class...
};
```

Inherited Member Ambiguity 514

Multiple inheritance can create problems. When writing classes for use in inheritance, you should avoid duplicating member names in the first instance. The ideal solution to this problem is to rewrite your classes. IF NOT, then you would be forced to qualify the function names in main(). Like so:

std::cout << "cornflakes volume is " << **cornflakes.Carton::volume()** << std::endl

<< "cornflakes weight is " << **cornflakes.Contents::getWeight()** << std::endl;

Alternative way of ^^ is by adding casts to a reference to either of the base classes (we cast to a reference and not the class type itself to avoid the creation of a new object):

std::cout << "cornflakes volume is " << **static_cast<Carton&>(cornflakes).volume()**

<< std::endl << "cornflakes weight is " << **static_cast<Contents&>(cornflakes).getWeight()**;

Yet Another way with the **using** statement: BEST OPTION- disambiguate the inheritance in the class definition to save the users of your class the hassle of fighting against the compiler errors that would otherwise surely follow.

```
class CerealPack : public Carton, public Contents
{ public:       // Constructor and destructor as before...
      using Carton::volume;
      using Contents::getWeight;
};              //Which would be Called this way:
```

---

std::cout << "cornflakes volume is " << **cornflakes.volume()** << std::endl

<< "cornflakes weight is " << **cornflakes.getWeight()** << std::endl;

Repeated Inheritance 519

-Another ambiguity in multiple inheritances is when a derived object contains multiple versions of a subobject of 1 of the base classes.

-Possible to have duplication of an indirect base class inheritance. Class B & C use A, but Class D uses B & C.

-The complications and ambiguities that arise from such repeated inheritance are often referred to as the **diamond problem.**

Virtual Base Classes 520

To avoid duplication of a base class, identify to the compiler the base class should only appear once in a derived class by specifying the class as a **virtual base class** using the **virtual** keyword. The Contents class would be defined like this:

```
class Contents : public virtual Common
{  ...Code…
};
```

The Box class would also be defined with a virtual base class:

```
class Box : public virtual Common // Any class that uses the Contents and Box classes as direct or indirect bases will inherit the other
{ ...Code                         // members of the base classes as usual but will inherit only 1 instance of the Common class.
};                  // No qualification of the member names is needed when referring to them in the derived class.
```

Converting Between Related Class Types 521

Carton carton{40, 50, 60, "fiberboard"}; // Definition of Carton Object

---Conversions from a derived type to its base are always legal and automatic.

Box box{carton}; // Use ***copy constructor***

Box box; // OR ***copy assignment***

box = carton; // Only the Box subobject part of carton is used; a Box object has no room for the Carton-specific member variables. This effect is called **object slicing**, as the Carton specific portion is sliced off, so to speak, and discarded.

Cast cornflakes to either Carton& or Contents& , since they both have members of the same name. It specifies which ones.

Common common{static_cast<Carton&>(cornflakes)}; //

Summary 522

Inheritance means to derive a class based on one or more existing classes & is fundamental characteristic of OOP & polymorphism.

- A **class** may be **derived** from one or more **base classes**, in which case the derived class **inherits** members from all of its bases.
- **Single inheritance-** deriving a class from a single base class. **Multiple inheritance**- deriving a class from 2+ base classes.
- **Access** to the inherited members of a derived class is controlled by two factors: the **access specifier** of member in the base class & the **access specifier** of the base class in the derived class declaration.
- A constructor for a derived class is responsible for initializing all members of the class, including the inherited members.
- Creation of a derived class object always involves the constructors of all of the direct and indirect base classes, which are called in sequence (from the most base through to the most direct) prior to the execution of the **derived class constructor**.

•A derived class constructor can, and often should, explicitly call constructors for its direct bases in the initialization list for the constructor. If you don't call one explicitly, the base class's default constructor is called. A copy constructor in a derived class, for
one, should always call the copy constructor of all direct base classes.
•A member name declared in a derived class, which is the same as an inherited member name, will hide the inherited member. To access the hidden member, use the scope resolution operator to qualify the member name with its class name.
•You can use **using** not only for type aliases but to inherit constructors (always with the same access specification as in the base class), to modify the access specifications of other inherited members, or to inherit functions that would otherwise be hidden by a derived class's function with the same name but different signature.
•When a derived class with 2+ direct base classes contains two or more inherited subobjects of the same class, the duplication can be prevented by declaring the duplicated class as a **virtual base class**.

Chapter 14: Polymorphism 525

Using Virtual Functions

Understanding Polymorphism 525

Polymorphism- Always uses a pointer OR a reference to an object to call a member function.
Polymorphism only operates with classes that share a common base class.

Using a Base Class Pointer 525

You can regard every derived class object as a base class object. Can always use a pointer-to-a-base class to store the address of a derived class object; in fact, you can use a pointer to any direct or indirect base class to store the address of a derived class object.

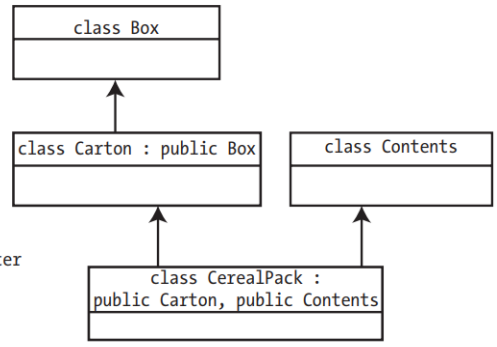
```
CerealPack breakfast;

// You can store the address of
// the breakfast object
// in any base class pointer:

Carton* pCarton {&breakfast};
Box* pBox {&breakfast};
Contents* pContents {&breakfast};

// You can store the address of
// a Carton object in a base pointer

Carton carton;
pBox = &carton;
```




```
// Object Pointers always Point up the Hierarchy.
// ClassC* -> ClassB  ||  ClassC* -> ClassA
// Class A makes Class B, but Class B doesn't make Class A.
//EX: ClassA* morphicObject = &ClassCObject
//morphicObject can now redefine/use ClassA Members.

//EX: SuperClass* NewObject = &SubClass
// NewObject is a Derived Object. From Super-to-SubClass
```

Figure 14-1. *Storing the address of a derived class object in a base class pointer*

Static Type= a pointer-to-***base*** class

D ynamic Type, which is whatever it is pointing to, which can also change. When pBox points to a Carton object, Dynamic Type = pointer-to-Carton. When pBox points to a ToughPack object, its dynamic type is a pointer to ToughPack.

Use *pBox* pointer to call a function defined in both base class & in each derived class & have the function called selected at runtime on the basis of the dynamic type of pBox.
-Polymorphism is good for situations that can be determined only at runtime. These arise frequently where the specific type of an object cannot be determined in advance (at design time or compile time).

Calling Inherited Functions 527

Static Resolution(**Static Binding**) (**Early Binding**)- PROBLEM- When compiler sets base class function as the fixed function to be used, instead of the derived function. EX: BASE volume() function called by DERIVED showVolume()

function, the compiler sets it once and for all as the version of volume() defined in the base class. No matter how you call Derived Function, it's a fixed base funct.

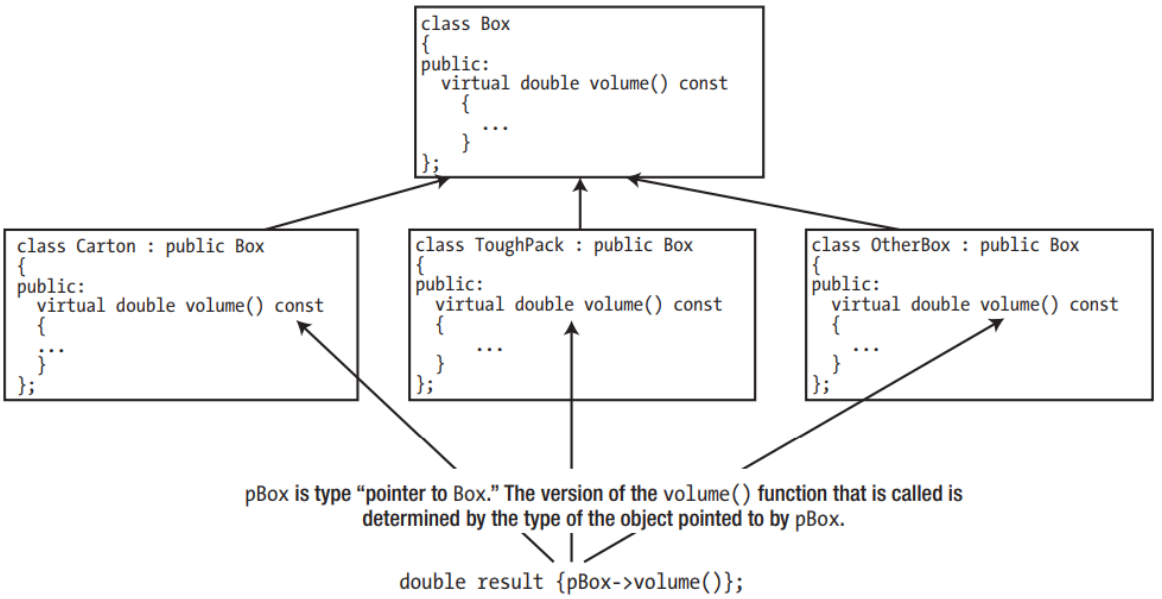
■ Note Any call to a function through a base class pointer that is resolved statically calls a base class function.
A static function call through a pointer is determined by the **static type** rather than the *dynamic type*.

When the volume() function is called through a base class pointer, we want the volume() function that is appropriate to the Derived object pointed to in order to be called. This sort of operation is referred to as **dynamic binding** (**late binding**).

To make this work, we have to tell the compiler that the volume() function in Box & any overrides in derived classes are special, & calls to them are resolved dynamically. Specify volume() in the base class is a **virtual function**, which will result in a **virtual function call** for volume().

Virtual Functions 530

To use the derived functions, rather than the base class functions, specify the base class function as **virtual**.
A **polymorphic** class means it's a derived class that has 1+ virtual functions.



--virtual in a base class will be virtual in all Derived classes.
--Polymorphic behavior= each derived class implementing its own version of the virtual function.
*-Make virtual function calls using a variable whose type is a pointer or a reference to a base class object
*-The type of the object to which the pointer points when the call executes determines which volume() function is called
*- A call to a virtual function using an object is always resolved statically.
*- Get Polymorphic calls to virtual functions through a pointer or a reference.

virtual double volume() const { return length * width * height; } THIS MAKES POLYMORPHISM Possible!!!!

-- Adding virtual in front of the function makes it a virtual function. Makes all inherited functions virtual.

■ Caution If a member function definition is outside the class definition, you can't use virtual functions.

// Now using a base pointer... THIS MAKES POLYMORPHIC CALLS!!!!

```
Box* pBox {&box};   // Points to type Box
std::cout << "\nbox volume through pBox is " << pBox->volume() << std::endl;
pBox->showVolume();
pBox = &hardcase; // Points to type ToughPack          // Equivalent to: Box* pBox { &hardcase };
std::cout << "hardcase volume through pBox is " << pBox->volume() << std::endl;
pBox->showVolume();
pBox = &carton; // Points to type Carton               // Equivalent to: Box* pBox { &carton };
```

```
std::cout << "carton volume through pBox is " << pBox->volume() << std::endl;
pBox->showVolume();
```

For a function to behave "virtually," its definition in a derived class must have the exact same signature as it has in the base class.

■ Note static member functions cannot be virtual.

Protect against errors by using the **override** specifier for every virtual function declaration in a derived class, like this:

```
class Carton : public Box
{
public:                                          // override For Safety!!
   double volume() const override                //This Carton volume function overrides the Box volume function.
   {     // Function body as before..
   }
};                  //The override specification, like the virtual one, only appears within the class definition.
```

override specification causes the compiler to verify that the base class declares a class member that is virtual & has same signature.

■ Tip Always add an override specification to the declaration of a virtual function override. First, this guarantees that you have not made any mistakes in the function signatures at the time of writing. Second, and perhaps even more important, it safeguards you & your team from forgetting to change any existing function overrides if the signature of the base class function needs to be changed.

Using final

final keyword- prevents member functions from being overridden in a derived class, even if it tries to override it won't.

double volume() const override **final**{}

You add **virtual** to allow function overrides, and you add **final** to prevent them.

You can also specify an entire class as final, like this: **class Carton final : public Box{}** // NO more Derived classes from Carton

Virtual Functions and Class Hierarchies 536

To get a function to be treated as virtual when it's called using a base class pointer, declare it as virtual in the base class, even if they do not repeat the virtual keyword.

You can call volume() for objects of any of these class types through a pointer of type Box* because the pointer can contain the address of an object of any class in the hierarchy.

A pointer pCarton, of type Carton*, could also be used to call volume(), but only for objects of the Carton class and the two classes that have Carton as a base: Crate and Packet.

In the Opposite Aspect, you cannot call doThat() using a pointer of type Box* because Box class doesn't define the function doThat().

Access Specifiers & Virtual Functions 537

-The access specification of a virtual function in a derived class can be different from the specification in the base class.

-If the virtual function is public in the base class, it can be called for any derived class through a pointer (or a reference) to the base class, regardless of the access specification in the derived class.

-When you use a class object, the call is determined statically by the compiler.

-Access specifiers determine if a function can be called based on the static type of an object. Changing an access specifier of a function override to a restricted one of the base class function is futile. Access Restriction is bypassed by using a pointer to the base class.

■ Tip A function's access specifier determines whether you can call a function, not whether you can override it. You can override a private virtual function of a given base class. In fact, it is often recommended that you declare your virtual functions private.

-**private virtual functions** give you the best of two worlds. 1st- the function is private & 2nd the function is virtual, allowing derived classes to override and customize its behavior.

Classic object-oriented design patterns, most prominently, **template method** pattern, best with using private virtual functions.

Default Argument Values in Virtual Functions 539

Base class declaration of a virtual function with a default argument value & function is called through a base pointer, you'll get the default argument value from the base class version of the function. default argument values in derived classes will have no effect.

```
virtual double volume(int i=5) const  // int i will always have a value of 5, if it's in the base class.
{   std::cout << "Box parameter = " << i << std::endl;
    return length * width * height;
}
```

Using References to Call Virtual Functions 541

--Calling virtual functions through reference parameters is a powerful tool for polymorphism, when functions use pass-by-reference.

-- You can pass a base class object or any derived class object to a function with a parameter that's a reference to the base class.

```
void showVolume(const Box& rBox)                    // Function Parameters as const Bass Class Reference.
{ std::cout << "Box usable volume is " << rBox.volume() << std::endl;}      //-----
int main()
{ Box box {20.0, 30.0, 40.0}; // A base box                                  //INSTANTIATION
 ToughPack hardcase {20.0, 30.0, 40.0}; // A derived box - same size        //----
 Carton carton {20.0, 30.0, 40.0, "plastic"}; // A different derived box    //----
 showVolume(box); // Display volume of base box                            //Calling a Function with a Reference.
 showVolume(hardcase); // Display volume of derived box                    //----
 showVolume(carton); // Display volume of derived box                      //----
}
```

//Each time showVolume() function is called, the reference parameter is initialized with the object that is passed as an argument. Because the parameter is a reference to a base class, the compiler arranges for dynamic binding to the virtual volume() function.

Polymorphic Collections 542

Polymorphic || Heterogeneous Collections of objects- name for the collections of base class pointers that contain objects with different dynamic types.

```
std::vector<Box> boxes;   //-→vector of Box, no smart pointer
boxes.push_back(Box{20.0, 30.0, 40.0});          // Careful: First attempt at a mixed collection is a bad idea (object slicing!)
boxes.push_back(ToughPack{20.0, 30.0, 40.0});    //…
boxes.push_back(Carton{20.0, 30.0, 40.0, "plastic"});      //…
for (const auto& p : boxes)                      //For Range Loop
p.showVolume();                                  // Call proves it's not polymorphic
std::cout << std::endl;                                  //…^THIS WHOLE CHUNK IS WRONG^
```

std::vector<std::unique_ptr<Box>> polymorphicBoxes; // CORRECT Polymorphic Vector<> // Base Class Pointer

polymorphicBoxes.push_back(std::make_unique<Box>(20.0, 30.0, 40.0)); // Equal to Box*(20.0, 30.0, 40.0) but Smart Pointer
polymorphicBoxes.push_back(std::make_unique<ToughPack>(20.0, 30.0, 40.0)); // Puts ToughPack in polymorphicBoxes[1]
polymorphicBoxes.push_back(std::make_unique<Carton>(20.0, 30.0, 40.0, "plastic")); // Holds any derived Box object
for (**const auto& p** : polymorphicBoxes) // Range-Based loop to iterate each Box object with a 'p' alias
p->showVolume(); // uses 'p' polymorphically, because it's a reference, shows volume of objects polymorfd.

■ Tip To obtain **memory-safe polymorphic collections** of objects, **Use standard smart pointers** such as std::unique_ptr<> and shared_ptr<> inside standard containers such as std::vector<> and array<>.

Destroying Objects Through a Pointer 543

--Using (smart)pointers to objects created in the free store, a Problem can arise when Derived Class Objects are Destroyed.
--The Base class Destructor is called for all objects, even if they are Derived Class Objects. The reason is that the Destructor Function is resolved Statically instead of dynamically. For the correct destructor to be called for a derived class, use **virtual destructor functions** for dynamic binding for the destructors.
--The C++ standard states that applying **delete** on a base class pointer to an derived class object results in undefined behavior & || memory leaks, unless that base class has a **virtual destructor**.
--**Virtual Class Destructors**- add the keyword virtual to the destructor declaration in the base class. This signals to the compiler that destructor calls through a pointer or a reference parameter should have dynamic binding. Like this:

virtual ~Box() { std::cout << "Box destructor called" << std::endl; } //Virtual Box Destructor

virtual ~Box() = default; //**BEST WAY** to do Virtual Destructors.

■ Tip When polymorphic use is expected (or even just possible), your class must have a virtual destructor to ensure that your objects are always properly destroyed. A Class with 1+ virtual member function should have a virtual destructor.

Converting Between Pointers to Class Objects 546

-Implicitly convert a pointer to a derived class to a pointer to a base class, and you can do this for both direct and indirect base classes:

auto pCarton{ std::make_unique<Carton>(30, 40, 10) };

Convert a pointer that is embedded in this smart pointer, implicitly, to a pointer to Box, which is a direct base class of Carton:

Box* pBox {pCarton.get()};

If you define a CerealPack class w/ Carton as public base class. Box is a direct base of Carton, so it is an indirect base of CerealPack:

CerealPack* pCerealPack{ new CerealPack{ 30, 40, 10, "carton" } };
Box* pBox {pCerealPack};

If you need to specify the conversion explicitly, you can use the **static_cast<>() operator**:

Box* pBox {static_cast<Box*>(pCerealPack)};

TIP: a pointer to a class type can only point to objects of that type or to objects of a derived class type and not the other way round.
Keep in mind of the Class Hierarchy while using pointers to avoid Object Slicing.

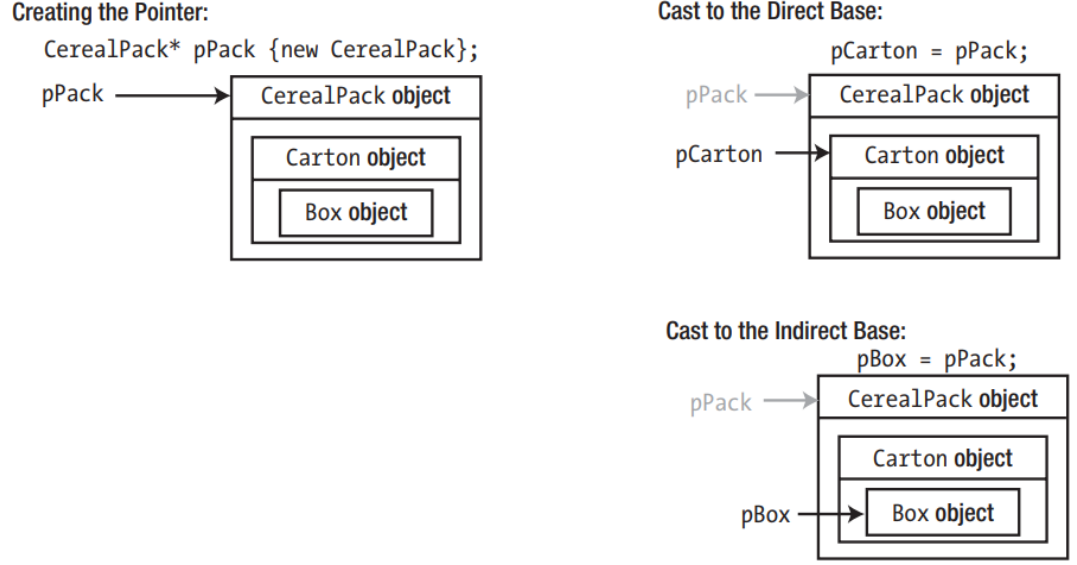


Figure 14-4. *Casting pointers up a class hierarchy*

Dynamic Casts 548

Dynamic cast- **dynamic_cast<>()** operator a conversion that's performed at runtime, can only apply these to pointers and references to-polymorphic class types, which are Abstract class types. This operator needs to check the validity of the conversion.

-This operator is only for converting between pointers or references to class types in the same hierarchy. Can't use for anything else.

Casting Pointers Dynamically 548

2 kinds of Dynamic Cast - **Downcast** -"cast down a hierarchy," from a pointer to a direct or indirect base type to a pointer to a derived type. **Crosscast** - cast across a hierarchy, still down though;

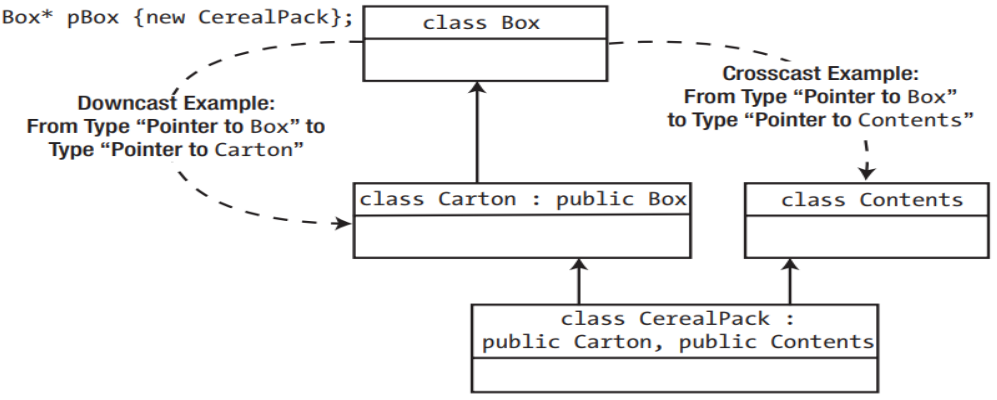


Figure 14-6. *Downcasts and crosscasts*

Downcast For a pointer, pBox, of type Box* that contains the address of a CerealPack object: //ct

Carton* pCarton {dynamic_cast<Carton*>(pBox)}; //Convert pBox from Box* pBox{} to Carton* pBox{} for an Object

Equivalence after casted= Carton* pCarton {&pBox} //

The **Crosscast** in Figure 14-6 could be written as follows:

Contents* pContents {dynamic_cast<Contents*>(pBox)}; //Convert pBox from Carton* pBox{} to Contents* pBox{}

A base class pointer only calls virtual member functions of a derived class, but dynamic_cast<>() enables calling nonvirtual functions. If surface() is a nonvirtual member function of the Carton class, you could call it with this statement:

dynamic_cast(pBox)->surface();

--To cast from a const pointer to a nonconst pointer, you must first cast to a non-const pointer of the same type as the original using the **const_cast<>() operator**, rarely recommended, Side-stepping const-ness often yields unexpected/inconsistent states.

```
void Base::DoSomething()
{ if (dynamic_cast<Derived*>(this)) // NEVER DO THIS!
    { /* do something else instead... */
 return;
}  //Downcasting a this pointer is never a good idea, so please don't ever do this!
```

Tip: In general, leveraging polymorphism may involve **template method design pattern,** splitting a function into multiple functions, some of which you can then override.

Converting References 551

Use dynamic_cast<>() operator to a reference parameter in a function to downcast class hierarchy to produce another reference.

```
double doThat(Box& rBox)
{ ...
 Carton& rCarton {dynamic_cast<Carton&>(rBox)}; // This statement casts from type reference to Box to type reference to Carton.
}  //If the object passed as an argument isn't a Carton object, the cast won't succeed.
```

```
        double doThat(Box& rBox)     //Dynamic casting to a reference blind is risky, the alternative:
        {
        Carton* pCarton {dynamic_cast<Carton*>(&rBox)};//Turn reference into a pointer & apply the cast to a pointer.
         if (pCarton) // Then you can again check the resulting pointer for nullptr:
         {
         }
        }
```

Calling the Base Class Version of a Virtual Function 552

What do you do when you actually want to call the base class function for a derived class object?

If you override a virtual base class function in a derived class, you'll often find that the latter is a slight variation of the former.

EXPLICITLY INSTRUCT THE COMPILER TO CALL THE BASE CLASS VERSION OF THE FUNCTION:

```
double volume() const override { return 0.85 * Box::volume(); } //Use Box volume(), instead of ToughPack volume()
```

Suppose you have a pointer pBox that's defined like this:

```
Carton carton {40.0, 30.0, 20.0};
Box* pBox {&carton};
```

```
double difference {pBox->Box::volume() - pBox->volume()}; //difference = Box::volume – Carton::Volume =R Box::volume
```

Calling Virtual Functions from Constructors or Destructors 553

Remember when an Derived object is constructed, Base Classes are constructed first, partially initializing the Derived Object, then the rest of the Derived Object with its constructor. Virtual function calls inside a constructor or destructor are always resolved statically.

■ **Caution** If in rare cases do need polymorphic calls during initialization, do so within an **init() member function**, often virtual, then call after the construction of the object has completed. This is called the **dynamic binding during initialization** idiom.

The Cost of Polymorphism 555 use the **sizeof** operator

--Polymorphism 2 CONS: Requires more Memory & Virtual Function calls have additional overhead.

The reason for More Memory: **vtable**- List of Virtual Functions with an index.

1. A special pointer to the **vtable** in the object pointed to finds the beginning of the vtable for the class.
2. The Function to be called is found in the vtable, usually by using an offset.
3. The Function Pointer indirectly calls through the vtable, which is a little slower than a direct call of a nonvirtual function.

---However, the overhead in calling a virtual function is small and shouldn't give you cause for concern.

■ **Note** The overhead of a **virtual function table pointer** isn't worth it when you have to manage millions of same type objects. A Point3D Class, representing a point in 3D space, can manipulate millions of such points. A Microsoft Kinect produces up to 9 million points per second, avoiding virtual functions in Point3D can save a significant amount of memory..

Determining the Dynamic Type 557

typeid() operator- determines the dynamic type of an object. It Returns reference to a **std::type_info** object that encapsulates the actual type of its operand.

Similar in use to the **sizeof** operator, the operand to the **typeid()** operator can be either an expression or a type.

typeid() operator can be an expression or a type. The semantics of the typeid() operator is:

- Operand of type, **typeid()** evaluates to a **reference to a type_info** object representing this type.
- If its operand is any expression that evaluates to a reference to a polymorphic type, this expression is evaluated, and the operand returns the dynamic type of the value referred to by the outcome of this evaluation.
- If its operand is any other expression, the expression is not evaluated, and the result is the static type of the expression.

typeid() - Enables easy type inspection of an expression OR Observe the Difference between an object's Static and Dynamic type.

■ Note To use the typeid() operator, include the **typeinfo header** from Standard Library. Makes **std::type_info** class available, the type of the object returned by the operator. There is an Underscore in the name of the type but not in that of the header.

```
int main()
{ std::cout << "Type double has name " << typeid(double).name() << std::endl; // Part 1: typeid() on types and == operator
 std::cout << "1 is " << (typeid(1) == typeid(int)? "an int" : "no int") << std::endl; // Part 1:
```

```
Carton carton{ 1, 2, 3, "paperboard" };                    // Part 2: typeid() on polymorphic references
 Box& boxReference = carton;                               // Part 2:
 std::cout << "Type of carton is " << typeid(carton).name() << std::endl;
 std::cout << "Type of boxReference is " << typeid(boxReference).name() << std::endl;
 std::cout << "These are " << (typeid(carton) == typeid(boxReference)? "" : "not ")
 << "equal" << std::endl;
```

```
Box* boxPointer = &carton;  // Part 3: typeid() on polymorphic pointers
 std::cout << "Type of &carton is " << typeid(&carton).name() << std::endl;
 std::cout << "Type of boxPointer is " << typeid(boxPointer).name() << std::endl;
 std::cout << "Type of *boxPointer is " << typeid(*boxPointer).name() << std::endl;
```

```
NonPolyDerived derived;      // Part 4: typeid() with non-polymorphic classes
 NonPolyBase& baseRef = derived;
 std::cout << "Type of baseRef is " << typeid(baseRef).name() << std::endl;
```

```
const auto& type_info1 = typeid(GetSomeBox()); // function call evaluated  // Part 5: typeid() on expressions
const auto& type_info2 = typeid(GetSomeNonPoly()); // function call not evaluated
 std::cout << "Type of GetSomeBox() is " << type_info1.name() << std::endl;
 std::cout << "Type of GetSomeNonPoly() is " << type_info2.name() << std::endl;
}
```

```
Box& GetSomeBox()
{  std::cout << "GetSomeBox() called..." << std::endl;
```

```
 static Carton carton{ 2, 3, 5, "duplex" };
 return carton;
}
NonPolyBase& GetSomeNonPoly()
{  std::cout << "GetSomeNonPoly() called..." << std::endl;
 static NonPolyDerived derived;
 return derived;
}
```

With some compilers, the type names that are returned are so-called **mangled names**, which are the names that the compiler uses internally. Hard for Humans to read.

■ **Note** To determine dynamic type of an object, typeid() operator needs so-called **runtime type information** (**RTTI**), normally accessed with the object's vtable. Because only objects of polymorphic types contain a vtable reference, typeid() can determine the dynamic type only for objects of polymorphic types. (This is also why dynamic_cast<> works only for polymorphic types.)

■ Caution Because this behavior of typeid() can be somewhat unpredictable—sometimes its operand is evaluated, sometimes it is not. Never include function calls in the operand to typeid(). Only apply this operator to variable names or types, to avoid bugs.

Pure Virtual Functions 561

// Virtual Functions **CAN** be Redefined in Derived Class. // Pure Virtual Functions **MUST** be redefined in the Derived Class.

class Shape // Generic base class for shapes // PURE Virtual Function are always: **virtual void vFunction() = 0;**

```
{ protected:                                  // Redefining Virtual Functions is Polymorphism
   Point position; // Position of a shape     // Because the constructor for an abstract class can't be used generally, ...
   Shape(const Point& shapePosition) : position {shapePosition} {}  //…it's a good idea to declare it as a protected member of the class
public:
 virtual ~Shape() = default;          // Remember: always use VIRTUAL DESTRUCTORS for ABSTRACT base classes!!!!
 virtual double area() const = 0;     // Pure virtual function to compute a shape's area
 virtual void scale(double factor) = 0;       // Pure virtual function to scale a shape
virtual void move(const Point& newPosition) { position = newPosition; };  // Regular virtual function to move a shape
};
```

Abstract Classes 561

Abstract Class - A class that contains 1+ pure virtual function & not allowed to Instantiate the Base class;

Used only to Derive Classes from it. A.k.A. **INTERFACE**.

Abstract Class Constructor- Initializes its member variables with Member Initiatilization List, which then is only used by the Derived Classes to make a Derived Object using the members of the Base Class.

Since you can't create Abstract objects, you cannot pass-by-value to a function; Also, can't return a Shape-by-value from a function.

For Polymorphic Behavior: Pointers OR References to an abstract class can be Parameters OR Return Types.

Defining Abstract Members as protected: Allows Initialization List call for Derived Constructor & prevents access to it otherwise.

If NOT DEFINED: Pure virtual function are inherited as Pure Virtual Functions(Undefined) & Derived Class will also be an Abstract Class.

Call Functions through a pointer-to-base class, and the calls will be resolved *dynamically*.

double volume() const override { return 0.85 * Box::volume(); }

--Box::volume() is now a pure virtual function. They can't call pure virtual function using static binding (it has no function body!). Because no base implementation is provided anymore, you'll again have to exchange the code back to: length * width * height

Abstract Classes as Interfaces 564

Interface- abstract class of only pure virtual functions.

```
int main()                                  // The base class is Vessel
{  Box box {40, 30, 20};                    // Makes Objects of type:Box, Can, Carton, ToughPack. All are Derived from
   Vessel.
   Can can {10, 3};                         // Objects made are: box, can, carton, hardcase. Now can be referenced w/ &
   Carton carton {40, 30, 20, "Plastic"};   // Polymorphism Possible: Derived Objects can be Referenced
   ToughPack hardcase {40, 30, 20};  // Sort Of: Vessel* vessels = &box  // But in Vector Syntax w/ multiple Instances
   std::vector<Vessel*> vessels {&box, &can, &carton, &hardcase};       // Vector of Type Vessel (Base Class can call any
   Derived Class)
   for (const auto* vessel : vessels)   // Vector name= vessels // Object Iterator= vessel
   { std::cout << "Volume is " << vessel->volume() << std::endl; }
}
```

Summary 567

- **Polymorphism-** calling a virtual member function of a class through a pointer OR reference and having the call resolved dynamically. The particular function to be called is determined by the object that is pointed to OR referenced during execution.
- A function in a base class can be declared **virtual**. All functions in classes, derived from the base, will then be virtual too.
- Always declare the destructor as virtual of classes as intended to be used as a base class as virtual (often done in combination

with **= default**). Ensures correct selection of a destructor for dynamically created derived class objects.

- Should use the **override** qualifier for each derived class member that overrides a virtual base class member. This causes the compiler to verify that the functions signatures in the base and derived classes are the same.
- **final** qualifier – can be used on a virtual function override to signal that it may NOT be overridden any further. If an entire class is specified to be **final**, no derived classes can be defined for it anymore.
- Default argument values for parameters in virtual functions are assigned statically. If default values for a virtual function exist, default values specified in a derived class will be ignored for dynamically resolved function calls.
- **dynamic_cast<>()**-Cast from pointer-to-**polymorphic**-**base**-class to a pointer-to-a-**derived**-class. If the pointer does not point to an object of the given derived class type, **dynamic_cast<>** evaluates to **nullptr**. This type check is performed dynamically, at runtime.
- **Pure Virtual Function** – has a member value set to 0. Used as an Interface. Derived Virtual Functions must redefine the Base's.
- **Abstract Class** - A class with 1+ pure virtual functions.

■Chapter 15: Runtime Errors and Exceptions 571

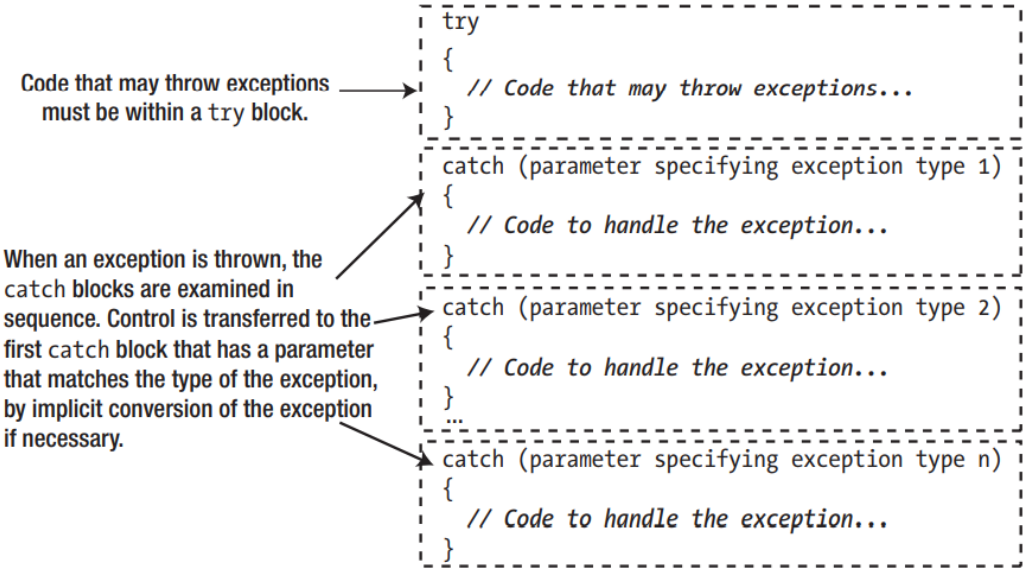


Figure 15-1. *A try block and its catch blocks*

try
{ // Code that may throw exceptions must be //in a try block...
if (test > 5)
throw "test is greater than 5";
// Throws an exception of type const char*
// This code only executes if the exception is //not thrown...
}
catch (const char* message**)**
{
// Code to handle the exception...
// ...which executes if an exception of type //'char*' or 'const char*' is thrown
std::cout << message << std::endl;
}

--Exceptions can be dictated by Flags. EX: If (int num/ 0) return -3; //which sends the message cannot divide by zero.

■ **GOLDEN RULE FOR EXCEPTIONS**- Always **THROW by value** and **CATCH by reference**, **ref-to-const** usually. Catch-by-value makes redundant copies & may Slice off exception object parts, which might slice off the info needed to diagnose which error occurred and why!

--Rethrow using a **throw;** statement.

--UNHANDLED EXCEPTIONS- the program essentially instantly crashes, std::terminate() is called after an **irrecoverable error**;

--**catch(...)**{ std::cout << "Oops. Something went wrong... Let's ignore it and cross our fingers..."; } <-Catches All Exceptions. Sweeps the errors under the rug. A program Crash is preferable over a Catch-All. Lazy Programming, unless Catch-alls that rethrow after some logging or cleanup, for instance, can be particularly useful.

-- **RAII idiom**, short for "**RESOURCE ACQUISITION IS INITIALIZATION**." Its premise, each time you acquire a resource you, do so by initializing an object. Memory in the free store is a resource.

■ Tip Even if you do not work with exceptions, it remains recommended to always use the RAII idiom to safely manage your resources.

Be safe: always, always use an RAII object!

Quite a few exception types are defined in the Standard Library, derived from the **std::exception** class that is defined in the **exception header**, and they all reside in the std namespace.

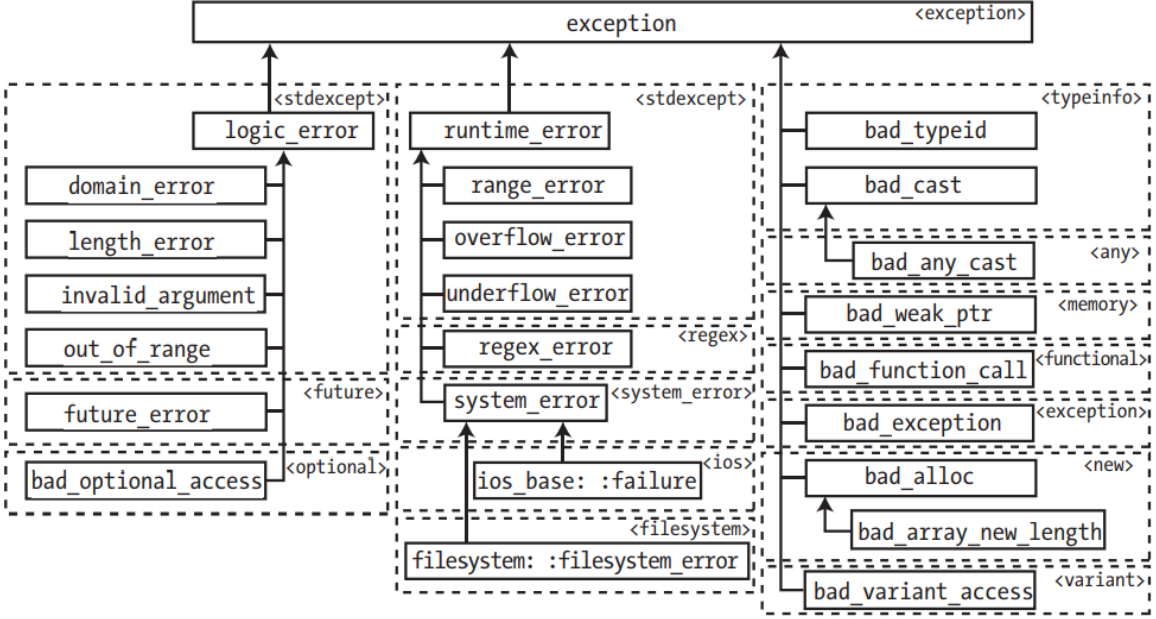


Figure 15-8. *Standard exception class types and the headers they are defined in*

Both these base classes, logic_error and runtime_error

Summary 609

- **Exceptions**: objects used to signal errors in a program.
- Code that may throw exceptions is usually contained within a **try block**, enables exceptions to be detected/processed.
- Code to handle exceptions that may be thrown in a try block is placed in one or more **catch blocks**, immediately following try block.
- A try block, along with its catch blocks, can be nested inside another try block.
- A catch block with a parameter of a base class type can catch an exception of a derived class type.
- **catch(…)** will catch an exception of any type.
- If an exception isn't caught by any catch block, **std::terminate()** function is called, which immediately aborts the program execution.
- **Every resource** should always be acquired and released by an **RAII object**. DON'T use keywords **new** & **delete** in modern C++ code.
- Standard Library offers various RAII types you should always use including: std::unique_ptr<>, shared_ptr<>, and vector<>.
- **noexcept** indicates a function doesn't throw exceptions. If a noexcept function does throw an exception, **std::terminate()** is called.
- If a destructor doesn't have an explicit noexcept specifier, the compiler will usually generate one. Never allow an exception to leave a destructor; otherwise, std::terminate() will be triggered.
- Std Library defines several exception types in **stdexcept header**, derived from **std::exception**, defined in exception header.

Chapter 16: Class Templates 613

Class templates are a powerful mechanism for generating new class types automatically & makes code easy to read & maintain,

it should be robust against unexpected conditions and exceptions, and so on.

Understanding Class Templates 614

Class templates: based on the same idea as function templates.

A class template is a **parameterized type**; it's a recipe for creating classes using one or more parameters.

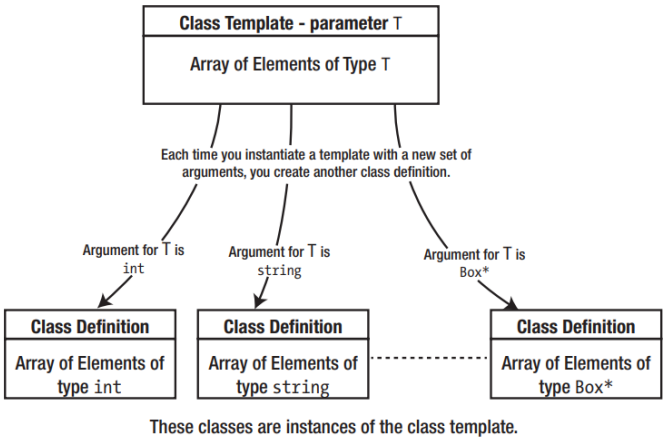


Figure 16-1. Instantiating a template

Many applications for class templates. Most commonly used to define **container classes**, classes that can contain sets of objects of a given type, organized in a particular way. LIke a Vector or Array.

Defining Class Templates 615

template <template parameter list> // The general form of a class template looks like this:
class ClassName
{ // Template class definition...
};

Template Parameters 615

Template parameter list can contain infinite parameters, only two types—**type parameters** & **nontype parameters**

TYPE PARAMETERS- such as int, std::string, or Box*. NONTYPE PARAMETERS- can be a literal, such as 200, an integral constant expression, a pointer or reference to an object, or a pointer to a function or a pointer that is null.

■ Note There's a third possibility for class template parameters. A parameter can also be a template where the argument must be an instance of a class template. A detailed discussion of this possibility is a little too advanced for this book.

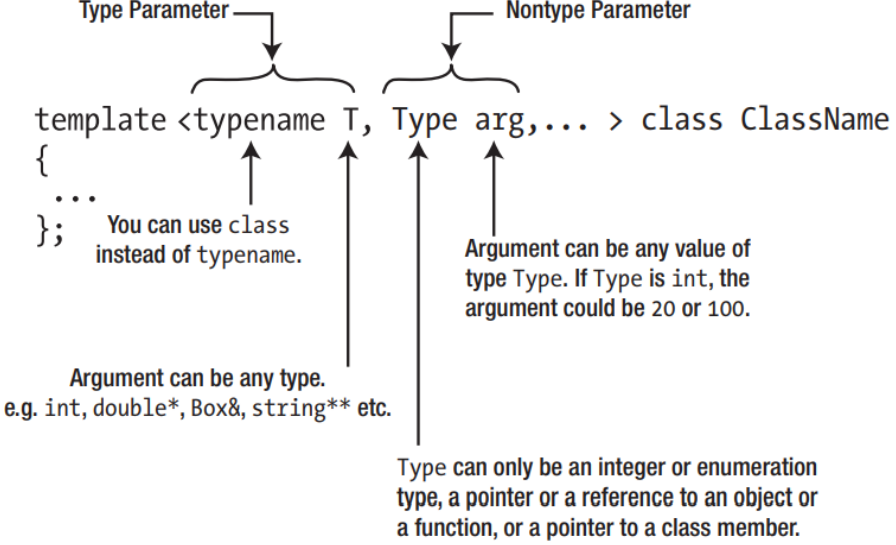


Figure 16-2. *Class template parameters*

T is often used as a type parameter name (or T1, T2, and so on, when there are several type parameters)

A Simple Class Template 616

Class Templates determine the type of the elements stored in an object of the return result.

The definition in the body of the template will be much the same as a class definition, with member variables and functions that are specified as public, protected, or private, and it will typically have constructors and a destructor.

■ Caution You must use the template ID to identify the template outside the body of the template. This will apply to member functions of a class template that are defined outside the template.

Defining Member Functions of a Class Template 618

All the member function definitions are templates, bound to the class template, they're Templates, not function definitions; Generated when needed by the compiler so that they are available in any source file that uses the template.

ALMOST ALWAYS put all the definitions of member functions for a class template in the header file that contains the class template itself.

Constructor Templates 618

Defining Constructor, its name must be qualified by the class template name, almost same as a member function of an ordinary class.

template <typename T> // This is a template with parameter T

Array<T>::Array(size_t arraySize) : elements {new T[arraySize]}, size {arraySize}

{} // *typename* keyword in the qualifier for the member name; it's only used in the template parameter list.

---

template <typename T> //If T is a class type, a public default constructor must exist in class T, or, the instance of this

Array<T>::Array(const Array& array) : Array{array.size} // constructor won't compile…

{ // The copy constructor has to create an array for the object being created that's the same size as that of its argument and then copy

for (size_t i {}; i < size; ++i) // the latter's member variables to the former.

elements[i] = array.elements[i];

}

The Destructor Template 619

```
template <typename T>
Array<T>::~Array()   // The destructor must release the memory for the elements array,
{
 delete[] elements;
}
```

Subscript Operator Templates 620

```
template <typename T>
T& Array<T>::operator[](size_t index)
{ // The operator[]() function is quite straightforward, but we must ensure illegal index values can't be used.
 if (index >= size)
 throw std::out_of_range {"Index too large: " + std::to_string(index)};
 return elements[index];
} //ALWAYS TRY TO AVOID CODE DUPLICATION
```

Code Duplication CONS: Not-Maintainable (Have to fix every duplication instead of just one function)

The principle of avoiding code duplication is also sometimes called the **Don't Repeat Yourself (DRY) principle**.

WAYS TO PREVENT CODE DUPLICATION: Functions are reusable blocks of computations and algorithms, templates instantiate functions or classes for any number of types, a base class encapsulates all that is common to its derived classes, and so on.

C++17 has introduced a little helper function, std::as_const(), that makes this code already a bit more bearable:

```
template <typename T>    //This Single Template overloads a const or non-const function, no Code duplication
T& Array<T>::operator[](size_t index)
{
 return const_cast<T&>(std::as_const(*this)[index]);
}
```

```
return const_cast<T&>(static_cast<const Array<T>&>(*this)[index]);
```

is the Equivalent of:

```
// dereferencing the this pointer gives us a reference of type Array&.
Array<T>& nonConstRef = *this; // Start from a non-const ref //The this pointer has a pointer-to-non-const type.
const Array<T>& constRef = std::as_const(nonConstRef); // Convert to const ref // call same function, with the same set of arguments
const T& constResult = constRef[index]; // Obtain the const result
return const_cast<T&>(constResult); // Convert to non-const result
```

■ **Tip** Use the **const-and-back-again idiom** to avoid code duplication of const & non-const overloads of a member function. In general, it works by implementing the non-const overload of a member in terms of its const counterpart using the following pattern:

```
ReturnType Class::Function(Arguments)
{
 return const_cast<ReturnType>(std::as_const(*this).Function(Arguments));
}
```

The Assignment Operator Template 622

■ Note Free store memory allocation is a rare occurrence these days because physical memory is large and because virtual memory is very large. So, checking for or considering bad_alloc is omitted in most code. Nevertheless, given that in this case we are implementing a class template whose sole responsibility is managing an array of elements, properly handling memory allocation failures does seem appropriate here.

■ **Tip** AS A RULE, assume that any function or operator you call might throw an exception and consequently consider how your code should behave if and when this occurs. The only exceptions to this rule are functions annotated with the **noexcept** keyword and most destructors, as these are generally implicitly noexcept.

Copy-and-Swap idiom- A programming pattern, used to for an all-or-nothing behavior for our assignment operator. If you have to modify the state of 1+ objects & any of the steps required may throw, follow this simple recipe:

1. Create a copy of the objects.

2. Modify this copy instead of the original objects. The latter still remain untouched!

3. If all modifications succeed, replace—or swap—the originals with the copies.

```
template <typename T> //If anything goes wrong either during the copy or any of the modification steps, simply abandon the
Array<T>& Array<T>::operator=(const Array& rhs) // copied, half-modified objects and let the entire operation fail. The
{                                                // original objects then remain as they were.
 if (this != &rhs)        // often used within a member function
 {
 Array<T> copy{rhs}; // Copy... (could go wrong and throw an exception)
 swap(copy); // ... and swap! (noexcept)
 }
 return *this;
}
```

■ **Tip** Implement the assignment operator in terms of the copy constructor & a noexcept swap() function. This basic instance of the copy-and-swap idiom will ensure an all-or-nothing behavior for assignment operators. swap() can be added as a member function, convention dictates making objects swappable involves defining a nonmember swap() function. This convention ensures that the swap() function gets used by various STDLIB algorithms. The copy-and-swap idiom can be used to make any nontrivial state modification exception safe, either inside other member functions or simply in the middle of any code. It comes in many variations, but the idea is always the same. First copy the object you want to change, then perform any number (zero or more) of risky steps onto that copy, and only once they all succeed commit the changes by swapping the state of the copy and the actual target object.

Instantiating a Class Template 627

```
Array<int> data {40};
```

---

implicit instantiation-The instantiation of a class template from a definition,a by-product of declaring an object.

explicit instantiation- of a template, which we'll get to shortly and which behaves a little differently.

-- Each time you define a variable using a class template with a different type argument, a new class is defined and included in the program. Because creating the class object requires a constructor to be called, the definition of the appropriate class constructor is also generated.

Note that a class template is only implicitly instantiated when an object of the specific template type needs to be created. Declaring a pointer to an object type won't cause an instance of the template to be created. Here's an example:

```
Array* pObject; // A pointer to a template type // No object of type Array<string> is created as a result of this statement
```

This defines pObject as type pointer to type Array<string>, so no template instance is created. Contrast the following statement:

```
Array<std::string*> pMessages {10}; //
```

Class Template Argument Deduction 632

```
template<typename T> T larger(T a, T b); // Function template prototype
```

```
int main()
{
 std::cout << "Larger of 1.5 and 2.5 is " << larger(1.5, 2.5) << std::endl;
 ...
```

Nontype Class Template Parameters 634

Templates for Member Functions with Nontype Parameters 636

■ Tip When you code, **MAKE SURE IT'S READABLE** to colleagues and even yourself later. WAYS TO: Rewrite Code for readability & understandability / Clarify by adding Comments

Arguments for Nontype Parameters 641

An argument for a nontype parameter that is not a reference or a pointer must be a compile-time constant expression.

Nontype Template Arguments vs. Constructor Arguments 641

Template arguments have to be compile-time constants.

code bloat -getting a lot more compiled code in your program than you might have anticipated

■ Note In principle, you could resort to advanced techniques such as adding member function templates to our Array<> class template to facilitate intermixing related instantiations of the Array<> template. These are templates for member functions that add extra template parameters—such as a different start index—on top of the two existing template parameters of the class.

Default Values for Template Parameters 642

Array<std::string, -100> messages {200}; // Array of 200 string objects indexed from -100

Explicit Template Instantiation 643

explicitly instantiate a class template without defining an object of the template type.

template class Array<double, 1>; //This creates an instance of the template that stores values of type double, indexed from 1.

■ Tip You can use explicit instantiation to quickly test whether a new class template and all its members instantiates for one or multiple types. It saves you the trouble of writing code that calls all the member functions!

Class Template Specialization 643

class template specialization- class definition specifically for a particular parameter. Like <typename T> vs <char>

Defining a Class Template Specialization 644

template <> // **COMPLETE SPECIALIZATION-** All the parameters of the template are specified ., which is why the set

class Array<const char*>{ // Definition of a class to suit type const char*...}; // of angle brackets following the template keyword is empty.

Partial Template Specialization 644

```
template <int start> // Because there is a parameter...
class Array<const char*, start>  // This is a partial specialization...
{                                // Definition to suit type const char*...
```

```
};
template <typename T, int start> // The first parameter is still T, but the T* between angle brackets following the template name
class Array<T*, start>           // indicates that this definition is to be used for instances where T is specified as a pointer type.
{       // Definition to suit pointer types other than const char*...
};
```

Choosing Between Multiple Partial Specializations 645

When several template specializations may fit a given declaration, the compiler uses the most specialized specialization.

Using static_assert() in a Class Template 645

use **static_assert()** makes compiler to fail compilation when a type argument in a class template is not appropriate.
static_assert() by default has two arguments; when the first argument is false, the compiler outputs the message specified by the second argument. If the second argument is omitted, a default message will be generated by the compiler.

type_traits **header**- Test properties of types & classify types in various ways.

Template	Result
is_default_constructible_v<T>	Is only true if type T is default constructible, which means for a class type that the class has a no-arg constructor
is_copy_constructible_v<T>	Is true if type T is copy constructible, which means for a class type that the class has a copy constructor
is_assignable_v<T>	Is true if type T is assignable, which means for a class type that it has an assignment operator function
is_pointer_v<T>	Is true if type T is a pointer type and false otherwise
is_null_pointer_v<T>	Is true only if type T is of type std::nullptr_t
is_class_v<T>	Is true only if type T is a class type

■ **Note** If Box class doesn't have a default constructor, the example will fail regardless of whether the static_assert() is added inside the Array<> template. Instantiated code attempts to use a default constructor that is not defined. Intricate diagnostic messages if the template instantiation fails. If the user of your class uses unsupported template arguments, it's courtesy to employ static_assert() declarations to improve compilation diagnostics. Note that the static_assert() declarations also double as code documentation, conveying usage intent to the person reading the code.

Friends of Class Templates 647

Friends of a class template can be classes, functions, or other templates.
If a class is a friend of a class template, then all its (member)functions are friends of every instance of the template.

```
class Box
{
 friend template <typename T > class Thing;
 // Rest of class definition...
};
```

Class Templates with Nested Classes 649

A class template definition can also contain a nested class or even a **nested class template**(Independently Parameterized).

Stack, a "last in, first out" storage mechanism for computer memory.

Has 2 operations- **push operation-**adds item at top of stack. **Pop operation-** removes item at top of stack.

Function Templates for Stack Members 651

```
template <typename T>
Stack<T>::Stack(const Stack& stack)
{
   if (stack.head)
   {
        head = new Node {*stack.head}; // Copy the top node of the original
        Node* oldNode {stack.head}; // Points to the top node of the original
        Node* newNode {head}; // Points to the node in the new stack
        while (oldNode = oldNode->next) // If next was nullptr, the last node was copied
        {
        newNode->next = new Node{*oldNode}; // Duplicate it
        newNode = newNode->next; // Move to the node just created
        }
   }
}
```

Disambiguating Dependent Names 656

The name of a member of a class template is said to be dependent on the template parameters of the class template.
As a rule of thumb, it is easiest to remember that whenever you're allowed to simply write Nested, you won't have to disambiguate the dependent name Outer::Nested either.

Summary 658

- **Class templates** define a family of class types.
- An **instance** of a *class template* is a **class definition**, generated by the compiler from the template using a set of template arguments that you specify in your code.
- **Implicit instantiation** of a class template arises out of a definition for an object of a class template type.
- **Explicit instantiation** of a class template defines a class for a given set of arguments for the template parameters.
- An *argument* for to a *type parameter* in a class template can be: fundamental type, class type, pointer type, or reference type.
- The type of a **nontype parameter** can be an integral or enumeration type, pointer type, or reference type.
- **Partial specialization** of class templates define a new template to be used specifically, restricted subset of the arguments for the original template.
- **Complete specialization** of a class template defines a new type for a specific, complete set of parameter arguments for the original template.
- A **friend** of a class template can be a function, a class, a function template, or a class template.
- An ordinary class can declare a class template or function template as a friend.

■Chapter 17: Move Semantics 661

move: Instead of copying objects; Efficiently transfers resources between Objects, without **deep copying**.

special class members—like: the default constructor, destructor, copy constructor, and copy assignment constructor.

Lvalues and Rvalues 662

a = b + c; // a = Lvalue // b + c = Rvalue

Lvalues on Left of assignment. Rvalues on Right of Assignment.

Lvalue = Persistent value (has a Memory Address) **Rvalue** = Temporary Value (The Space for operation b + c)

■ **Note** Most function call expressions are rvalues. Only function calls that return a reference are lvalues. One indication for the latter is that function calls that return a reference can appear on the left side of a built-in assignment operator just fine. Prime examples are the subscript operators (operator[]()) and at() functions of your typical container. If v is a vector, for example, both v[1] = -5; and v.at(2) = 132; would make for perfectly valid statements. v[1] and v.at(2) are therefore clearly lvalues.

TRICK: If the value that it evaluates to persists long enough for you to take and later use its address, then that value is an lvalue.

```
int* x = &(b + c); // Error!
int* y = &std::abs(a * d); // Error!
int* z = &123; // Error! //ALL NUMERIC LITERALS ARE Rvalues.
int* w = &a; // Ok!  // TYPICAL Lvalue Reference!
int* u = &v.at(2); // Ok! (u contains the address of the third value in v)
```

Rvalue References 663

Rvalue reference can be a variable alias, it differs from lvalue references because it can also reference the outcome of an rvalue expression, even though this value is generally transient. Being bound to an rvalue reference extends the lifetime of such a transient value. Its memory will not be discarded as long as the rvalue reference is in scope.

```
int count {5};
int&& rtemp {count + 3}; // rvalue reference – Uses '&&' for Referencing
std::cout << rtemp << std::endl; // Output value of expression
int& rcount {count}; // lvalue reference
```

Moving Objects 664

Traditional Workarounds 667

a **move constructor** - moves data into a new object, without actually copying them.

```
public:
 explicit Array(size_t arraySize); // Constructor
 Array(const Array& array); // Copy constructor
 Array(Array&& array); // Move constructor
```

```
// Move constructor
template <typename T>
Array<T>::Array(Array&& moved)
 : size{moved.size}, elements{moved.elements}
{
 std::cout << "Array of " << size << " elements moved" << std::endl;
 moved.elements = nullptr; // Otherwise destructor of moved would delete[] elements!
}
```

--**Shallow copy-** Copies all members of an object one by one, even if these members are pointers to dynamic memory.

--Deep copy- Copies all dynamic memory referred to by any of its pointer members as well.

Defining Move Members 668

Like copy constructors is mostly accompanied by a copy assignment operator, user-defined move constructors is typically paired with a user-defined move assignment operator.

Move Constructors 668

Explicitly Moved Objects 672

USING **std::move()**- Can turn any lvalue into an rvalue reference.

Move-Only Types 672

MOST COMMON TO MOVE: std::unique_ptr<>

---

std::unique_ptr<int> other;

other = one; // **ERROR**: copy assignment operator is deleted!

other = std::move(one); // Move assignment operator is defined **// CORRECT**

---

std::unique_ptr<int> yet_another{ other }; // **ERROR**: copy constructor is deleted!

std::unique_ptr<int> yet_another{ std::move(other) }; // Move constructor is defined **// CORRECT**

```
namespace std
{
   template <typename T>
   class unique_ptr
 { ...MoreCode...      // To define a move-only type, you always start by deleting its two copy members
   unique_ptr(const unique_ptr&) = delete; // Prevent copying:
   unique_ptr& operator=(const unique_ptr&) = delete; // Prevent copying:
 ----------------------------------------------------------------------------------------------------------------------------------------------------------------
   unique_ptr(unique_ptr&& source); // Allow moving:
   unique_ptr& operator=(unique_ptr&& rhs); // Allow moving:
};
```

Extended Use of Moved Objects 673

■ **Caution** As a rule, only move an object if it is Absolutely no longer required. Unless otherwise specified, you're not supposed to keep on using an object that was moved. By default, any extended use of a moved object results in undefined behavior(crashes).

■ **Tip** If need be, you can safely revive a move()'d vector<> by calling its clear() member. After calling clear(), the vector<> is guaranteed to be equivalent to an empty vector<> and thus safe to use again.

■ **Tip** The Standard Library specification clearly stipulates that you may continue using smart pointers of type std::unique_ptr<> and std::shared_ptr<> after moving out the raw pointer, and this without first calling reset(). For both types, move operations must always set the encapsulated raw pointer to nullptr.

A Barrel of Contradictions 675 A lot about Move Semantics is confusing or contradictory.

std::move() Does Not Move 675

std::move() does not move anything, it turns an lvalue into an rvalue reference. std::move() is alot like cast operators, EX: static_cast<>() & dynamic_cast<>(). A version of the simplified function:

```
template <typename T>                // move() makes an lvalue eligible for binding with the rvalue reference parameter of a move
T&& move(T& x) noexcept { return static_cast<T&&>(x); }          // constructor or assignment operator.
```

you can move() all you want; but if there's no function overload standing ready to receive the resulting rvalue, it's all in vain.

If no move assignment operator for Array<> to accept rvalue, copy assignment operator will be used.

An Rvalue Reference Is an Lvalue 676

the **name** of a named variable with an rvalue reference type is an lvalue.

To move the contents of a named variable, you must therefore always add std::move():

strings = std::move(rvalue_ref);

Defining Functions Revisited 676

How & When to pass arguments by rvalue reference to regular functions, not move constructors or move assignment operators.

Pass-by-Rvalue-Reference 676

```
template <typename T>
void Array<T>::push_back(const T& element)        //Avoids Redundant Copies    //VERY IMPORTANT!!!
{
  Array<T> newArray(size + 1);                    // Allocate a larger Array<>
  for (size_t i = 0; i < size; ++i)        newArray[i] = elements[i];     // Copy all existing elements...
  newArray[size] = element;                       // Copy the new one...
  swap(newArray);                                 // ... and swap!
}                 //INSTEAD OF COPY AND SWAP       VERY IMPORTANT
  for (size_t i = 0; i < size; ++i)        //Move all existing elements //Apply std::move() to turn the lvalue elements[i] into an rvalue to
  newArray[i] = std::move(elements[i]); // avoids copying, if template arg type T has a good move assignment operator:
```

When dealing with a: reference to a const T, meaning caller function expects a const argument. Moving its contents into another object is therefore completely out of the question.

an extra overload of push_back() that accepts rvalue arguments:

```
template <typename T>
void Array<T>::push_back(T&& element)
{...
 newArray[size] = std::move(element); // Move the new element...
}
```

The Return of Pass-by-Value 678

Move semantics are better because pass-by-lvalue-reference is no longer always your best option.

A way to work around duplication is to redefine the const T& overload in terms of the T&& one like so:

```
template <typename T>
void Array<T>::push_back(const T& element)
{
 push_back(T{ element }); // Create a temporary, transient copy and push that
}
```

BEST

```
template <typename T>
void Array<T>::push_back(T element)          // Pass by value (copy of lvalue, or moved rvalue!)
{...
 newArray[size] = std::move(element);        // Move the new element...
}
```

■ **Tip** For fundamental **types & pointers**, simply use **pass-by-value**. For **objects**(expensive copy), normally use a **const T& parameter**.

This avoids any lvalue arguments from being copied, and rvalue arguments will bind just fine with a const T& parameter as well.

If function copies its T argument, pass it by value instead, even when it concerns a large object. Lvalue arguments will then be copied when passed to the function, and rvalue arguments will be moved. This presumes the parameter types support move semantics—as all types should these days. Less likely case the parameter type lacks proper move members, stick with pass-byreference. Most types support move semantics, though—not in the least all Standard Library types—so pass-by-value is most certainly back on the table!

Return-by-Value 681

•Return statement: *return vName;* The compiler treats vName as an rvalue expression, if name is a locally defined automatic variable or that of a function parameter.

• Return statement: *return vName;* The compiler is *allowed* to apply the **named return value optimization (NRVO)**, provided name is the name of a locally defined automatic variable (not a parameter name).

•If returned object has no move constructor, adding std::move() causes the compiler to fall back to the copy constructor! Yes, that's right. Adding move() can cause a copy, where before the compiler would probably have applied NRVO.

•Even if the returned object can be moved, adding std::move() can only make matters worse—never better. The reason is that NRVO generally leads to code that is even more efficient than move construction (move construction typically still involves some shallow copying and/or other statements; NRVO does not).

■ **Tip** If value is a local variable (w/ automatic storage duration) OR function parameter, Never use return std::move(value); Always write return value; instead.

WHEN TO USE std::move()? *MOSTLY*, ***returning by value WITHOUT std::move() is what you want,*** but not always:

•*RARELY*: If the variable being returned has ***static*** or ***thread-local storage duration***, add std::move() if moving is what you want.

• When returning an ***object's member variable***, std::move() is again REQUIRED to prevent copies.

•If *return statement contains an lvalue expression* **besides the name of a single variable**, NRVO doesn't apply, the compiler won't treat this lvalue as an rvalue when looking for a constructor.

Conditional expression: *condition? var1 : var2* is an lvalue. It doesn't return a Value, so it is copied. **3 WAYS TO PREVENT**:

```
return std::move(condition? var1 : var2);    // PREVENTS COPIES
```

```
return condition? std::move(var1) : std::move(var2); // PREVENTS COPIES
```

```
if (condition)     // BEST WAY TO DO IT        // Allows a clever compiler to apply NRVO
 return var1;
else
 return var2;
```

Defining Move Members Revisited 683

With Custom move constructors & move assignment operators: Always declare them as **noexcept**

Without noexcept, your move members are not nearly as effective.

Always Add noexcept 683

■ Caution Standard Library **utility** header, **std::move_if_noexcept()**, function is to conditionally invoke the move constructor or copy constructor, if move constructor is noexcept. Std. Lib. has no equivalent for conditionally invoking a move assignment operator.

Implementing **std::move_if_noexcept()** requires **template metaprogramming**(Advanced Topic).

Template metaprogramming- Branches to specific Code based on the Input Data Type of the template arguments, at Compile Time.

---

how to convey logic using some of the *template metaprogramming* primitives, **type traits**, provided by the type_traits header:

std::conditional_t<std::is_nothrow_move_assignable_v<T>, T&&, const T&>

With the previous meta-expression, composing a fully functioning *move_assign_if_noexcept()* function is easy:

```
template<class T> //depending on whether T&& values can be assigned without throwing—the return type
std::conditional_t<std::is_nothrow_move_assignable_v<T>, T&&, const T&> // ^will be either T&& or const T&.
move_assign_if_noexcept(T& x) noexcept
{
 return std::move(x); // If the return type in the template's instantiation for type T is const T&, then the rvalue
}                     // reference that is returned gets turned right back into an lvalue reference.
```

Moving Within Standard Library Containers

all container types of the Standard Library are optimized to move objects whenever possible

■ **Tip** Standard containers & functions typically only use move semantics if the corresponding move members are declared with noexcept. Almost always crucial that all your **move constructors** and **move assignment operators** are declared **noexcept**.

The "Move-and-Swap" Idiom 688

For copy assignment operators, The standard technique, copy-and-swap idiom. Similar pattern for **move assignment operators**:

```
template <typename T> // MOVE ASSIGNMENT OPERATOR
Array<T>& Array<T>::operator=(Array&& rhs) noexcept // Array&& is Rvalue in next line = std::move(rhs)
{  Array<T> moved(std::move(rhs)); // move... (noexcept)
   swap(moved);       // ... and swap (noexcept)
   return *this;  // return lhs
}
```

■ **Tip** If you define them at all, ***always define separate Copy & Move assignment operators***. The latter should be noexcept, the former typically not (copying typically risks triggering a bad_alloc exception, at the least).

-To avoid additional duplication, Use Move-&-Swap on Move assignment operator. Use Copy-&-Swap on Copy-Assignment-Operators.

-Always pass-by-reference-to-const on Copy Assignment Operator, Never pass-by-value, or errors of ambiguous assignments happen.

Special Member Functions 689

There are six **Special Member Functions**, and you now know all of them:

•**Default Constructor** (Chapter 11)	•**Copy Constructor** (Chapter 11)	•**Move Constructor**
•**Destructor** (Chapter 11)	•**Copy Assignment Operator** (Chapter 12)	•**Move Assignment Operator**

"Special" because the compiler automatically generates them under the right circumstances.

Here's a class with just a single data member:

```
class Data
{                  // It's interesting to note what the compiler may provide you with a simple class.
 int value {1};
};
```

WHAT YOU ACTUALLY GET is the following, assuming your compiler conforms to the current language standard:

```
class Data
{
  public:
   Data() : value{1} {} // Default constructor
   Data(const Data& data) : value{data.value} {} // Copy constructor
   Data(Data&& data) noexcept : value{std::move(data.value)} {} // Move constructor
   ~Data() {} // Destructor (implicitly noexcept)
   Data& operator=(const Data& data) // Copy assignment operator
         {
            value = data.value;
             return *this;
         }
   Data& operator=(Data&& data) noexcept // Move assignment operator
         {
            value = std::move(data.value);
             return *this;
         }
  private:
   int value;
};
```

Default Move Members 690

■ **Tip** When any four copy / move members / destructor are defined, the compiler doesn't generate any missing move members.

The Rule of Five 691

■**Rule of Five**-As soon as you declare any of the five special member functions other than the default constructor, you should normally declare all five of them.

The Rule of Zero 692

■ **Rule of Zero**- Avoid having to implement any of the special member functions as much as possible.

TO FOLLOW RULE OF ZERO: Generally just follow various guidelines regarding Containers & Resource Management:

•*All dynamically allocated objects* should be managed by a ***smart pointer*** (Chapter 4).

•*All dynamic arrays* should be managed by a ***std::vector<>*** (Chapter 5).

- Generally, *collections of objects* are to be managed by ***container objects*** such as those provided by Std Library (see also Chapter 19).
- *Any other resources that need cleanup* (network connections, file handles, …) are managed by a ***dedicated RAII object*** (Chapter 16).

Summary 693

Move operations- Since arguments are temporary, the function instead of copying data members, it can steal them.

- **rvalue**: an expression that typically results in a temporary value; **lvalue** is one that results in a more persistent value. Named Var.
- **std::move()** used to convert an lvalues (like variables) into rvalues. ONCE MOVED, objects shouldn't be used anymore.
- An **rvalue reference** type is declared using a **double ampersand, &&**.
- ***Move Constructors*** & ***Move Assignment Operators*** have rvalue reference parameters, called for temporary arguments.
- If a *function copies an input*, ***passing arg by value*** is preferred, it caters both lvalue & rvalue inputs with one single function definition. This Also applies to Class Object
- Automatic variables & function parameters should be *returned by value* and without adding std::move() to the return statement.
- Move members should normally be noexcept; if not, they risk not being invoked by Standard Library containers and other templates.
- **Rule of five-** You either declare all copy members, move members, and the destructor together, ***or none of them at all***.
- **Rule of zero-**Strives to define none at all. The means to achieve rule of zero compliance you actually already know: always manage dynamic memory and other resources using **smart pointers, containers,** & **other RAII techniques**!

■Chapter 18: First-Class Functions 695

First-class functions- if it can treat functions like any other variable.

Pointers to Functions 696

pointer-to-a-function (**function-pointer**)- A variable that can store the address of a function & point to different functions at different times during execution. An address is not sufficient to call a function, though.
-To work properly, a pointer to a function must store each parameter type & return type.

An int pointer can only point at locations containing an int type value.

Defining Pointers to Functions 696

long (*pfun)(long*, int); //This pointer stores the address of functions w/ parameters of type long* & int & return a value of type long:

---

long *pfun(long*, int); // Prototype for a function pfun() that returns a long* value //DIFFERENT

---

return_type (*pointer_name)(list_of_parameter_types); // The general form of a pointer to a function definition:

A Pointer only points at functions with the same **return_type** and **list_of_parameter_types** as those specified in its definition

---

auto pfun = find_maximum; // Using auto to define a pointer
auto* pfun = find_maximum; // Use auto* to highlight the fact that pfun is a pointer

---

pfun = find_minimum; // the same effect as the earlier ones

```
auto* pfun = &find_maximum;        // the same effect as the earlier ones        // ALWAYS add '&' to make it apparent the pointer is changing
pfun = &find_minimum;        // the same effect as the earlier ones        // It's address.. For Readability
```

```
long data[] {23, 34, 22, 56, 87, 12, 57, 76};                    // Call find_minimum() using pfun
std::cout << "Value of minimum is " << pfun(data, std::size(data));  // Use the pointer name as though it were a function name.
```

Callback Functions for Higher-Order Functions 699

Callback function- A function passed to another function as an argument;

higher-order function-The function that accepts another function as an argument.

■ **Note** **First-class callback functions** have a lot of uses beyond serving as the argument to higher-order functions.

--Callbacks are used in day-to-day OOP. Objects often store 1+ callback functions inside their member variables. They could constitute a user-configurable step in the logic implemented by one of the object's member functions, or may be used to signal other objects that some event has occurred. Callback members in various forms are standard OO idioms & patterns, most notably perhaps variations on the classical **Observer pattern**.

Type Aliases for Function Pointers 701

```
template <typename T>// This defines an alias template, which is a template that generates type aliases. In Optimum.h, you could
using Comparison = bool (*)(const T&, const T&); // Use this template to simplify the signature of find_optimum():

template <typename T>
const T* find_optimum(const std::vector<T>& values, Comparison<T> comp);
```

```
Comparison<std::string> string_comp{ longer }; // Of course, you can also use it to define a variable with a concrete type:
```

Function Objects 703

Function Objects (Functors)- Like a function pointer, a function object acts precisely like a function; but unlike a raw function pointer, it's a class type object—complete with its own member variables and possibly even various other member functions.

Basic Function Objects 703

Functors- an Object that can be called like a function.

```
#ifndef LESS_H        // Less.h - A basic class of functor objects
#define LESS_H
class Less
{
public:
 bool operator()(int a, int b) const;
};
#endif                        // LESS_H
```

```
bool Less::operator()(int a, int b) const      // Less.cpp - definition of a basic function call operator
  {      return a < b;   }
```

With this class definition, you can create your very first function object and then call it as if it were an actual function like so:

```
Less less;                        // Create a 'less than' functor...
const bool is_less = less(5, 6);  // ... and 'call' it
std::cout << (is_less? "5 is less than 6" : "Huh?") << std::endl;
```

Standard Function Objects 705

Table 18-1. The Function Object Class Templates Offered by the <functional> Header

Comparisons	**less<>, greater<>, less_equal<>, greater_equal<>, equal_to<>, not_equal_to<>**
Arithmetic operations	**plus<>, minus<>, multiplies<>, divides<>, modulus<>, negate<>**
Logical operations	**logical_and<>, logical_or<>, logical_not<>**
Bitwise operations	**bit_and<>, bit_or<>, bit_xor<>, bit_not<>**

Parameterized Function Objects 706

-Function objects are useful when you add more members— either variables or functions.

```
// Nearer.h
#ifndef NEARER_H    // A class of function objects that compare two values based on how close they are
#define NEARER_H    // to some third value that was provided to the functor at construction time.
#include <cmath>    // For std::abs()
class Nearer
{  public:
        Nearer(int value) : n(value) {}
        bool operator()(int x, int y) const { return std::abs(x - n) < std::abs(y - n); }
   private:
        int n;
};
#endif // NEARER_H
```

```
int main()
{   std::vector<int> numbers{ 91, 18, 92, 22, 13, 43 };
    int number_to_search_for {};
    std::cout << "Please enter a number: ";
    std::cin >> number_to_search_for;
    std::cout << "The number nearest to " << number_to_search_for << " is "
         << *find_optimum(numbers, Nearer{ number_to_search_for }) << std::endl;
}
```

Lambda Expressions 707

lambda expressions- offer convenient, compact syntax that quickly defines callback functions or functors.

--A lot in common with *function definitions*. The most basic form defines a function with no name, an **anonymous function**

--defines a full-blown function object that can carry any number of member variables.

--How it Differs from Regular Functions-- Can access variables existing in the enclosing scope, where it is defined.

Defining a Lambda Expression 708

[] (int x, int y) { return x < y; } // How to define basic ***unnamed*** or ***anonymous functions*** using a lambda expression.

--Don't specify *return type or function name.* Always start with square brackets '**[]**', called **lambda introducer**.

--'**[]**' is followed by the **lambda parameter list** between round parentheses. So Far-- **[](int x, int y)** // x & y in Param List

--Possible to omit *Parameter List*, (), if not needed. **[]() {...}** can be shortened **[] {...}**

--Cannot omit ***Lambda initializer***, []. The lambda initializer is always required to signal the start of a lambda expression.

Naming a Lambda Closure 709

Summary 718

-lambda expressions- have versatile & expressive syntax for defining anonymous Functions & to create lambda closures capable of capturing any number of variables from their surroundings. They are much more powerful than function pointers;

-Unlike function objects, they do not require you to specify a complete class

•Pointer to a function stores the address of a function, which can store the address of any function with the specified return type and number and types of parameters.

•Can use pointer to a function to *call the function* at the address it contains. Can also pass a pointer to a *function as a function arg*.

•**Functors**(Function objects)- objects that behave precisely like a function by overloading the function call operator.

•Any number of member variables / functions can be added to a function object, making them far more versatile than plain function pointers. For one, functors can be parameterized with any number of additional local variables.

•Function objects are powerful but do require quite some coding to set up. **Lambda Expressions-** alleviates the need to define the class for each function object you need.

•**Lambda expression** defines anonymous functions or function objects & typically pass a function as an argument to another function.

•Lambda expressions always begin with a **lambda introducer** '**[]**' that consists of a pair of square brackets that can be empty.

•Lambda [] can contain a **capture**, which specifies which variables in the enclosing scope can be accessed from the body of the lambda expression. Variables can be **captured** by *value* or by *reference*.

•2 Default Capture Clauses: **[=]** specifies all variables are *captured by value*. **[&]** specifies all variables are *captured by reference*.

•*Capture clause* can specify variables to be captured by *value or by reference*.

•Variables captured by value *has a local copy created*. The copy is not modifiable by default.

-Adding **mutable** keyword *after* the parameter list allows local copies of variables captured by value to be modified.

•If return type isn't specified, the compiler deduces the return type from the first return statement in the lambda.

-You can specify the return type for a lambda expression using this **trailing return type syntax**.

•Can use **std::function<>** **template type**, Inside -> **#include <functional>** to specify the type of a function parameter that will accept any first-class function as an argument, including lambda expressions. It allows you to specify a named type for a variable—be it a function parameter, member variable, or automatic variable—that can hold a lambda closure. Which would otherwise be very hard as the name of this type is known only to the compiler.

Parallel Algorithms 756

Summary 756

3 MOST IMPORTANT, Most Frequently used features of the Standard Library: **containers, iterators,** & **algorithms**

Containers- organize data using various data structures.

Typical container(& Sequential Containers) do not offer much functionality beyond **adding, removing,** & **traversing elements**.

Algorithms- Operations to manipulate the data stored inside containers. In the form of generic, **higher-order function templates**.

Regularly consult a Standard Library reference — All developers regularly need guidance from a good reference book or website.

A Reference Should: **1.** Lists all Member Functions of the various containers **2.** The many Algorithm Templates that exist (100+)

3. Specifies Precise Semantics of all this powerful functionality.

- **Sequence containers-** Store Data in a user-determined, linear order, one element after the other.
- GO-TO SEQUENTIAL CONTAINER is **std::vector<>**. Other containers Aren't Practical usually, like: **list<>**, **forward_list<>**, **deque<>**
- 3 CONTAINER ADAPTERS—**std::stack<>**, **queue<>**, **priority_queue<>**— all encapsulate a sequential container to implement a limited set of operations. Allow Adding & Deleting Elements. The difference is the order of the elements coming out.
- **Sets-** *Duplicate-free containers.* Good at determining if they contain a given element.
- **Maps-** *Uniquely associate keys with values*. Allows for quickly retrieving a value given their keys.
- 2 KINDS OF SETS & MAPS: **ordered** & **unordered**. *Ordered-* if you need a sorted view on your data;

  -*Unordered-* For efficiency , except you may have to define an **effective hash function first** (Check Std Lib reference).
- CAN USE ITERATORS to enumerate elements in a container, w/out knowing how the data is organized.
- ITERATORS IN C++ typically make heavy use of operator overloading in order to look and feel like pointers.
- Std Lib has 100+ different algorithms, most in the algorithms header.
- All algorithms operate on half-open ranges of iterators, & many accept a first-class callback function. Mostly you call algorithms w/ lambda expression if its default behavior does not suit you.
- Algorithms that retrieve a single element from a range do so by returning an iterator. (**find()**, **find_if()**, **min_element()**, **max_element()**, …) The **End iterator** of the range is always used to denote "not found."
- ALGORITHMS THAT PRODUCE MULTIPLE OUTPUT ELEMENTS (**copy(), copy_if(),** …) normally are always used in with the **std::back_inserter()**, **front_inserter()**, and **inserter()** utilities in the iterator header.
- USE THE **remove-erase idiom-** To remove multiple elements from sequence containers.
- BY PASSING THE **std::execution::par** EXECUTION POLICY AS THE FIRST ARGUMENT TO MOST ALGORITHMS- To take advantage of the extensive multiprocessing capabilities of current hardware.

Index 761

C++ Command List

Compiling

To compile a C++-program, you can use either g++ or c++
g++ -o executable filename.out sourcefilename.cc
c++ -o executable filename.out sourcefilename.cc
e.g. g++ -o C++sample inout.out C++sample inout.cc
For the following commands you can find at the end of this summary sample programs
Each command in C++ is followed by ";" . Carriage return has no meaning in C++

Comments

/* . . . */ puts the text . . . into comment (more than one line possible)
// . . . puts text . . . for rest of same line into comment

Data Types

Data Type Declaration (Example) Assignment (Example)
integer short i1,i2; i1 = 3;
int i1,i2;
long i1,i2;
unsigned i1,i2;
unsigned long i1,i2;
real float f1,f2; f1 = 3.2; f2 = 2.1E-3;
double f1,f2;
long double f1,f2;
single character char c1,c2; c1 = 'R'
string of characters string s1,s2; s1 = "Farmer"
logical bool b1,b2; b1 = true; b2 = false

Input And Output Input/Output With Screen:

To be able to use the following commands you need to write
#include <iostream> using namespace std;
at the beginning of your program:
output: cout << "string of characters"; cout << variable; << endl; input: cin >> variable;

Input/Output With Files:

To be able to use the following commands you need to write
#include <fstream> at the beginning of your program:
output: ofstream outfilevariable ("outputfilename",ios::out);
outfilevariable << . . . will write into file with name outputfilename
input: ifstream infilevariable ("inputfilename",ios::in);
infilevariable >> . . . will read from file with name outputfilename

Arithmetic Calculations

Operations: + − ∗ /
Functions:
To be able to use all of the following functions you need to to write at the beginning of your program:
#include <cmath>
#include <cstdlib>

pow(x,y) x
y
sin(x)
cos(x)
tan(x)
asin(x) sin−1
(x) in range [−π/2, π/2]
acos(x) cos−1
(x) in range [0, π]
atan(x) tan−1
(x) in range [−π/2, π/2]
sinh(x)
cosh(x)
tanh(x)
exp(x) e
x
log(x) ln(x)
sqrt(x) √
x
fabs(x) |x|
floor(x) largest integer not greater than x; example: floor(5.768) = 5
ceil(x) smallest integer not less than x; example: ceil(5.768) = 6
fmod(x,y) floating-point remainder of x/y with the same sign as x
x % y remainder of x/y, both x and y are integers; example: 7%5=2

Decision Statements

Comparison Operators:

== = example: i1 == i2
!= 6= i1 != i2
> > i1 > i2
< < i1 < i2
>= ≥ i1 >= i2
<= ≤ i1 <= i2
&& and (i1 != i2)&& (i1 == i3)
|| and/or (i1 == i2) || (i1 == i3)

Statements:

if(condition) {
statements
}

---

if(condition) {
statements
}
else {
statements
}

---

if(condition) {
statements
}
else if {
statements

```
}
else {
statements
}
```

```
switch ( casevariable ) {
case value1a:
case value1b:
{statements}
break;
```

```
switch ( casevariable ) {
case value2a:
case value2b:
{statements}
break;
default:
{statements}
}
```

Loops / Repetitions

```
while (conditions) {
statements
}
```

```
for ( init; conditions; update ){
statements
}
```

```
do {
statements these statements are done before the while statement check
} while ( condition) ;
```

functions

A function is a set of commands. It is useful to write a user-defined function, i.e. your own function, whenever you need to do the same task many times in the program. All programs start execution at the function main. For Functions you need steps 1-3:

Step 1: At the beginning of your program you need to declare any functions:
function type function name (types of parameter list);
example 1: double feetinchtometer (int,double);
example 2: void metertofeetinch (double, int&, double&);
The function type declares which variable is passed back from the function (void means none via the function type). The variables without "&" are all input parameters, i.e. only passed to the function and are not changed within the function. The variables with "&" may be passed both to and from the function and may be changed in the function.

Step 2: In the program you use the function with:
function name (actual parameter list);
example 1: feetinchtometer(5.0,3.2);
example 2: metertofeetinch(1.3,feet,inch);

Step 3: After main { . . . } define your function:
function type function name (parameter types and names) { declarations and statements }

example 1: double feetinchtometer(int feet, double inch){ . . .};
example 2: void metertofeetinch (double m, int& feet, double& inch){. . .};

Arrays

Data Type Declaration (Example) Assignment (Example)
array int street[100]; street[0]=0; street[50]=7;
double matrix[7]; matrix[6]=3.2;
etc. etc.
multidim. array int lattice[10][10]; lattice[0][1]=1;
double wateruse[5][3][2]; wateruse[3][1][0]=1.2;
etc. etc.

Escape Sequences 22

Using **character constants** such as a single character or a character string in a program, certain characters are prob

Escape sequence is an indirect way of specifying a Special character, and it always begins with a backslash.

Escape Sequence	Control Character
\n	New Line
\t	New Tab
\v	Vertical Tab
\b	Backspace
\r	Carriage Return
\f	Form Feed
\a	Alert/Bell
	PROBLEM CHARACTI
\\	Backslash
\’	Single Quote
\”	Double Quote

```
std::cout << "\"Least \'said\' \\\n\t\tsoonest \'mended\'.\"" << std::endl;
std::cout << "" << std::endl; ------IS THE MAIN COMMAND
```