1. Introduction to C++

1.1 Programming (general)

program
A computer program consists of instructions executing one at a time.
Input
Input: A program gets data, perhaps from a file, keyboard, touchscreen, network, etc.
Process
Process: A program performs computations on that data, such as adding two values like x + y.
Output
Output: A program puts that data somewhere, such as to a file, screen, network, etc.
variables
Programs use variables to refer to data, like x.
computational thinking
In the information age, many people believe computational thinking, or creating a sequence of instructions to solve a problem, will become increasingly important for work and everyday life.
algorithm
A sequence of instructions that solves a problem is called an algorithm.
Animation
1.1.1 A basic computer program.
Question set
1.1.2 A basic computer program.

Aside
A program is like a recipe
Learning tool
1.1.3 A first programming activity.
Question set
1.1.4 Instructions.
Learning tool
1.1.5 Computational thinking: Creating algorithms to draw shapes using turtle graphics.

1.2 Programming basics

program
A program starts in main(), executing the statements within main's braces { }, one at a time.
semicolon
Each statement typically appears alone on a line and ends with a semicolon, as English sentences end with a period.
Code
Code is the textual representation of a program.
cin
The following statement gets an input value and puts that value into variable x: cin >> x;.
cout
The cout construct supports output; cout is short for *characters out*.
string literal
Text in double quotes " " is known as a string literal.
endl
The statement cout << endl starts a new output line, called a newline.
newline
The statement cout << endl starts a new output line, called a newline.

newline character
A new output line can also be produced by inserting \n, known as a newline character, within a string literal.
Animation
1.2.1 Program execution begins with main, then proceeds one statement at a time.
Question set
1.2.2 A first program.
Animation
1.2.3 A program can get an input value from the keyboard.
Question set
1.2.4 Basic input.
Question set
1.2.5 Basic input.

Figure
1.2.1 Outputting text and newlines.
Question set
1.2.6 Basic text output.
Question set
1.2.7 Basic text output.

Figure
1.2.2 Outputting a variable's value.
Question set
1.2.8 Basic variable output.
Question set
1.2.9 Basic variable output.

Figure
1.2.3 Outputting multiple items using one output statement.
Question set
1.2.10 Basic output.
Learning tool
1.2.11 Output simulator.
Progression
1.2.1 Enter the output.
Progression
1.2.2 Output basics.
Coding challenge
1.2.3 Read multiple user inputs.

Aside
Newline character

1.3 Comments and whitespace

comment
A comment is text a programmer adds to code, to be read by humans to better understand the code but ignored by the compiler.
single-line comment
A single-line comment starts with // and includes all the following text on that line.
multi-line comment

A multi-line comment starts with /* and ends with */, where all text between /* and */ is part of the comment.
block comment
A multi-line comment is also known as a block comment.
Whitespace
Whitespace refers to blank spaces (space and tab characters) between items within a statement and blank lines between statements (called newlines).

Figure
1.3.1 Comments example.
Question set
1.3.1 Comments.

Figure
1.3.2 Good use of whitespace.

Figure
1.3.3 Bad use of whitespace.
Question set
1.3.2 Whitespace.
Animation
1.3.3 A compiler scans code line by line, left to right; whitespace is mostly irrelevant.
Question set
1.3.4 Compiling code with whitespace and comments.

1.4 Errors and warnings

syntax error
One kind of mistake, known as a syntax error, is to violate a programming language's rules on how symbols can be combined to create a program.
compile-time error
Because a syntax error is detected by the compiler, a syntax error is known as a type of compile-time error.
logic error
A logic error, also called a bug, is an error that occurs while a program runs.
bug
A logic error, also called a bug, is an error that occurs while a program runs.
warning
A compiler will sometimes report a warning, which doesn't stop the compiler from creating an executable program but indicates a possible logic error.

Figure
1.4.1 Compiler reporting a syntax error.
Question set
1.4.1 Syntax errors.

Figure
1.4.2 Misleading compiler error message.
Animation
1.4.2 The compiler error message's line may be past the line with the actual error.
Question set
1.4.3 Unclear error messages.

Figure
1.4.3 Good practice for fixing errors reported by the compiler.
Question set
1.4.4 Fixing the first error.
Coding challenge
1.4.1 Basic syntax errors.
Coding challenge
1.4.2 More syntax errors.

Figure
1.4.4 Logic errors.

Aside
Bugs
Animation
1.4.5 Compile and run after writing just a few statements.
Question set
1.4.6 Compiling and running frequently.
Question set
1.4.7 Compiler warnings.

1.5 Computers and programs (general)

bits
0s and 1s are known as bits (*b*inary dig*its*).
processors
To support different calculations, circuits called processors were created to process (aka *execute*) a list of desired calculations.
instruction
memory
A memory is a circuit that can store 0s and 1s in each of a series of thousands of addressed locations.
program
application
app
machine instructions
Instructions represented as 0s and 1s are known as machine instructions.
executable program
A sequence of machine instructions together form an executable program.
assembly
high-level languages
compilers
To support high-level languages, programmers created compilers, which are programs that automatically translate high-level language programs into executable programs.

Figure
1.5.1 Looking under the hood of a car.

Figure
1.5.2 Early computer made from thousands of switches.

Figure
1.5.3 As switches shrunk, so did computers. The computer processor chip on the right has millions of switches.

Figure
1.5.4 Memory.
Animation
1.5.1 Computer processor and memory.

Table
1.5.1 Sample processor instructions.
Animation
1.5.2 Memory stores instructions and data as 0s and 1s.
Animation
1.5.3 Processor executing instructions.
Question set
1.5.4 Computer basics.
Animation
1.5.5 Program compilation and execution.
Question set
1.5.6 Programs.

1.6 Computer tour

Input/output devices
screen
A screen (or monitor) displays items to a user.
keyboard
A keyboard allows a user to provide input to the computer.
Storage
disk
A disk (aka *hard drive*) stores files and other data, such as program files, song/movie files, or office documents.
Memory
RAM
RAM (random-access memory) temporarily holds data read from storage and is designed such that any address can be accessed much faster than disk, in just a few clock ticks (see below) rather than hundreds of ticks.
byte
A byte is 8 bits.
Processor
processor
The processor runs the computer's programs, reading and executing instructions from memory, performing operations, and reading/writing data from/to memory.
operating system
The operating system allows a user to run other programs and interfaces with the many other peripherals.
cache
A processor may contain a small amount of RAM on its own chip, called cache memory, accessible in one clock tick rather than several, for maintaining a copy of the most-used instructions/data.
Clock
clock
A processor's instructions execute at a rate governed by the processor's clock, which ticks at a specific frequency.
transistors
Engineers created smaller switches called transistors, which in 1958 were integrated onto a single chip.

integrated circuit
Moore's Law
Moore's Law: the doubling of IC capacity roughly every 18 months, .
Animation
1.6.1 Some computer components.
Question set
1.6.2 Programs.

1.7 Language history

C
In 1978, Brian Kernighan and Dennis Ritchie at AT&T Bell Labs (which used computers extensively for automatic phone call routing) published a book describing a new high-level language with the simple name C.
C++
In 1985, Bjarne Stroustrup published a book describing a C-based language called C++, adding constructs to support a style of programming known as *object-oriented programming*, along with other improvements.

Table
1.7.1 Top languages ranked by popularity.
Question set
1.7.1 C/C++ history.

1.8 Problem solving

problem solving
Programming is largely about problem solving: creating a methodical solution to a given task.

Example
1.8.1 Solving a (nonprogramming) problem: Matching socks.
Question set
1.8.1 Matching socks solution approach.
Question set
1.8.2 Greeting people problem.

Example
1.8.2 Example: Sorting name tags.
Question set
1.8.3 Sorting name tags.

1.9 Why programming

computational thinking
The thought processes needed to build correct, precise, logical programs is sometimes called computational thinking and has benefits beyond programming.

Table
1.9.1 Best jobs of 2019, per U.S. News and World Report.
Question set
1.9.1 Computing jobs are often ranked among the best jobs.

Table
1.9.2 Computing jobs.
Question set
1.9.2 Computing jobs.
Question set
1.9.3 Programming in non-computing jobs.
Animation
1.9.4 Learning programming tends to aid in precise, logical thought, aspects of computational thinking.
Question set
1.9.5 Computational thinking.

Aside
Even the best programmers make mistakes
Question set
1.9.6 Programming.

1.10 Why whitespace matters

whitespace
Whitespace is any blank space or newline.
Animation
1.10.1 Precisely formatting a meeting invite.
Question set
1.10.2 Program correctness includes correctly-formatted output.
Question set
1.10.3 Thinking precisely, and attention to detail.

Aside
Programmer attention to details

1.13 zyLab training: Basics

Lab activity
1.13.1 zyLab training: Basics

1.14 zyLab training: Interleaved input / output

Lab activity
1.14.1 zyLab training: Interleaved input / output

2. Variables / Assignments

2.1 Variables and assignments (general)

variable
In a program, a variable is a named item, such as x or numPeople, used to hold a value.
assignment
An assignment assigns a variable with a value, such as x = 5.
incrementing
Increasing a variable's value by 1, as in x = x + 1, is common, and known as incrementing the variable.
Learning tool

2.1.1 People on bus.
Animation
2.1.2 Variables and assignments.

Aside
= is not equals
Question set
2.1.3 Valid assignments.
Question set
2.1.4 Variables and assignments.
Learning tool
2.1.5 Trace the variable value.
Animation
2.1.6 A variable may appear on the left and right of an assignment.
Question set
2.1.7 Variable on both sides.

2.2 Variables (int)

variable declaration
A variable declaration is a statement that declares a new variable, specifying the variable's name and type.
assignment statement
An assignment statement assigns the variable on the left-side of the = with the current value of the right-side expression.
expression
An expression may be a number like 80, a variable name like numApples, or a simple calculation like numApples + 1.
integer literal
An integer like 80 appearing in an expression is known as an integer literal.
Animation
2.2.1 A variable refers to a memory location.
Question set
2.2.2 Declaring integer variables.

Aside
Compiler optimization

Figure
2.2.1 Assigning a variable.
Question set
2.2.3 Assignment statements.
Progression
2.2.1 Enter the output of the variable assignments.
Coding challenge
2.2.2 Assigning a sum.

Figure
2.2.2 Variable initialization: Example program.
Question set
2.2.4 Declaring and initializing integer variables.
Coding challenge

2.2.3 Declaring and initializing variables.
Animation
2.2.5 Variable assignments overwrite a variable's previous values: People-known example.

Aside
Six degrees of separation
Question set
2.2.6 Assignment statements with same variable on both sides.
Coding challenge
2.2.4 Adding a number to a variable.
Question set
2.2.7 Common errors.

2.3 Identifiers

identifier
A name created by a programmer for an item like a variable or function is called an identifier.
case sensitive
Identifiers are case sensitive, meaning upper and lower case letters differ.
reserved word
A reserved word is a word that is part of the language, like int, short, or double.
keyword
A reserved word is also known as a keyword.
Lower camel case
Lower camel case abuts multiple words, capitalizing each word except the first, as in numApples or peopleOnBus.
Question set
2.3.1 Valid identifiers.
Question set
2.3.2 Meaningful identifiers.

Aside
zyBook's naming conventions

Table
2.3.1 C++ reserved words / keywords.

2.4 Arithmetic expressions (general)

expression
An expression is any individual item or combination of items, like variables, literals, operators, and parentheses, that evaluates to a value, like 2 * (x + 1).
literal
A literal is a specific value in code like 2.
operator
An operator is a symbol that performs a built-in calculation, like +, which performs addition.
addition
The addition operator is +, as in x + y.
+
The addition operator is +, as in x + y.
subtraction
The subtraction operator is -, as in x - y.
-

The subtraction operator is -, as in x - y.
negation
The - operator is for negation, as in -x + y, or x + -y.
multiplication
The multiplication operator is *, as in x * y.
*
The multiplication operator is *, as in x * y.
division
The division operator is /, as in x / y.
/
The division operator is /, as in x / y.
evaluates
An expression evaluates to a value, which replaces the expression. Ex: If x is 5, then x + 1 evaluates to 6, and y = x + 1 assigns y with 6.
precedence rules
An expression is evaluated using the order of standard mathematics, such order known in programming as precedence rules.

Table
2.4.1 Arithmetic operators.
Question set
2.4.1 Expressions.
Question set
2.4.2 Capturing behavior with an expression.

Table
2.4.2 Precedence rules for arithmetic operators.
Animation
2.4.3 Evaluating expressions.
Question set
2.4.4 Evaluating expressions and precedence rules.

Aside
Using parentheses to make the order of evaluation explicit
Question set
2.4.5 Converting a formatted expression to a program expression.

2.5 Arithmetic expressions (int)

unary minus
Minus (-) used as negative is known as unary minus.
compound operators
Special operators called compound operators provide a shorthand way to update a variable, such as userAge += 1 being shorthand for userAge = userAge + 1. Other compound operators include -=, *=, /=, and %=.
+=
Special operators called compound operators provide a shorthand way to update a variable, such as userAge += 1 being shorthand for userAge = userAge + 1. Other compound operators include -=, *=, /=, and %=.
-=
Special operators called compound operators provide a shorthand way to update a variable, such as userAge += 1 being shorthand for userAge = userAge + 1. Other compound operators include -=, *=, /=, and %=.
*=

Special operators called compound operators provide a shorthand way to update a variable, such as userAge += 1 being shorthand for userAge = userAge + 1. Other compound operators include -=, *=, /=, and %=.
/=
Special operators called compound operators provide a shorthand way to update a variable, such as userAge += 1 being shorthand for userAge = userAge + 1. Other compound operators include -=, *=, /=, and %=.
%=
Special operators called compound operators provide a shorthand way to update a variable, such as userAge += 1 being shorthand for userAge = userAge + 1. Other compound operators include -=, *=, /=, and %=.

Figure
2.5.1 Expressions examples: Leasing cost.
Question set
2.5.1 Simple program with an arithmetic expression.
Question set
2.5.2 Single space around operators.
Question set
2.5.3 Compound operators.
Question set
2.5.4 Expression in statements.
Progression
2.5.1 Enter the output of the integer expressions.
Coding challenge
2.5.2 Compute an expression.
Coding challenge
2.5.3 Total cost.

2.6 Example: Health data

Incremental development
Incremental development is the process of writing, compiling, and testing a small amount of code, then writing, compiling, and testing a small amount more (an incremental amount), and so on.

Figure
2.6.1 Health data: Calculating user's age in days.
Question set
2.6.1 Calculating user age in days.

Figure
2.6.2 Health data: Calculating user's age in days and minutes.
Question set
2.6.2 Calculating user age in days.

Figure
2.6.3 Health data: Calculating total heartbeats lifetime.
Question set
2.6.3 Calculating user's heartbeats.

Aside
Limits on int values

2.7 Floating-point numbers (double)

floating-point number
A floating-point number is a real number containing a decimal point that can appear anywhere (or "float") in the number. Ex: 98.6, 0.0001, or -55.667.
double
A double variable stores a floating-point number.
floating-point literal
A floating-point literal is a number with a fractional part, even if the fraction is 0, as in 1.0, 0.0, or 99.573.
infinity
Dividing a nonzero floating-point number by zero results in infinity or -infinity, depending on the signs of the operands.
-infinity
Dividing a nonzero floating-point number by zero results in infinity or -infinity, depending on the signs of the operands.
Not a number
Not a number (NaN) indicates an unrepresentable or undefined value.
NaN
Not a number (NaN) indicates an unrepresentable or undefined value.

Figure
2.7.1 Variables of type double: Travel time example.
Question set
2.7.1 Declaring and assigning double variables.
Question set
2.7.2 Floating-point literals.

Aside
Scientific notation

Aside
Floating-point for money
Question set
2.7.3 Floating-point versus integer.

Figure
2.7.2 Floating-point division by zero example.
Question set
2.7.4 Floating-point division.
Animation
2.7.5 Reducing the output of Pi.
Question set
2.7.6 Reducing floating-point output.
Coding challenge
2.7.1 Sphere volume.

2.8 Scientific notation for floating-point literals

scientific notation
A floating-point literal using scientific notation is written using an e preceding the power-of-10 exponent, as in 6.02e23 to represent 6.02 x 10^{23}.

Figure

2.8.1 Calculating atoms of gold.
Question set
2.8.1 Scientific notation.
Coding challenge
2.8.1 Acceleration of gravity.

2.9 Constant variables

constant variable
An initialized variable whose value cannot change is called a constant variable.

Figure
2.9.1 Constant variable example: Lightning distance.
Question set
2.9.1 Constant variables.
Coding challenge
2.9.1 Using constants in expressions.

2.10 The #define directive

#define
The #define directive, of the form #define MACROIDENTIFIER replacement, instructs the processor to replace any occurrence of MACROIDENTIFIER in the subsequent program code by the replacement text.
macro
#define is sometimes called a macro.
#undef
#ifdef
#if
#else
#elif
#pragma
#line
#error

Construct
2.10.1 #define directive.
Question set
2.10.1 #define.

2.11 Using math functions

math library
A standard math library has about 20 math operations, known as functions.
function
A function is a list of statements executed by invoking the function's name, such invoking known as a function call.
function call
A function is a list of statements executed by invoking the function's name, such invoking known as a function call.
arguments
Any function input values, or arguments, appear within ( ), separated by commas if more than one.
Animation
2.11.1 Using a math function.

Table
2.11.1 A few common math functions from the math library.
Question set
2.11.2 Math functions.

Figure
2.11.1 Math function example: Mass growth.
Question set
2.11.3 Growth rate.
Question set
2.11.4 Calculate Pythagorean theorem using math functions.
Animation
2.11.5 Function call in an argument.
Question set
2.11.6 Function calls in arguments.

Aside
cmath and cstdlib
Progression
2.11.1 Math functions.
Progression
2.11.2 Writing math calculations.
Coding challenge
2.11.3 Using math functions to calculate the distance between two points.

2.12 Integer division and modulo

divide-by-zero error
A divide-by-zero error occurs at runtime if a divisor is 0, causing a program to terminate.
runtime error
A divide-by-zero error is an example of a runtime error, a severe error that occurs at runtime and causes a program to terminate early.
modulo operator
The modulo operator (%) evaluates the remainder of the division of two integer operands. Ex: 23 % 10 is 3.
%
The modulo operator (%) evaluates the remainder of the division of two integer operands. Ex: 23 % 10 is 3.
Animation
2.12.1 Integer division does not generate any fraction.
Question set
2.12.2 Integer division modulo.

Figure
2.12.1 Divide-by-zero example: Compute salary per day.
Question set
2.12.3 More integer division.

Figure
2.12.2 Division and modulo example: Minutes to hours/minutes.
Question set
2.12.4 Modulo.

Question set
2.12.5 Integer division and modulo.

Aside
Why parentheses matter
Progression
2.12.1 Enter the output of the integer expressions.

Example
2.12.1 Random number in range.

Example
2.12.2 Getting digits.

Example
2.12.3 Get prefix of a phone number.
Question set
2.12.6 Modulo examples.
Coding challenge
2.12.2 Compute change.

2.13 Type conversions

type conversion
A type conversion is a conversion of one data type to another, such as an int to a double.
implicit conversion
The compiler automatically performs several common conversions between int and double types, such automatic conversion known as implicit conversion.
type cast
A type cast explicitly converts a value of one type to another type.
static_cast
The static_cast operator (`static_cast<type>(expression)`) converts the expression's value to the indicated type.
Animation
2.13.1 Implicit type conversion: int-to-double.
Question set
2.13.2 Implicit conversions among double and int.
Question set
2.13.3 Implicit conversions among double and int with variables.

Aside
Assigning doubles with integer literals

Figure
2.13.1 Using type casting to obtain floating-point division.
Question set
2.13.4 Type casting.

Figure
2.13.2 Common error: Forgetting cast results in integer division.

Animation
2.13.5 Common error: Casting final result instead of operands.
Question set
2.13.6 Type casting.
Progression
2.13.1 Type conversions.
Coding challenge
2.13.2 Type casting: Computing average kids per family.

2.14 Binary

binary number
Because each memory location is composed of bits (0s and 1s), a processor stores a number using base 2, known as a binary number.
decimal number
For a number in the more familiar base 10, known as a decimal number, each digit must be 0-9 and each digit's place is weighed by increasing powers of 10.
base 2
In base 2, each digit must be 0-1 and each digit's place is weighed by increasing powers of 2.

Table
2.14.1 Decimal numbers use weighed powers of 10.

Table
2.14.2 Binary numbers use weighed powers of 2.
Question set
2.14.1 Binary numbers.

2.15 Characters

char
A variable of char type, as in `char myChar;`, can store a single character like the letter m.
character literal
A character literal is surrounded with single quotes, as in `myChar = 'm';`.
ASCII
ASCII is an early standard for encoding characters as numbers.
escape sequence
Escape sequence: A two-character sequence starting with \ that represents a special character.

Figure
2.15.1 Simple char example: Arrow.
Question set
2.15.1 char data type.

Figure
2.15.2 Getting a character from input.
Animation
2.15.2 A char variable stores a number.

Table

2.15.1 Character encodings as numbers in the ASCII standard.
Question set
2.15.3 Character encodings.

Table
2.15.2 Common escape sequences.
Question set
2.15.4 Escape sequences.
Coding challenge
2.15.1 Printing a message with ints and chars.
Coding challenge
2.15.2 Outputting all combinations.

2.16 Strings

string
A string is a sequence of characters.
string literal
A string literal surrounds a character sequence with double quotes, as in "Hello", "52 Main St.", or "42".
whitespace character
A whitespace character is a character used to represent horizontal and vertical spaces in text, and includes spaces, tabs, and newline characters.
getline
The function getline(cin, stringVar) gets all remaining text on the current input line, up to the next newline character (which is removed from input but not put in stringVar).
Learning tool
2.16.1 A string is stored as a sequence of characters in memory.
Question set
2.16.2 String literals.

Figure
2.16.1 Declaring and assigning a string.
Question set
2.16.3 Declaring and assigning a string variable.
Question set
2.16.4 Getting a string without whitespace from input.
Question set
2.16.5 Getting a string without whitespace from input (continued).

Figure
2.16.2 Strings example: Word game.
Question set
2.16.6 Getting a string with whitespace from input.

Figure
2.16.3 Reading an input string containing spaces using getline.
Animation
2.16.7 Combining cin and getline() can be tricky.
Question set
2.16.8 Getting strings without and with whitespace.
Coding challenge

2.16.1 Reading and outputting strings.

2.17 Integer overflow

overflow
An overflow occurs when the value being assigned to a variable is greater than the maximum value the variable can store.
compiler warning
The compiler may not report a syntax error (the syntax is correct), but may output a compiler warning message that indicates a potential problem.
Animation
2.17.1 Overflow error.
Question set
2.17.2 Overflow.

2.18 Numeric data types

long long
Long long is used for integers expected to exceed about 2 billion.
overflow
An overflow occurs when the value being assigned to a variable is greater than the maximum value the variable can store.

Table
2.18.1 Integer numeric data types.
Question set
2.18.1 Integer types.

Table
2.18.2 Floating-point numeric data types.
Question set
2.18.2 Floating-point numeric types.

2.19 Unsigned

Table
2.19.1 Unsigned integer data types.

Figure
2.19.1 Unsigned variables example: Memory size converter.
Question set
2.19.1 Unsigned variables.

2.20 Random numbers

rand()
The rand() function, in the C standard library, returns a random integer each time the function is called, in the range 0 to RAND_MAX.
seed

For the first call to rand(), no previous random integer exists, so the function uses a built-in integer known as the seed.

time()

The function time() returns the number of seconds since Jan 1, 1970.

Figure
2.20.1 Outputting three random integers.

Animation
2.20.1 Restricting random integers to a specific number of possible values.

Question set
2.20.2 Random number basics.

Animation
2.20.3 Generating random integers in a specific range not starting from 0.

Question set
2.20.4 Generating random integers in a specific range.

Question set
2.20.5 Specific range.

Figure
2.20.2 Randomly moving a student from one seat to another.

Question set
2.20.6 Random integer example: Moving seats.

Figure
2.20.3 Using a unique seed for each program run.

Question set
2.20.7 Using a unique seed for each program run.

Progression
2.20.1 Generate a random integer.

Coding challenge
2.20.2 rand function: Seed and then get random numbers.

Coding challenge
2.20.3 Fixed range of random numbers.

2.21 Debugging

Debugging

Debugging is the process of determining and fixing the cause of a problem in a computer program.

Troubleshooting

Troubleshooting is another word for debugging.

Figure
2.21.1 A methodical debugging process.

Figure
2.21.2 Circle area program: Problem detected.

Figure
2.21.3 Circle area program: Predict problem is bad output.

Figure
2.21.4 Circle area program: Predict problem is bad area computation.

Figure
2.21.5 Circle area program: Predict problem is bad radius computation.
Question set
2.21.1 Debugging.

2.22 Auto (since C++11)

auto
In a variable declaration, using auto as the type specifier causes the compiler to automatically deduce the type from the initializer.
Question set
2.22.1 Auto in variable declarations.

2.23 Style guidelines

style guidelines
Each programming team, whether a company, open source project, or a classroom, may have style guidelines for writing code.
K&R style
K&R style for braces and indents is named after C language creators Kernighan and Ritchie.
Stroustrup style
Stroustrup style for braces and indents is named after C++ language creator Bjarne Stroustrup.

Table
2.23.1 Sample style guide.

2.24 LAB: Divide by x

Lab activity
2.24.1 LAB: Divide by x

2.25 LAB: Expression for calories burned during workout

Lab activity
2.25.1 LAB: Expression for calories burned during workout

2.26 LAB: Using math functions

Lab activity
2.26.1 LAB: Using math functions

3. Branches

3.1 If-else branches (general)

branch
A branch is a program path taken only if an expression's value is true.
If

If branch: A branch taken only if an expression is true.
if-else
An if-else structure has two branches: The first branch is taken if an expression is true, else the other branch is taken.
Animation
3.1.1 Branching concept.
Question set
3.1.2 Branch concept.
Animation
3.1.3 A simple branch: Hotel discount.
Question set
3.1.4 Branches.
Animation
3.1.5 Example if branch: Computing absolute value.
Question set
3.1.6 Example if branch: Absolute value.
Animation
3.1.7 If-else branches.
Question set
3.1.8 If-else branches.
Animation
3.1.9 If-else example: Max.
Question set
3.1.10 If-else example: Max.
Animation
3.1.11 If-elseif-else branch.
Question set
3.1.12 If-elseif-else.

3.2 If-else

if
An if statement executes a group of statements if an expression is true.
Braces
Braces {_}, sometimes redundantly called curly braces, represent a grouping, such as a grouping of statements.
if-else
An if-else statement executes one group of statements when an expression is true, and another group of statements when the expression is false.
equality operator
The equality operator == evaluates to true if the left side and right side are equal.
==
The equality operator == evaluates to true if the left side and right side are equal.
Animation
3.2.1 if statement: Hotel discount.
Question set
3.2.2 If statement.

Construct
3.2.1 If-else statement.
Animation
3.2.3 if-else statement: Car insurance.

Aside

Car insurance prices
Question set
3.2.4 If-else statements.
Question set
3.2.5 Writing an if-else statement.
Progression
3.2.1 Enter the output for the if-else branches.
Progression
3.2.2 Basic if-else expression.
Progression
3.2.3 Basic if-else.

Construct
3.2.2 Multi-branch if-else statement. Only 1 branch will execute.

Figure
3.2.1 Multi-branch if-else example: Anniversaries.
Question set
3.2.6 Multi-branch if-else statements.
Animation
3.2.7 Common error when omitting braces.
Question set
3.2.8 Braces are important.
Coding challenge
3.2.4 If-else statement: Fix errors.

3.3 More if-else

nested if-else
A branch's statements can include any valid statements, including another if-else statement, which are known as nested if-else statements.

Figure
3.3.1 Nested if-else.
Question set
3.3.1 Nested if-else statements.

Figure
3.3.2 Multiple distinct if statements.
Question set
3.3.2 If statements.
Progression
3.3.1 Enter the output for the multiple if-else branches.
Progression
3.3.2 If-else statements.

3.4 Equality and relational operators

equality operator
An equality operator checks whether two operands' values are the same (==) or different (!=).
Boolean

A Boolean is a type that has just two values: true or false.
==
A == b means a is equal to b.
!=
A != b means a is not equal to b.
relational operator
A relational operator checks how one operand's value relates to another, like being greater than.
<
A < b means a is less than b.
>
A > b means a is greater than b.
<=
A <= b means a is less than or equal to b.
>=
A >= b means a is greater than or equal to b.

Table
3.4.1 Equality operators.
Question set
3.4.1 Evaluating expressions that have equality operators.
Question set
3.4.2 Creating expressions with equality operators.

Table
3.4.2 Relational operators.
Question set
3.4.3 Evaluating equations having relational operators.
Question set
3.4.4 Creating expressions with relational operators.
Progression
3.4.1 Enter the output for the branches with relational and equality operators.
Progression
3.4.2 Equality and relational expressions.
Question set
3.4.5 Comparing various types.
Question set
3.4.6 Watch out for assignment in an if-else expression.
Progression
3.4.3 If-else statement: Fix errors.
Coding challenge
3.4.4 If-else statement: Print senior citizen.

3.5 Detecting ranges (general)

Animation
3.5.1 An if-elseif-else structure can elegantly detect ranges.
Question set
3.5.2 Using if-elseif-else to detect increasing ranges.
Question set
3.5.3 More ranges with if-elseif-else.

3.6 Detecting ranges with if-else statements

Figure
3.6.1 Using sequential nature of multi-branch if-else for ranges: Insurance prices.
Question set
3.6.1 Ranges and multi-branch if-else.
Question set
3.6.2 Complete the multi-branch if-else.
Progression
3.6.1 Detect ranges using branches.
Coding challenge
3.6.2 Multi-branch if-else statement: Print century.

3.7 Logical operators

logical operator
A logical operator treats operands as being true or false, and evaluates to true or false. Logical operators include AND, OR, and NOT.
Logical AND
Logical AND: true when both of its operands are true .
Logical OR
Logical OR: true when at least one of its two operands are true .
Logical NOT
Logical NOT: true when its one operand is false, and vice-versa.
Logical AND
Logical AND (&&): true when both of its operands are true .
Logical OR
Logical OR (||): true when at least one of its two operands are true .
Logical NOT
Logical NOT (!): true when its one operand is false, and vice-versa.
Animation
3.7.1 Logical operators: AND, OR, and NOT.

Table
3.7.1 Logical operators.
Question set
3.7.2 Evaluating expressions with logical operators.
Animation
3.7.3 Using AND to detect if a value is within a range.
Question set
3.7.4 Using AND to detect if a value is within a range.

Table
3.7.2 Logical operators.
Question set
3.7.5 Logical operators.
Question set
3.7.6 Evaluating expressions with logical operators.
Question set
3.7.7 Logical operators: Complete the expressions to detect the desired range.
Question set
3.7.8 Creating expressions with logical operators.
Learning tool

3.7.9 Logical expression simulator.

Figure
3.7.1 Detecting ranges: Cable TV channels.
Question set
3.7.10 TV channel example: Detecting ranges.
Animation
3.7.11 Detecting ranges implicitly vs. explicitly.
Question set
3.7.12 Detecting ranges implicitly vs. explicitly.
Progression
3.7.1 Enter the output of the Boolean expressions.
Coding challenge
3.7.2 Detect specific values.
Coding challenge
3.7.3 Detect number range.

3.8 Example: Toll calculation

Table
3.8.1 Weekday toll schedule.

Figure
3.8.1 Calculating toll based on time of day.
Question set
3.8.1 Toll calculation.

Table
3.8.2 Toll schedule for weekends and holidays.

Figure
3.8.2 Calculating toll based on time of day and day of week.
Question set
3.8.2 If-else statements for calculating toll amount and formatting time.

Figure
3.8.3 Calculating toll with carpool discount.
Question set
3.8.3 Toll calculation.

3.9 Order of evaluation

precedence rules
The order in which operators are evaluated in an expression are known as precedence rules.
bitwise operators
& and | represent bitwise operators, which perform AND or OR on corresponding individual bits of the operands.

Table

3.9.1 Precedence rules for arithmetic, logical, and relational operators.
Animation
3.9.1 Applying the precedence rules to an expression can be thought of as a 'tree'.
Question set
3.9.2 Order of evaluation.
Question set
3.9.3 Common errors in expressions.
Question set
3.9.4 Order of evaluation.
Question set
3.9.5 Expression for detecting a range.
Question set
3.9.6 Bitwise vs. logical operators.

3.10 Switch statements

switch
A switch statement can more clearly represent multi-branch behavior involving a variable being compared to constant values.
case
default case
break
Animation
3.10.1 Switch statement.
Question set
3.10.2 Switch statement.

Aside
Multi-branch if-else statement

Construct
3.10.1 Switch statement general form.

Figure
3.10.1 Switch example: Estimates a dog's age in human years.

Figure
3.10.2 Switch example: Dog years with months.
Question set
3.10.3 Switch statement.
Coding challenge
3.10.1 Rock-paper-scissors.
Coding challenge
3.10.2 Switch statement to convert letters to Greek letters.

3.11 Boolean data type

Boolean
Boolean refers to a quantity that has only two possible values, true or false.
bool
The language has the built-in data type bool for representing Boolean quantities.

Figure
3.11.1 Variables of bool data type: Life expectancy calculator.
Question set
3.11.1 Boolean variables.

Figure
3.11.2 Using Boolean variables to simplify expressions.
Question set
3.11.2 Simplifying expressions.
Coding challenge
3.11.1 Using bool.
Coding challenge
3.11.2 Bool in branching statements.

3.12 String comparisons

Question set
3.12.1 Equal strings.

Figure
3.12.1 String equality example: Censoring.
Question set
3.12.2 Comparing strings for equality.
Animation
3.12.3 String comparison.
Question set
3.12.4 Case matters in string comparisons.
Question set
3.12.5 Relational string comparison.
Coding challenge
3.12.1 String comparison: Detect word.
Coding challenge
3.12.2 Print two strings in alphabetical order.

3.13 String access operations

index
Each string character has a position number called an index, starting with 0.
at()
At(): The notation someString.at(x) accesses the character at index x of a string.
size()
The function s1.size() returns s1's length. Ex: If s1 is "Hey", s1.size() returns 3.
append
The function s1.append(s2) appends string s2 to string s1. Ex: If s1 is "Hey", s1.append("!!!") makes s1 "Hey!!!".
exception
An exception is a detected runtime error that commonly prints an error message and terminates the program.
Animation
3.13.1 A string's characters each has an index, starting with 0.
Question set
3.13.2 String indices.

Figure
3.13.1 String character access: Word scramble.
Question set
3.13.3 Accessing string characters.

Figure
3.13.2 Example: Changing a character.
Question set
3.13.4 Assigning a string character.

Figure
3.13.3 Example: Adding a period to a caption if no punctuation.

Aside
size() and length()
Question set
3.13.5 Working with the end of a string.
Question set
3.13.6 String length.
Question set
3.13.7 Working with the end of a string.
Animation
3.13.8 Common error: Out-of-range access yields an exception.
Question set
3.13.9 Out-of-range string access.
Coding challenge
3.13.1 String library functions.
Coding challenge
3.13.2 Looking for characters.

3.14 Character operations

cctype library
Including the cctype library via `#include <cctype>` provides access to several functions for working with characters.
isalpha
toupper
isdigit
tolower
isspace

Table
3.14.1 Character functions return values.

Figure
3.14.1 State abbreviation capitalization.
Question set
3.14.1 Character functions.
Coding challenge

3.14.1 String with digit.
Coding challenge
3.14.2 Alphabetic replace.

3.15 More string operations

find
Find(item) returns index of first item occurrence, else returns string::npos (a constant defined in the string library).
substr
Substr(index, length) returns substring starting at index and having length characters.
push_back
Push_back(c) appends character c to the end of a string.
insert
Insert(indx, subStr) Inserts string subStr starting at index indx.
replace
Replace(indx, num, subStr) replaces characters at indices indx to indx+num-1 with a copy of subStr.

Table
3.15.1 find() and substr() functions, invoked as myString.find().

Figure
3.15.1 Example: Get username from email address.
Question set
3.15.1 find() and substr().

Table
3.15.2 String modify functions, invoked as myString.push_back(c). Each increases/decreases string's length appropriately.

Figure
3.15.2 String modify example: Greeting.
Question set
3.15.2 String modification functions.
Coding challenge
3.15.1 Combining strings.
Coding challenge
3.15.2 Name song.
Coding challenge
3.15.3 Using find().

3.16 Conditional expressions

conditional expression
A conditional expression has the form `condition ? exprWhenTrue : exprWhenFalse`.
ternary operator
Animation
3.16.1 Conditional expression.
Question set
3.16.2 Conditional expressions.
Coding challenge
3.16.1 Conditional expression: Print negative or positive.

Coding challenge
3.16.2 Conditional assignment.
Progression
3.16.3 Conditional expressions: Enter the output of the code.

3.17 Floating-point comparison

epsilon
The difference threshold indicating that floating-point numbers are equal is often called the epsilon.
Animation
3.17.1 Floating-point comparisons.
Question set
3.17.2 Using == with floating-point numbers.
Question set
3.17.3 Floating-point comparisons.
Question set
3.17.4 Floating point statements.

Figure
3.17.1 Example of comparing floating-point numbers for equality: Body temperature.
Question set
3.17.5 Body temperature in Fahrenheit.

Figure
3.17.2 Observing the inexact values stored in floating-point variables.
Question set
3.17.6 Representing floating-point numbers.
Coding challenge
3.17.1 Floating-point comparison: Print Equal or Not equal.

3.18 Short circuit evaluation

Short circuit evaluation
Short circuit evaluation skips evaluating later operands if the result of the logical operator can already be determined.
Animation
3.18.1 Short circuit evaluation: Logical AND.

Table
3.18.1 Short circuit evaluation.
Question set
3.18.2 Determine which operands the program evaluates.

3.21 LAB: Interstate highway numbers

Lab activity
3.21.1 LAB: Interstate highway numbers

3.22 LAB: Leap year

Lab activity

3.22.1 LAB: Leap year

3.23 LAB: Name format

Lab activity
3.23.1 LAB: Name format

4. Loops

4.1 Loops (general)

loop
A loop is a program construct that repeatedly executes the loop's statements (known as the loop body) while the loop's expression is true; when false, execution proceeds past the loop.
loop body
A loop is a program construct that repeatedly executes the loop's statements (known as the loop body) while the loop's expression is true; when false, execution proceeds past the loop.
iteration
Each time through a loop's statements is called an iteration.
Animation
4.1.1 Loop concept: Driving a baby around the block.
Question set
4.1.2 Loop concept.
Animation
4.1.3 A simple loop: Summing the input values.
Animation
4.1.4 Loop example: Computing an average.
Question set
4.1.5 Loop example: Average.
Learning tool
4.1.6 Counting negative values in a list of values.
Question set
4.1.7 Counting negative values.
Learning tool
4.1.8 Find the maximum value in the list of values.
Question set
4.1.9 Determining the max value.

4.2 While loops

while loop
A while loop is a program construct that repeatedly executes a list of sub-statements (known as the loop body) while the loop's expression evaluates to true.
loop body
A while loop is a program construct that repeatedly executes a list of sub-statements (known as the loop body) while the loop's expression evaluates to true.
iteration
Each execution of the loop body is called an iteration.
infinite loop
An infinite loop is a loop that never stops iterating.

Construct

4.2.1 While loop.
Animation
4.2.1 While loop.
Question set
4.2.2 While loops: Number of iterations.

Figure
4.2.1 While loop example: Celsius to Fahrenheit.
Question set
4.2.3 While loop example: Celsius to Fahrenheit.

Figure
4.2.2 Common pattern: Getting input before and at end of loop.
Question set
4.2.4 While loops: Number of iterations, with input gotten before the loop.
Animation
4.2.5 While loop using a relational operator in the loop expression.
Question set
4.2.6 Loop expressions.

Figure
4.2.3 While loop example: Ancestors printing program.
Question set
4.2.7 Ancestors example.
Animation
4.2.8 Infinite loops.
Question set
4.2.9 While loop iterations.
Progression
4.2.1 Enter the output of the while loop.
Coding challenge
4.2.2 Basic while loop with user input.
Coding challenge
4.2.3 Basic while loop expression.

4.3 Do-while loops

do-while loop
A do-while loop is a loop construct that first executes the loop body's statements, then checks the loop condition.

Construct
4.3.1 Do-while loop.
Animation
4.3.1 Do-while loop.
Question set
4.3.2 Do-while loop.
Coding challenge
4.3.1 Basic do-while loop with user input.
Coding challenge
4.3.2 Do-while loop to prompt user input.

4.4 More while examples

sentinel value
A sentinel value is a special value indicating the end of a list, such as a list of positive integers ending with 0, as in 10 1 6 3 0.

Figure
4.4.1 While loop example: GCD (greatest common divisor) program.
Question set
4.4.1 GCD program.

Figure
4.4.2 While loop example: Conversation program.
Question set
4.4.2 Conversation program.

Figure
4.4.3 Computing average of a list with a sentinel.
Question set
4.4.3 Average example with a sentinel.
Progression
4.4.1 While loop with sentinel.
Coding challenge
4.4.2 Bidding example.
Coding challenge
4.4.3 While loop: Insect growth.

4.5 For loops

for loop
A for loop is a loop with three parts at the top: a loop variable initialization, a loop expression, and a loop variable update. A for loop describes iterating a specific number of times more naturally than a while loop.
++i
The statement i = i + 1 is so common that the language supports the shorthand ++i, with ++ known as the increment operator.
increment operator
The statement i = i + 1 is so common that the language supports the shorthand ++i, with ++ known as the increment operator.
--
-- is the decrement operator, --i means i = i - 1.
decrement operator
-- is the decrement operator, --i means i = i - 1.
pre-increment
Two increment operators exist: ++i (pre-increment) and i++ (post-increment). ++i increments before evaluating to a value, while i++ increments after.
post-increment
Two increment operators exist: ++i (pre-increment) and i++ (post-increment). ++i increments before evaluating to a value, while i++ increments after.

Aside
Survey

Construct
4.5.1 For loop.
Animation
4.5.1 For loops.

Figure
4.5.1 A standard way to loop N times, using a for loop.
Question set
4.5.2 For loops.
Question set
4.5.3 For loops.

Aside

Figure
4.5.2 For loop: Savings interest program.
Question set
4.5.4 Savings interest program.

Figure
4.5.3 Computing an average, with first value indicating list size.
Question set
4.5.5 Computing the average.

Table
4.5.1 Choosing between while and for loops: General guidelines (not strict rules though).
Question set
4.5.6 While loops and for loops.
Progression
4.5.1 Enter the for loop's output.

4.6 More for loop examples

prefix form
The ++ operator can appear as ++i (prefix form) or as i++ (postfix form).
postfix form
The ++ operator can appear as ++i (prefix form) or as i++ (postfix form).

Figure
4.6.1 Finding the max in a list.
Question set
4.6.1 Finding the max.

Figure
4.6.2 Outputting multiples of 5 from 10 to 50.
Question set
4.6.2 For loops beyond iterating N times.

Figure
4.6.3 Auto-generate a data table: Celsius to Fahrenheit.
Question set
4.6.3 For loop generating a table of temperature values.
Question set
4.6.4 Miscellaneous for loop and ++ topics.

Figure
4.6.4 Common error: loop variable updated twice.

Figure
4.6.5 Avoid these for loop variations.
Question set
4.6.5 For loop: Common errors / good practice.
Progression
4.6.1 For loops.

4.7 Loops and strings

Figure
4.7.1 Iterating through a string: Counting letters.
Question set
4.7.1 Iterating through a string.

Figure
4.7.2 Iterating until done: Replacing all occurrences of a word.
Question set
4.7.2 Replacing until done.

4.8 Nested loops

nested loop
A nested loop is a loop that appears in the body of another loop.
inner loop
outer loop

Figure
4.8.1 Nested loops example: Two-letter domain name printing program.

Figure
4.8.2 Nested loop example: Histogram.
Question set
4.8.1 Nested loops: Inner loop execution.
Question set
4.8.2 Nested loops: What is the output.
Coding challenge
4.8.1 Nested loops: Indent text.
Coding challenge
4.8.2 Nested loops: Print seats.

4.9 Developing programs incrementally

incrementally
Experienced programmers develop programs incrementally, meaning they create a simple program version, and then grow the program little-by-little into successively more-complete versions.
FIXME comment
A FIXME comment is commonly used to indicate program parts to be fixed or added.

Figure
4.9.1 Incremental program development.

Figure
4.9.2 Second version echoes numbers, and has FIXME comment.

Figure
4.9.3 Third version echoes hyphens too, and handles first three letters.

Figure
4.9.4 Fourth and final version sample input/output.
Question set
4.9.1 Incremental programming.

4.10 Break and continue

break statement
A break statement in a loop causes an immediate exit of the loop.
continue statement
A continue statement in a loop causes an immediate jump to the loop condition check.

Figure
4.10.1 Break statement: Meal finder program.
Question set
4.10.1 Break statements.

Figure
4.10.2 Continue statement: Meal finder program that ensures items purchased is evenly divisible by the number of diners.
Question set
4.10.2 Continue.
Progression
4.10.1 Enter the output of break and continue.
Coding challenge
4.10.2 Simon says.

4.11 Variable name scope

scope
A declared name is only valid within a region of code known as the name's scope.
block
A block is a brace-enclosed {...} sequence of statements, such as found with an if-else, for loop, or while loop.

Animation
4.11.1 Variable name scope extend to the end of the declaration's block.
Question set
4.11.2 Variable name scope.

Table
4.11.1 Index variable declared in a for loop's initialization statement.

Aside
This material avoids declaring index variables in for loops
Question set
4.11.3 For loop index declared in loop's initialization statement.

Figure
4.11.1 Common error: A variable declared within a loop block is (unexpectedly) re-initialized every iteration.
Question set
4.11.4 Common error of a variable declared within a loop block being reinitialized every iteration.

4.12 Enumerations

enumeration type
An enumeration type (enum) declares a name for a new type and possible values for that type.
state machine

Construct
4.12.1 Enumeration type.

Figure
4.12.1 Enumeration example.
Question set
4.12.1 Enumeration syntax.
Question set
4.12.2 Enumerations.
Coding challenge
4.12.1 Enumerations: Grocery items.
Coding challenge
4.12.2 Soda machine with enums.

4.15 LAB: Varied amount of input data

Lab activity
4.15.1 LAB: Varied amount of input data

4.16 LAB: Count characters

Lab activity
4.16.1 LAB: Count characters

4.17 LAB: Checker for integer string

Lab activity
4.17.1 LAB: Checker for integer string

5. Arrays / Vectors

5.1 Array/vector concept (general)

array
An array is a special variable having one name, but storing a list of data items, with each item being directly accessible.
vector
Some languages use a construct similar to an array called a vector.
element
Each item in an array is known as an element.
index
In an array, each element's location number is called the index, myArray[2] has index 2.
Animation
5.1.1 Sometimes a variable should store a list, or array, of data items.

Figure
5.1.1 A normal variable is like a truck, whereas an array variable is like a train.
Learning tool
5.1.2 Update the array's data values.
Question set
5.1.3 Array basics.
Question set
5.1.4 Arrays with element numbering starting with 0.

5.2 Vectors

vector
A vector is an ordered list of items of a given data type.
element
Each item in a vector is called an element.
braces
{_} are braces.
angle brackets
< > are angle brackets, or chevrons.
chevrons
< > are angle brackets, or chevrons.
index
In a vector access, the number in .at() parentheses is called the index of the corresponding element.
size()
A vector's size() function returns the number of vector elements.

Construct
5.2.1 Vector declaration.
Animation
5.2.1 A vector declaration creates multiple variables in memory, each accessible using .at().
Question set
5.2.2 Vector basics.

Figure
5.2.1 Vector's ith element can be directly accessed using .at(i): Oldest people program.
Question set
5.2.3 Nth oldest person program.
Question set
5.2.4 Vector declaration and accesses.

Figure
5.2.2 Vectors combined with loops are powerful together: User-entered numbers.
Question set
5.2.5 Vector with loops.
Question set
5.2.6 Vector initialization.

Aside
Common error: Forgetting to include <vector>
Progression
5.2.1 Enter the output for the vector.
Coding challenge
5.2.2 Printing vector elements.
Coding challenge
5.2.3 Printing vector elements with a for loop.

5.3 Arrays

array
An array is an ordered list of items of a given data type.
element
Each item in an array is called an element.
brackets
[ ] are brackets.
braces
{ } are braces.
index
In an array access, the number in brackets is called the index of the corresponding element.
const

Construct
5.3.1 Array declaration.
Animation
5.3.1 An array declaration creates multiple variables in memory, each accessible using [ ].
Learning tool
5.3.2 Select the index shown.
Question set
5.3.3 Array basics.

Figure
5.3.1 Array's nth element can be directly accessed using [n-1]: Oldest people program.
Question set
5.3.4 Nth oldest person program.

Question set
5.3.5 Array declaration and accesses.

Figure
5.3.2 Arrays combined with loops are powerful together: User-entered numbers.
Question set
5.3.6 Array with loops.
Question set
5.3.7 Array initialization.
Progression
5.3.1 Enter the output for the array.
Coding challenge
5.3.2 Printing array elements.
Coding challenge
5.3.3 Printing array elements with a for loop.

5.4 Array/vector iteration drill

Learning tool
5.4.1 Find the maximum value in the array.
Learning tool
5.4.2 Negative value counting in array.
Learning tool
5.4.3 Array sorting largest value.

5.5 Iterating through arrays

Figure
5.5.1 Common for loop structure for iterating through an array.
Question set
5.5.1 Iterating through an array.

Figure
5.5.2 Iterating through an array example: Program that computes the sum of an array's elements.

Figure
5.5.3 Iterating through an array example: Program that finds the max item.
Question set
5.5.2 Array iteration.
Animation
5.5.3 Writing to an out-of-range index using an array.
Question set
5.5.4 Iterating through an array.
Progression
5.5.1 Enter the output for the array.
Coding challenge
5.5.2 Finding values in arrays.
Coding challenge
5.5.3 Populating an array with a for loop.
Coding challenge

5.5.4 Array iteration: Sum of excess.
Coding challenge
5.5.5 Printing array elements separated by commas.

5.6 Iterating through vectors

Figure
5.6.1 Common for loop structure for iterating through a vector.
Question set
5.6.1 Iterating through a vector.

Figure
5.6.2 Iterating through a vector example: Program that finds the sum of a vector's elements.

Figure
5.6.3 Iterating through a vector example: Program that finds the max item.
Question set
5.6.2 Iterating through vectors.

Figure
5.6.4 Sample error message when accessing an out of range vector index.
Question set
5.6.3 Iterating through a vector.
Progression
5.6.1 Enter the output for the vector.
Coding challenge
5.6.2 Finding values in vectors.
Coding challenge
5.6.3 Populating a vector with a for loop.
Coding challenge
5.6.4 Vector iteration: Sum of excess.
Coding challenge
5.6.5 Printing vector elements separated by commas.

5.7 Multiple vectors

Figure
5.7.1 Multiple vector example: TV watching time program.
Question set
5.7.1 Multiple vectors.
Coding challenge
5.7.1 Printing the sum of two vector elements.
Coding challenge
5.7.2 Multiple vectors: Key and value.

5.8 Vector resize

resize(N)

A vector's size can be set or changed while a program executes using resize(N).
Animation
5.8.1 Vector resize.

Figure
5.8.1 Resizing a vector based on user input.
Question set
5.8.2 Vector resize and size functions.
Coding challenge
5.8.1 Determining the size of a vector.
Coding challenge
5.8.2 Resizing a vector.

5.9 Vector push_back

push_back()
A programmer can append a new element to the end of an existing vector using a vector's push_back() function.
push_back()
back()
pop_back()
Animation
5.9.1 The vector push_back() function.
Question set
5.9.2 Vector push_back().

Table
5.9.1 Functions on the back of a vector.

Figure
5.9.1 Using push_back(), back(), and pop_back(): A grocery list example.
Question set
5.9.3 Vector back() and pop_back() functions.
Coding challenge
5.9.1 Appending a new element to a vector.
Coding challenge
5.9.2 Removing an element from the end of a vector.
Coding challenge
5.9.3 Reading the vector's last element.

5.10 Loop-modifying or copying/comparing vectors

vector copy operation
In C++, the = operator conveniently performs an element-by-element copy of a vector, called a vector copy operation.
vector equality operation
In C++, the == operator conveniently compares vectors element-by-element, called a vector equality operation, with vectorA == vectorB evaluating to true if the vectors are the same size AND each element pair is equal.

Figure
5.10.1 Modifying a vector during iteration example: Converting negatives to 0.
Question set

5.10.1 Modifying a vector in a loop.

Figure
5.10.2 Using = to copy a vector: Original and sale prices.
Question set
5.10.2 Vector copy operation.
Question set
5.10.3 Vector comparing.
Coding challenge
5.10.1 Decrement vector elements.
Coding challenge
5.10.2 Copy and modify vector elements.
Coding challenge
5.10.3 Modify vector elements using other elements.
Coding challenge
5.10.4 Modify a vector's elements.
Coding challenge
5.10.5 Comparing and copying vectors.

5.11 Swapping two variables (General)

Swapping
Swapping two variables x and y means to assign y's value to x, and x's value to y.
temporary variable
A temporary variable is a variable used briefly to store a value.
Animation
5.11.1 Swap idea: Use a temporary location.
Animation
5.11.2 Swapping two variables using a third temporary variable.
Question set
5.11.3 Swap.
Animation
5.11.4 Reversing a list using swaps.
Question set
5.11.5 Reversing a list using swaps.

5.12 Debugging example: Reversing a vector

Figure
5.12.1 First program attempt to reverse vector: Aborts due to invalid access of vector element.

Figure
5.12.2 Revised vector reversing program: Doesn't abort, but still a problem.

Figure
5.12.3 Revised vector reversing program with proper swap: Output isn't reversed.

Figure
5.12.4 Vector reversal program with correct output.

Question set
5.12.1 Find the error in the vector reversal code.

5.13 Arrays vs. vectors

Animation
5.13.1 Writing to an out-of-range index using an array.
Question set
5.13.2 Arrays and vectors.

5.14 Two-dimensional arrays

row-major order
The compiler maps two-dimensional array elements to one-dimensional memory, each row following the previous row, known as row-major order.
Animation
5.14.1 Two-dimensional array.

Figure
5.14.1 Using a two-dimensional array: A driving distance between cities example.

Construct
5.14.1 Initializing a two-dimensional array during declaration.
Question set
5.14.2 Two-dimensional arrays.
Coding challenge
5.14.1 Find 2D array max and min.

5.15 Char arrays / C strings

string
A programmer can use an array to store a sequence of characters, known as a string.
C strings
Char arrays were the only kinds of strings in C++'s predecessor language C, and thus are sometimes called C strings to distinguish them from C++'s string type.
null character
A string in a char array must end with a special character known as a null character, written as '\0'.
null-terminated string
An array of characters ending with a null character is known as a null-terminated string.
Animation
5.15.1 A char array declaration and initialization with null-terminated string.

Figure
5.15.1 Printing stops when reaching the null character at each string's end.
Question set
5.15.2 Char array strings.

Figure
5.15.2 Traversing a C string.
Question set
5.15.3 C string errors.

Figure
5.15.3 A C string is an array of characters, ending with the null character.
Question set
5.15.4 C string without null character.

5.16 Multiple arrays

Figure
5.16.1 Multiple array example: Letter postage cost program.
Question set
5.16.1 Multiple arrays in the above postage cost program.
Question set
5.16.2 Multiple arrays.
Progression
5.16.1 Multiple arrays.

5.17 String library functions

cstring
strcpy()
strncpy()
strcat()
strncat()
strchr()
strlen()
strcmp()

Table
5.17.1 Some C string modification functions.
Question set
5.17.1 String modification functions.

Table
5.17.2 Some C string information functions.
Animation
5.17.2 String comparison.

Figure
5.17.1 Iterating through a C string using strlen.
Animation
5.17.3 Some C string library functions.
Question set
5.17.4 String information functions.

5.18 Char library functions: ctype

isalpha(c)
Isalpha(c) -- Returns true if c is alphabetic: a-z or A-Z.

isdigit(c)
Isdigit(c) -- Returns true if c is a numeric digit: 0-9.
isalnum(c)
Isalnum(c) -- Returns true if c is alphabetic or a numeric digit. Thus, returns true if either isalpha or isdigit would return true.
isspace(c)
Isspace(c) -- Returns true if character c is a whitespace.
islower(c)
Islower(c) -- Returns true if character c is a lowercase letter a-z.
isupper(c)
Isupper(c) -- Returns true if character c is an uppercase letter A-Z.
isblank(c)
Isblank(c) -- Returns true if character c is a blank character. Blank characters include spaces and tabs.
isxdigit(c)
Isxdigit(c) -- Returns true if c is a hexadecimal digit: 0-9, a-f, A-F.
ispunct(c)
Ispunct(c) -- Returns true if c is a punctuation character.
isprint(c)
Isprint(c) -- Returns true if c is a printable character.
iscntrl(c)
Iscntrl(c) -- Returns true if c is a control character.
toupper(c)
Toupper(c) -- If c is a lowercase alphabetic character (a-z), returns the uppercase version (A-Z). If c is not a lowercase alphabetic character, just returns c.
tolower(c)
Tolower(c) -- If c is an uppercase alphabetic character (A-Z), returns the lowercase version (a-z). If c is not an uppercase alphabetic character, just returns c.

Table
5.18.1 Functions that check whether a character is of a given category.

Table
5.18.2 Functions that convert a character is of a given category.

Figure
5.18.1 Use of some functions in cctype.
Question set
5.18.1 Character type functions.

5.21 LAB: Output numbers in reverse

Lab activity
5.21.1 LAB: Output numbers in reverse

5.22 LAB: Middle item

Lab activity
5.22.1 LAB: Middle item

5.23 LAB: Output values below an amount

Lab activity
5.23.1 LAB: Output values below an amount

6. User-Defined Functions

6.1 User-defined function basics

function
Program redundancy can be reduced by creating a grouping of predefined statements for repeatedly used operations, known as a function.
function
A function is a named list of statements.
function definition
A function definition consists of the new function's name and a block of statements.
function call
A function call is an invocation of a function's name, causing the function's statements to execute.
block
A block is a list of statements surrounded by braces.
parameter
A parameter is a function input specified in a function definition.
argument
An argument is a value provided to a function's parameter during a function call.
Animation
6.1.1 Functions can reduce redundancy and keep the main program simple.
Question set
6.1.2 Reasons for functions.
Animation
6.1.3 Function example: Printing a pizza area.
Question set
6.1.4 Function basics.
Animation
6.1.5 Function with parameters example: Printing a pizza area for different diameters.
Question set
6.1.6 Parameters.

Figure
6.1.1 Function with multiple parameters.
Question set
6.1.7 Multiple parameters.
Question set
6.1.8 Calls with multiple parameters.
Progression
6.1.1 Function parameters.
Coding challenge
6.1.2 Basic function call.
Coding challenge
6.1.3 Function call with parameter: Printing formatted measurement.

6.2 Return

return statement
A function may return one value using a return statement.
void

A return type of void indicates that a function does not return any value.
Animation
6.2.1 Function returns computed square.
Question set
6.2.2 Return.
Question set
6.2.3 More on return.
Question set
6.2.4 Calls in an expression.

Figure
6.2.1 Program with a function to convert height in feet/inches to centimeters.

Aside
Question set
6.2.5 Mathematical functions.

Figure
6.2.2 Functions calling functions.
Question set
6.2.6 Functions calling functions.
Progression
6.2.1 Enter the output of the returned value.
Coding challenge
6.2.2 Function call in expression.
Coding challenge
6.2.3 Function definition: Volume of a pyramid.

6.3 Reasons for defining functions

Modular development
Modular development is the process of dividing a program into separate modules that can be developed and tested separately and then integrated into a single program.
Incremental development
Incremental development is a process in which a programmer writes, compiles, and tests a small amount of code, then writes, compiles, and tests a small amount more (an incremental amount), and so on.
function stub
A function stub is a function definition whose statements have not yet been written.

Figure
6.3.1 With program functions: main() is easy to read and understand.

Figure
6.3.2 Without program functions: main() is harder to read and understand.
Question set
6.3.1 Improved readability.
Animation
6.3.2 Function stub used in incremental program development.
Question set
6.3.3 Incremental development.
Animation

6.3.4 Redundant code can be replaced by multiple calls to one function.
Question set
6.3.5 Reasons for defining functions.
Coding challenge
6.3.1 Functions: Factoring out a unit-conversion calculation.
Coding challenge
6.3.2 Function stubs: Statistics.

6.4 Functions with branches/loops

Figure
6.4.1 Function example: Determining fees given an item selling price for an auction website.
Question set
6.4.1 Analyzing the eBay fee calculator.

Figure
6.4.2 User-defined functions make main() easy to understand.
Question set
6.4.2 Analyzing the least common multiple program.
Progression
6.4.1 Output of functions with branches/loops.
Coding challenge
6.4.2 Function with branch: Popcorn.
Coding challenge
6.4.3 Function with loop: Shampoo.

6.5 Unit testing (functions)

Unit testing
Unit testing is the process of individually testing a small part or unit of a program, typically a function.
testbench
A unit test is typically conducted by creating a testbench, a.k.a. test harness, which is a separate program whose sole purpose is to check that a function returns correct output values for a variety of input values.
test vector
Each unique set of input values is known as a test vector.
border cases

Figure
6.5.1 Test harness for the function HrMinToMin().
Question set
6.5.1 Unit testing.

Figure
6.5.2 Test harness with assert for the function HrMinToMin().

Aside
Using branches for unit tests
Question set
6.5.2 Assertions and test cases.

Coding challenge
6.5.1 Unit testing.

6.6 How functions work

stack frame
Animation
6.6.1 Function calls and returns.
Animation
6.6.2 How function call/return works.
Question set
6.6.3 How functions work.

6.7 Functions: Common errors

Figure
6.7.1 Copy-paste common error: Pasted code not properly modified. Find error on the right.
Question set
6.7.1 Copy-pasted sum-of-squares code.

Figure
6.7.2 Missing return statement common error: Program may sometimes work, leading to hard-to-find bug.
Question set
6.7.2 Common function errors.
Question set
6.7.3 Common function errors.
Coding challenge
6.7.1 Function errors: Copying one function to create another.

6.8 Pass by reference

pass by value
Normal parameters are pass by value, meaning the argument's value is copied into a local variable for the parameter.
pass by reference
A pass by reference parameter does *not* create a local copy of the argument, but rather the parameter refers directly to the argument variable's memory location.
reference
A reference is a variable type that refers to another variable.
Animation
6.8.1 Assigning a normal pass by value parameter has no impact on the corresponding argument.
Animation
6.8.2 A pass by reference parameter allows a function to update an argument variable.
Question set
6.8.3 Function definition returns and arguments.
Question set
6.8.4 Function definitions with pass by value and pass by reference.

Figure
6.8.1 Programs should not assign pass by value parameters.

Question set
6.8.5 Assigning a pass by value parameter.

Figure
6.8.2 Reference variable example.
Question set
6.8.6 Reference variables.
Coding challenge
6.8.1 Function pass by reference: Transforming coordinates.

6.9 Functions with string/vector parameters

const
The keyword const can be prepended to a function's vector or string parameter to prevent the function from modifying the parameter.

Figure
6.9.1 Modifying a string parameter, which should be pass by reference.

Figure
6.9.2 Normal and constant pass by reference vector parameters in a vector reversal program.
Question set
6.9.1 Constants and pass by reference.
Question set
6.9.2 Vector parameters.
Coding challenge
6.9.1 Use an existing function.
Coding challenge
6.9.2 Modify a string parameter.
Coding challenge
6.9.3 Modify a vector parameter.

6.10 Functions with C string parameters

Figure
6.10.1 Modifying a C string parameter.
Question set
6.10.1 Functions with string parameters.

Figure
6.10.2 Modifying a C string using a pointer parameter.
Question set
6.10.2 Functions with C string parameters.
Coding challenge
6.10.1 Modify a C string parameter.

6.11 Scope of variable/function definitions

scope

The name of a defined variable or function item is only visible to part of a program, known as the item's scope.

global variable

A variable declared outside any function is called a global variable, in contrast to a *local variable* declared inside a function.

side effects

If a function updates a global variable, the function has effects that go beyond its parameters and return value, known as side effects,.

function declaration

A function declaration specifies the function's return type, name, and parameters, ending with a semicolon where the opening brace would have gone.

function prototype

A function declaration is also known as a function prototype.

Figure

6.11.1 Local variable scope.

Question set

6.11.1 Variable/function scope.

Figure

6.11.2 A function declaration allows a function definition to appear later in a file.

Question set

6.11.2 Function declaration and definition.

6.12 Default parameter values

default parameter value

A function can have a default parameter value for the last parameter(s), meaning a call can optionally omit a corresponding argument.

Figure

6.12.1 Parameter with a default value.

Figure

6.12.2 Valid function calls with default parameter values.

Figure

6.12.3 Compiler error if parameters corresponding to omitted arguments don't have default values.

Question set

6.12.1 Function parameter defaults.

Progression

6.12.1 Functions with default parameters.

Coding challenge

6.12.2 Return number of pennies in total.

6.13 Function name overloading

function name overloading

Sometimes a program has two functions with the same name but differing in the number or types of parameters, known as function name overloading.

function overloading

Figure
6.13.1 Overloaded function name.
Question set
6.13.1 Function name overloading.
Coding challenge
6.13.1 Overload salutation printing.
Coding challenge
6.13.2 Convert a height into inches.

6.14 Parameter error checking

Figure
6.14.1 Function with parameter error checking.
Question set
6.14.1 Checking parameter values.

6.15 Preprocessor and include

preprocessor
The preprocessor is a tool that scans the file from top to bottom looking for any lines that begin with #, known as a hash symbol. Each such line is not a program statement, but rather directs the preprocessor to modify the file in some way before compilation continues, each such line being known as a preprocessor directive.
hash symbol
The preprocessor is a tool that scans the file from top to bottom looking for any lines that begin with #, known as a hash symbol. Each such line is not a program statement, but rather directs the preprocessor to modify the file in some way before compilation continues, each such line being known as a preprocessor directive.
preprocessor directive
The preprocessor is a tool that scans the file from top to bottom looking for any lines that begin with #, known as a hash symbol. Each such line is not a program statement, but rather directs the preprocessor to modify the file in some way before compilation continues, each such line being known as a preprocessor directive.
#include
include directive

Construct
6.15.1 Include directives.
Animation
6.15.1 Preprocessor's handling of an include directive.
Question set
6.15.2 Include directives.

6.16 Separate files

Header file guards
Header file guards are preprocessor directives, which cause the compiler to only include the contents of the header file once.

Figure
6.16.1 Putting related functions in their own file.

Figure
6.16.2 Compiling multiple files together.

Construct
6.16.1 Header file guards.

Figure
6.16.3 All header files should be guarded.
Question set
6.16.1 Header files.

6.17 Functions with array parameters

const
The keyword const can be prepended to a function's array parameter to prevent the function from modifying the parameter.

Figure
6.17.1 Array parameters in an average test score calculation program.
Question set
6.17.1 Functions with array parameters.
Question set
6.17.2 Functions with arrays.
Animation
6.17.3 Functions can modify elements of an array argument.

Figure
6.17.2 Normal and const array parameters in test score adjustment and averaging program.
Question set
6.17.4 Const array function parameters.
Progression
6.17.1 Functions with array parameters.
Coding challenge
6.17.2 Modify an array parameter.

6.18 Functions with array parameters: Common errors

Animation
6.18.1 Common error: Functions with arrays without a parameter indicating the number of array elements.
Question set
6.18.2 Function with array missing parameter indicating number of array elements.

Figure
6.18.1 FindMaxAbsValueIncorrect() incorrectly modifies inputVals array to find element with the largest absolute value.

Figure
6.18.2 FindMaxAbsValue() computes the largest absolute value without modifying the array.

Question set
6.18.3 Functions array parameters.

6.21 LAB: Flip a coin

Lab activity
6.21.1 LAB: Flip a coin

6.22 LAB: Exact change - functions

Lab activity
6.22.1 LAB: Exact change - functions

6.23 LAB: Output values below an amount - functions

Lab activity
6.23.1 LAB: Output values below an amount - functions

7. Objects and Classes

7.1 Objects: Introduction

object
An object is a grouping of data (variables) and operations that can be performed on that data (functions).
Abstraction
Abstraction means to have a user interact with an item at a high-level, with lower-level internal details hidden from the user (aka information hiding or encapsulation).
information hiding
Abstraction means to have a user interact with an item at a high-level, with lower-level internal details hidden from the user (aka information hiding or encapsulation).
encapsulation
Abstraction means to have a user interact with an item at a high-level, with lower-level internal details hidden from the user (aka information hiding or encapsulation).
abstract data type
An abstract data type (ADT) is a data type whose creation and update are constrained to specific well-defined operations.
ADT
An abstract data type (ADT) is a data type whose creation and update are constrained to specific well-defined operations.
Animation
7.1.1 The world is viewed not as materials, but rather as objects.
Animation
7.1.2 Programs commonly are not viewed as variables and functions/methods, but rather as objects.
Question set
7.1.3 Objects.
Animation
7.1.4 Objects strongly support abstraction / information hiding.
Question set
7.1.5 Abstraction / information hiding.

7.2 Using a class

class
The class construct defines a new type that can group data and functions to form an object.
public member functions
A class' public member functions indicate all operations a class user can perform on the object.
object
Declaring a variable of a class type creates an object of that type.
member access operator
The "." operator, known as the member access operator, is used to invoke a function on an object.
Animation
7.2.1 A class example: Restaurant class.
Question set
7.2.2 Using a class.
Animation
7.2.3 Using the Restaurant class.
Question set
7.2.4 Using the Restaurant class.

Figure
7.2.1 Some string public member functions (many more exist).
Question set
7.2.5 Using the string class.

7.3 Defining a class

private data members
A class definition has private data members: variables that member functions can access but class users cannot.
function declaration
A function declaration provides the function's name, return type, and parameter types, but not the function's statements.
function definition
A function definition provides a class name, return type, parameter names and types, and the function's statements.
scope resolution operator
A member function definition has the class name and two colons (::), known as the scope resolution operator, preceding the function's name.
Animation
7.3.1 Private data members.
Question set
7.3.2 Private data members.
Animation
7.3.3 Defining a member function of a class using the scope resolution operator.

Figure
7.3.1 A complete class definition, and use of that class.
Question set
7.3.4 Class definition.

Figure
7.3.2 Simple class example: RunnerInfo.
Question set
7.3.5 Class example: RunnerInfo.
Progression
7.3.1 Classes.

Coding challenge
7.3.2 Basic class use.
Coding challenge
7.3.3 Basic class definition.

7.4 Inline member functions

inline member function
A member function's definition may appear within the class definition, known as an inline member function.
Animation
7.4.1 Inline member functions.

Figure
7.4.1 A class with two inline member functions.
Question set
7.4.2 Inline member functions.

Aside
Exception to variables being declared before used
Question set
7.4.3 Inline member functions.

Aside
Inline member functions on one line
Progression
7.4.1 Inline member functions.

7.5 Mutators, accessors, and private helpers

mutator
A mutator function may modify ("mutate") a class' data members.
accessor
An accessor function accesses data members but does not modify a class' data members.
setter
Commonly, a data member has two associated functions: a mutator for setting the value, and an accessor for getting the value, known as a setter and getter function, respectively, and typically with names starting with set or get.
getter
Commonly, a data member has two associated functions: a mutator for setting the value, and an accessor for getting the value, known as a setter and getter function, respectively, and typically with names starting with set or get.
const
The keyword const after a member function's name and parameters causes a compiler error if the function modifies a data member.
private helper functions
A programmer commonly creates private functions, known as private helper functions, to help public functions carry out tasks.

Figure
7.5.1 Mutator and accessor public member functions.
Question set
7.5.1 Mutators and accessors.
Animation

7.5.2 Private helper member functions.
Question set
7.5.3 Private helper functions.
Progression
7.5.1 Mutators, accessors, and private helpers.

7.6 Initialization and constructors

constructor
C++ has a special class member function, a constructor, called *automatically* when a variable of that class type is declared, and which can initialize data members.
default constructor
A constructor callable without arguments is a default constructor.

Figure
7.6.1 A class definition with initialized data members.
Question set
7.6.1 Initialization.

Figure
7.6.2 Adding a constructor member function to the Restaurant class.
Question set
7.6.2 Default constructors.
Coding challenge
7.6.1 Basic constructor definition.

7.7 Classes and vectors/classes

Figure
7.7.1 Classes and vectors: A reviews program.
Question set
7.7.1 Reviews program.

Figure
7.7.2 Improved reviews program with a Reviews class.
Question set
7.7.2 Reviews program.

Figure
7.7.3 Improved reviews program with a Restaurant class.
Question set
7.7.3 Restaurant program with reviews.
Progression
7.7.1 Enter the output of classes and vectors.
Progression
7.7.2 Writing vectors with classes.

7.8 Separate files for classes

ClassName.h
ClassName.cpp

Figure
7.8.1 Using two separate files for a class.

Aside
Good practice for .cpp and .h files
Question set
7.8.1 Separate files.

Figure
7.8.2 .h and .cpp files for Review, Reviews, and Restaurant classes.
Animation
7.8.2 Restaurant reviews program's main.cpp.
Question set
7.8.3 Restaurant review program .h and .cpp files.
Progression
7.8.1 Enter the output of separate files.

7.9 Choosing classes to create

Animation
7.9.1 Creating a program by first sketching classes.
Question set
7.9.2 Decomposing a program into classes.

Figure
7.9.1 SoccerTeam and TeamPerson classes.
Question set
7.9.3 Coding classes.
Question set
7.9.4 Classes and includes.

7.10 Grouping data: struct

struct
The struct construct defines a new type, which can be used to declare a variable with subitems.
data member
Each struct subitem is called a data member.
member access
For a declared variable, each struct data member can be accessed using ".", known as a member access operator.
dot notation
Question set
7.10.1 Naturally grouped data.
Animation
7.10.2 A struct enables creating a variable with data members.

Construct
7.10.1 Defining and using a new struct type.

Animation
7.10.3 Assigning a struct type.
Question set
7.10.4 The struct construct.
Progression
7.10.1 Enter the output using struct.
Coding challenge
7.10.2 Declaring a struct.
Coding challenge
7.10.3 Accessing a struct's data members.

7.11 Unit testing (classes)

testbench
A testbench is a program whose job is to thoroughly test another program (or portion) via a series of input/output checks known as test cases.
test cases
A testbench is a program whose job is to thoroughly test another program (or portion) via a series of input/output checks known as test cases.
Unit testing
Unit testing means to create and run a testbench for a specific item (or "unit") like a function or a class.
100% code coverage
100% code coverage: Every line of code is executed.
border cases
Border cases: Unusual or extreme test case values.
Regression testing
Regression testing means to retest an item like a class anytime that item is changed; if previously-passed test cases fail, the item has "regressed".
Animation
7.11.1 Unit testing of a class.

Figure
7.11.1 Unit testing of a class.
Question set
7.11.2 Unit testing of a class.
Question set
7.11.3 Regression testing.

Figure
7.11.2 Correct implementation of StatsInfo class.
Animation
7.11.4 Erroneous unit test code causes failures even when StatsInfo is correctly implemented.
Question set
7.11.5 Identifying erroneous test cases.
Progression
7.11.1 Enter the output of the unit tests.
Coding challenge
7.11.2 Unit testing of a class.

7.12 Constructor overloading

overload

A class creator can overload a constructor by defining multiple constructors differing in parameter types.
Animation
7.12.1 Overloaded constructors.
Question set
7.12.2 Overloaded constructors.

Figure
7.12.1 Error - The programmer defined a constructor, so the compiler does not automatically define a default constructor.
Question set
7.12.3 Constructor definitions.

Figure
7.12.2 A constructor with default parameter values can serve as the default constructor.
Question set
7.12.4 Constructor with default parameter values may serve as default constructor.
Progression
7.12.1 Enter the output of the constructor overloading.
Coding challenge
7.12.2 Constructor overloading.

7.13 Constructor initializer lists

constructor initializer list
A constructor initializer list is an alternative approach for initializing data members in a constructor, coming after a colon and consisting of a comma-separated list of variableName(initValue) items.

Figure
7.13.1 Member initialization: (left) Using statements in the constructor, (right) Using a constructor initializer list.
Question set
7.13.1 Member initialization.

Figure
7.13.2 Member initialization in a constructor.
Question set
7.13.2 Constructor initializer list.
Progression
7.13.1 Enter the output of constructor initializer lists.
Progression
7.13.2 Writing constructors.
Coding challenge
7.13.3 Creating a constructor with a constructor initializer list.

7.14 Structs and vectors

Figure
7.14.1 A vector of struct items rather than two vectors of more basic types.
Question set
7.14.1 Using structs with vectors.

Progression
7.14.1 Enter the output of the struct and vector.
Progression
7.14.2 Structs and vectors.

7.15 The 'this' implicit parameter

implicit parameter
The object variable before the function name is known as an implicit parameter of the member function.
this
Within a member function, the implicitly-passed object pointer is accessible via the name this.

Figure
7.15.1 Using 'this' to refer to an object's member.
Question set
7.15.1 The 'this' implicit parameter.
Animation
7.15.2 How a member function works.
Question set
7.15.3 Using the 'this' pointer in member functions and constructors.
Progression
7.15.1 Enter the output of the function.
Coding challenge
7.15.2 The this implicit parameter.

7.16 Structs and functions

Animation
7.16.1 Using a struct that is returned from a function; the struct's data members are copied upon return.
Question set
7.16.2 Functions returning struct values.
Question set
7.16.3 Functions with struct parameters.
Progression
7.16.1 Enter the output of the struct and function.
Progression
7.16.2 Structs and functions.

7.17 Operator overloading

operator overloading
C++ allows a programmer to redefine the functionality of built-in operators like +, -, and *, to operate on programmer-defined objects, a process known as operator overloading.
Animation
7.17.1 Operator overloading allows use of operators like + on classes.
Question set
7.17.2 Operator overloading.

Figure
7.17.1 TimeHrMn class implementation with overloaded + operator.
Animation
7.17.3 TimeHrMn::operator+ is called when two TimeHrMn objects are added with the + operator.

Question set
7.17.4 Operator overloading basics.

Figure
7.17.2 Overloading the + operator multiple times.
Question set
7.17.5 Determining which function is invoked.
Progression
7.17.1 Enter the output of operator overloading.
Coding challenge
7.17.2 Operator overloading.

7.18 Overloading comparison operators

sort()
The sort() function, defined in the C++ Standard Template Library's (STL) algorithms library, can sort vectors containing objects of programmer-defined classes.
Animation
7.18.1 Overloading the == operator.
Question set
7.18.2 Overloading == operator for Restaurant class.

Figure
7.18.1 Overloading the Reviews class' < operator.
Question set
7.18.3 Overloading the < operator.
Question set
7.18.4 Overloading comparison operators.

Figure
7.18.2 Sorting a vector of Review objects.
Question set
7.18.5 Sorting vectors.
Progression
7.18.1 Enter the output of the program using overloading operators.

7.19 Vector ADT

standard template library
The standard template library (STL) defines classes for common Abstract Data Types (ADTs).
STL
The standard template library (STL) defines classes for common Abstract Data Types (ADTs).
vector
A vector is an ADT of an ordered, indexable list of items.
at()
size()
empty()
clear()
push_back()
erase()
insert()

Table
7.19.1 Vector ADT functions.
Question set
7.19.1 Vector functions at(), size(), empty(), and clear().
Animation
7.19.2 Vector push_back() member function.

Figure
7.19.1 Using vector member functions: A player jersey numbers program.
Question set
7.19.3 push_back() function.
Animation
7.19.4 insert() and erase() vector member functions.

Figure
7.19.2 Using the vector erase() function.
Animation
7.19.5 Intuitive depiction of how to add items to a vector while maintaining items in ascending sorted order.
Question set
7.19.6 The insert() and erase() functions.
Progression
7.19.1 Enter the output of the vector ADT functions.
Coding challenge
7.19.2 Modifying vectors.

7.20 Structs, vectors, and functions: A seat reservation example

Figure
7.20.1 A seat reservation system involving a struct, vector, and functions.
Question set
7.20.1 Seat reservation example with struct, vector, and functions.

7.21 Namespaces

name conflict
A name conflict occurs when two or more items like variables, classes, or functions, have the same name.
namespace
A namespace defines a region (or scope) used to prevent name conflicts.
scope resolution operator
The scope resolution operator :: allows specifying in which namespace to find a name.
::
The scope resolution operator :: allows specifying in which namespace to find a name.
std
All items in the C++ standard library are part of the std namespace (short for standard).
Namespace directive
Namespace directive: A programmer can add the statement using `namespace std;` to direct the compiler to check the std namespace for any names later in the file that aren't otherwise declared.
using

Namespace directive: A programmer can add the statement using `namespace std;` to direct the compiler to check the std namespace for any names later in the file that aren't otherwise declared.
Animation
7.21.1 Namespaces can resolve name conflicts.
Question set
7.21.2 Namespaces.
Question set
7.21.3 std namespace.
Progression
7.21.1 Enter the output from the proper namespace.

7.22 Static data members and functions

static
The keyword static indicates a variable is allocated in memory only once during a program's execution.
static data member
A static data member is a data member of the class instead of a data member of each class object.
static member function
A static member function is a class function that is independent of class objects.
Animation
7.22.1 Static data member used to create object ID numbers.
Question set
7.22.2 Static data members.

Figure
7.22.1 Static member function used to access a private static data member.
Question set
7.22.3 Static member functions.
Progression
7.22.1 Enter the output with static members.

7.25 LAB*: Warm up: Online shopping cart (Part 1)

Lab activity
7.25.1 LAB*: Warm up: Online shopping cart (Part 1)

7.26 LAB: Triangle area comparison (classes)

Lab activity
7.26.1 LAB: Triangle area comparison (classes)

7.27 LAB: Winning team (classes)

Lab activity
7.27.1 LAB: Winning team (classes)

8. Streams

8.1 Output and input streams

ostream

An ostream, short for "output stream," is a class that supports output, available via `#include <iostream>` and in namespace "std".

<< operator

Ostream provides the << operator, known as the insertion operator, for converting different types of data into a sequence of characters.

insertion operator

Ostream provides the << operator, known as the insertion operator, for converting different types of data into a sequence of characters.

cout

Cout is a predefined ostream object (declared as `ostream cout;` in the iostream library) that is pre-associated with a system's standard output, usually a computer screen.

istream

An istream, short for "input stream," is a class that supports input.

>> operator

Istream provides the >> operator, known as the extraction operator, to extract data from a data buffer and write the data into different types of variables.

extraction operator

Istream provides the >> operator, known as the extraction operator, to extract data from a data buffer and write the data into different types of variables.

cin

Cin is a predefined istream pre-associated with a system's standard input, usually a computer keyboard.

Animation

8.1.1 ostream supports output.

Question set

8.1.2 ostream and cout.

Animation

8.1.3 istream and the extraction operator support input.

Question set

8.1.4 istream and cin.

8.2 Output formatting

manipulator

A manipulator is a function that overloads the insertion operator << or extraction operator >> to adjust the way output appears. Manipulators are defined in the iomanip and ios libraries in namespace std.

Animation

8.2.1 Floating-point manipulators.

Table

8.2.1 Floating-point manipulators.

Question set

8.2.2 Output formatting for floating-point manipulators.

Animation

8.2.3 Text-alignment manipulators.

Table

8.2.2 Text-alignment manipulators.

Question set

8.2.4 Output formatting for text manipulators.

Table

8.2.3 Buffer manipulators.

Question set
8.2.5 Buffer manipulators.
Progression
8.2.1 Output formatting.
Coding challenge
8.2.2 Output formatting: Printing a maximum number of digits.
Coding challenge
8.2.3 Output formatting: Printing a maximum number of digits in the fraction.

8.3 Input string stream

input string stream
A new input string stream variable of type istringstream can be created that reads input from an associated string instead of the keyboard (standard input).
istringstream
A new input string stream variable of type istringstream can be created that reads input from an associated string instead of the keyboard (standard input).
eof()
Input streams have a Boolean function called eof() or end of file that returns true or false depending on whether or not the end of the stream has been reached.
end of file
Input streams have a Boolean function called eof() or end of file that returns true or false depending on whether or not the end of the stream has been reached.
Animation
8.3.1 Reading a string as an input stream.
Question set
8.3.2 Input string streams.

Figure
8.3.1 Using a string stream to process a line of input text.
Question set
8.3.3 Using getline() with string streams.
Animation
8.3.4 Reaching the end of a string stream.
Question set
8.3.5 Reaching the end of a string stream.

Figure
8.3.2 Input stream example: Phone number formats.
Question set
8.3.6 String stream error state.
Progression
8.3.1 Input string streams.
Coding challenge
8.3.2 Reading from a string.

8.4 Output string stream

output string stream
An output string stream variable of type ostringstream can insert characters into a string buffer instead of the screen (standard output).
ostringstream

An output string stream variable of type ostringstream can insert characters into a string buffer instead of the screen (standard output).
str()
The ostringstream member function str() returns the contents of an ostringstream buffer as a string.
Animation
8.4.1 Using an output string stream.
Question set
8.4.2 Output string streams.

Figure
8.4.1 Output string stream example.
Question set
8.4.3 Output string stream example.
Progression
8.4.1 Output string streams.
Coding challenge
8.4.2 Output using string stream.

8.5 File input

eof()
The eof() function returns true if the previous stream operation reached the end of the file.
fail()
The fail() function returns true if the previous stream operation had an error.
stream error
A stream error occurs when insertion or extraction fails, causing the stream to enter an error state.
good()
Good() returns true if no stream errors have occurred.
eof()
Eof() returns value of eofbit, if end-of-file reached on extraction.
fail()
Fail() returns true if either failbit or badbit is set, indicating an error for the previous stream operation.
bad()
Bad() returns true if badbit is set, indicating the stream is bad.
Animation
8.5.1 Input from a file.
Question set
8.5.2 File input.

Aside
File open() with C strings in older C++ versions

Figure
8.5.1 Reading a varying amount of data from a file.
Question set
8.5.3 File input.

Figure
8.5.2 How many times a word appears in a file.
Question set
8.5.4 Counting instances of specific word example.

Figure
8.5.3 Program that reads business reviews from a file and computes the average rating.
Question set
8.5.5 Reading and using data from a file.

Table
8.5.1 Stream error state flags and functions to check error state.
Animation
8.5.6 Check for errors while reading a file.
Question set
8.5.7 Stream error functions.

8.6 C++ example: Parsing and validating input files

Figure
8.6.1 Program that reads from teams.txt.
Question set
8.6.1 Parsing input files.

8.7 File output

ofstream
An ofstream, short for "output file stream", is a class that supports writing to a file.

Table
8.7.1 Basic steps for opening and writing a file.
Question set
8.7.1 Opening and writing to a file.
Animation
8.7.2 Writing to an output text file.
Question set
8.7.3 Writing a text file.
Animation
8.7.4 Writing simple html to a file and to the console.
Question set
8.7.5 Writing a simple HTML file.

8.8 C++ example: Saving and retrieving program data

Figure
8.8.1 Program that reads restaurant reviews from reviews.txt.
Question set
8.8.1 Saving and retrieving program data.
Question set
8.8.2 Save the reviews.

8.9 Overloading stream operators

insertion operator
The C++ << operator is known as the insertion operator.
extraction operator
The C++ >> operator is known as the extraction operator.

Figure
8.9.1 VoteCounter class demo.
Question set
8.9.1 VoteCounter << operator.

Figure
8.9.2 WaitingLine class.
Animation
8.9.2 The insertion and extraction operators add and remove from the WaitingLine object.
Question set
8.9.3 Overloading the extraction operator.

Figure
8.9.3 WaitingLine class with stream friend functions.
Animation
8.9.4 Using cin and cout with the WaitingLine class.
Question set
8.9.5 Using cin and cout with the WaitingLine class.

8.10 LAB: Warm up: Parsing strings

Lab activity
8.10.1 LAB: Warm up: Parsing strings

8.11 LAB*: Program: Data visualization

Lab activity
8.11.1 LAB*: Program: Data visualization

8.12 LAB: Parsing dates

Lab activity
8.12.1 LAB: Parsing dates

9. Searching and Sorting Algorithms

9.1 Searching and algorithms

algorithm
An algorithm is a sequence of steps for accomplishing a task.
Linear search
Linear search is a search algorithm that starts from the beginning of a list, and checks each element until the search key is found or the end of the list is reached.
runtime
An algorithm's runtime is the time the algorithm takes to execute.
Animation

9.1.1 Linear search algorithm checks each element until key is found.

Figure
9.1.1 Linear search algorithm.
Question set
9.1.2 Linear search algorithm execution.
Question set
9.1.3 Linear search runtime.

9.2 O notation

Big O notation
Big O notation is a mathematical way of describing how a function (running time of an algorithm) generally behaves in relation to the input size.
Animation
9.2.1 Determining Big O notation of a function.
Question set
9.2.2 Big O notation.

Figure
9.2.1 Rules for determining Big O notation of composite functions.
Question set
9.2.3 Big O notation for composite functions.

Table
9.2.1 Growth rates for different input sizes.
Learning tool
9.2.4 Computational complexity graphing tool.

Figure
9.2.2 Runtime complexities for various pseudocode examples.
Question set
9.2.5 Big O notation and growth rates.

9.3 Algorithm analysis

worst-case runtime
The worst-case runtime of an algorithm is the runtime complexity for an input that results in the longest execution.
Animation
9.3.1 Runtime analysis: Finding the max value.
Question set
9.3.2 Worst-case runtime analysis.
Animation
9.3.3 Simplified runtime analysis: A constant number of constant time operations is O(1).
Question set
9.3.4 Constant time operations.
Animation
9.3.5 Runtime analysis of nested loop: Selection sort algorithm.

Figure

9.3.1 Common summation: Summation of consecutive numbers.
Question set
9.3.6 Nested loops.

9.4 Sorting: Introduction

Sorting
Sorting is the process of converting a list of elements into ascending (or descending) order.
Learning tool
9.4.1 Sort by swapping tool.
Question set
9.4.2 Sorted elements.

9.5 Selection sort

Selection sort
Selection sort is a sorting algorithm that treats the input as two parts, a sorted part and an unsorted part, and repeatedly selects the proper next value to move from the unsorted part to the end of the sorted part.
Animation
9.5.1 Selection sort.
Question set
9.5.2 Selection sort algorithm execution.

Figure
9.5.1 Selection sort algorithm.
Question set
9.5.3 Selection sort runtime.
Progression
9.5.1 Selection sort.

9.6 Binary search

Binary search
Binary search is a faster algorithm for searching a list if the list's elements are sorted and directly accessible (such as an array).
Animation
9.6.1 Using binary search to search contacts on your phone.
Question set
9.6.2 Using binary search to search a contact list.
Animation
9.6.3 Binary search efficiently searches sorted list by reducing the search space by half each iteration.

Figure
9.6.1 Binary search algorithm.
Question set
9.6.4 Binary search algorithm execution.
Animation
9.6.5 Speed of linear search versus binary search to find a number within a sorted list.
Question set
9.6.6 Linear and binary search efficiency.
Progression
9.6.1 Binary search.

9.7 Insertion sort

Insertion sort
Insertion sort is a sorting algorithm that treats the input as two parts, a sorted part and an unsorted part, and repeatedly inserts the next value from the unsorted part into the correct location in the sorted part.
nearly sorted
A nearly sorted list only contains a few elements not in sorted order.
Animation
9.7.1 Insertion sort.

Figure
9.7.1 Insertion sort algorithm.
Question set
9.7.2 Insertion sort algorithm execution.
Question set
9.7.3 Insertion sort runtime.
Question set
9.7.4 Nearly sorted lists.
Animation
9.7.5 Using insertion sort for nearly sorted list.
Question set
9.7.6 Insertion sort algorithm execution for nearly sorted input.

9.8 Binary search

Binary search
Binary search is a faster algorithm for searching a list if the list's elements are sorted and directly accessible (such as an array).
Animation
9.8.1 Using binary search to search contacts on your phone.
Question set
9.8.2 Using binary search to search a contact list.
Animation
9.8.3 Binary search efficiently searches sorted list by reducing the search space by half each iteration.

Figure
9.8.1 Binary search algorithm.
Question set
9.8.4 Binary search algorithm execution.
Animation
9.8.5 Speed of linear search versus binary search to find a number within a sorted list.
Question set
9.8.6 Linear and binary search runtime.

10. Pointers

10.1 Why pointers?

pointer
A pointer is a variable that contains a memory address.
dynamically allocated array

The C++ vector class is a container that internally uses a dynamically allocated array, an array whose size can change during runtime.
vector insert/erase performance problem
If a program has a vector with thousands of elements, a single call to insert() or erase() can require thousands of instructions and cause the program to run very slowly, often called the vector insert/erase performance problem.
linked list
A linked list consists of items that contain both data and a pointer—a *link*—to the next list item.
this
When a class member function is called on an object, a pointer to the object is automatically passed to the member function as an implicit parameter called the this pointer.
Animation
10.1.1 Dynamically allocated arrays.
Question set
10.1.2 Dynamically allocated arrays.
Animation
10.1.3 Vector insert performance problem.
Animation
10.1.4 A list avoids the shifting problem.

Table
10.1.1 Comparing vectors and linked lists.
Question set
10.1.5 Inserting/erasing in vectors vs. linked lists.
Animation
10.1.6 Pointers used to call class member functions.
Question set
10.1.7 The 'this' pointer.

10.2 Pointer basics

pointer
A pointer is a variable that holds a memory address, rather than holding data like most variables.
reference operator
The reference operator (&) obtains a variable's address.
dereference operator
The dereference operator (*) is prepended to a pointer variable's name to retrieve the data to which the pointer variable points.
Null
Null means "nothing".
nullptr
A pointer that is assigned with the keyword nullptr is said to be null.
Animation
10.2.1 Assigning a pointer with an address.

Aside
Printing memory addresses
Question set
10.2.2 Declaring and initializing a pointer.
Animation
10.2.3 Using the dereference operator.
Question set
10.2.4 Dereferencing a pointer.
Animation

10.2.5 Checking to see if a pointer is null.

Aside
Null pointer
Question set
10.2.6 Null pointer.
Animation
10.2.7 Common pointer errors.
Question set
10.2.8 Common pointer errors.

Aside
Two pointer declaration styles

Aside
Advanced compilers can check for common errors
Progression
10.2.1 Enter the output of pointer content.
Coding challenge
10.2.2 Printing with pointers.

10.3 Operators: new, delete, and ->

new operator
The new operator allocates memory for the given type and returns a pointer to the allocated memory.
member access operator
When using a pointer to an object, the member access operator (->) allows access to the object's members with the syntax `a->b` instead of `(*a).b`.
->
When using a pointer to an object, the member access operator (->) allows access to the object's members with the syntax `a->b` instead of `(*a).b`.
delete operator
The delete operator deallocates (or frees) a block of memory that was allocated with the new operator.
delete[] operator
The delete[] operator is used to free an array allocated with the new operator.
Animation
10.3.1 The new operator allocates space for an object, then calls the constructor.
Question set
10.3.2 The new operator.
Animation
10.3.3 Constructor arguments.
Question set
10.3.4 Constructor arguments.

Table
10.3.1 Using the member access operator.
Question set
10.3.5 The member access operator.
Animation
10.3.6 The delete operator.
Question set

10.3.7 The delete operator.
Animation
10.3.8 Allocating and deleting an array of Point objects.
Question set
10.3.9 Allocating and deleting object arrays.
Progression
10.3.1 Operators: new, delete, and ->.
Coding challenge
10.3.2 Deallocating memory

10.4 Array-based lists

array-based list
An array-based list is a list ADT implemented using an array.
append
Given a new element, the append operation for an array-based list of length X inserts the new element at the end of the list, or at index X.
Prepend
The Prepend operation for an array-based list inserts a new item at the start of the list.
InsertAfter
The InsertAfter operation for an array-based list inserts a new item after a specified index.
search
Given a key, the search operation returns the index for the first element whose data matches that key, or -1 if not found.
remove-at
Given the index of an item in an array-based list, the remove-at operation removes the item at that index.
Animation
10.4.1 Appending to array-based lists.
Question set
10.4.2 Array-based lists.
Animation
10.4.3 Array-based list resize operation.
Question set
10.4.4 Array-based list resize operation.

Aside
InsertAt operation.
Animation
10.4.5 Array-based list prepend and insert after operations.
Question set
10.4.6 Array-based list prepend and insert after operations.
Animation
10.4.7 Array-based list search and remove-at operations.
Question set
10.4.8 Search and remove-at operations.
Question set
10.4.9 Search and remove-at operations.
Progression
10.4.1 Array-based lists.

10.5 String functions with pointers

strchr()

strrchr()
strstr()
Animation
10.5.1 strcmp() and strcpy() string functions.
Question set
10.5.2 C string library functions.

Table
10.5.1 Some C string search functions.
Question set
10.5.3 C string search functions.

Figure
10.5.1 String searching example.
Question set
10.5.4 Modifying and searching strings.
Progression
10.5.1 Enter the output of the string functions.
Coding challenge
10.5.2 Find char in C string
Coding challenge
10.5.3 Find C string in C string.

10.6 Memory regions: Heap/Stack

Code
Static memory
The stack
automatic memory
The heap
free store
Animation
10.6.1 Use of the four memory regions.
Question set
10.6.2 Stack and heap definitions.

10.7 A first linked list

list node

Figure
10.7.1 A basic example to introduce linked lists.
Animation
10.7.1 Inserting nodes into a basic linked list.
Question set
10.7.2 A first linked list.

Figure
10.7.2 Managing many new items using just a few pointer variables.
Progression
10.7.1 Enter the output of the program using Linked List.

Coding challenge
10.7.2 Linked list negative values counting.

10.8 Destructors

destructor
A destructor is a special class member function that is called automatically when a variable of that class type is destroyed.
Animation
10.8.1 LinkedList nodes are not deallocated without a LinkedList class destructor.
Question set
10.8.2 LinkedList class destructor.

Figure
10.8.1 LinkedListNode and LinkedList classes.
Animation
10.8.3 The LinkedList class destructor, called when the list is deleted, frees all nodes.
Question set
10.8.4 LinkedList class destructor.
Animation
10.8.5 Destructors are called automatically only for non-reference/pointer variables.
Question set
10.8.6 When a destructor is called.
Progression
10.8.1 Enter the output of the destructors.
Coding challenge
10.8.2 Write a destructor.

10.9 Memory leaks

memory leak
A memory leak occurs when a program that allocates memory loses the ability to access the allocated memory, typically due to failure to properly destroy/free dynamically allocated memory.
garbage collection
Some programming languages, such as Java, use a mechanism called garbage collection wherein a program's executable includes automatic behavior that at various intervals finds all unreachable allocated memory locations (e.g., by comparing all reachable memory with all previously-allocated memory), and automatically frees such unreachable memory.
Animation
10.9.1 Memory leak can use up all available memory.

Aside
Garbage collection
Question set
10.9.2 Memory leaks.
Animation
10.9.3 Lack of destructor yields memory leak.
Question set
10.9.4 Memory not freed in a destructor.
Question set
10.9.5 Which results in a memory leak?

10.10 Copy constructors

copy constructor
The solution is to create a copy constructor, a constructor that is automatically called when an object of the class type is passed by value to a function and when an object is initialized by copying another object during declaration.
deep copy
The copy constructor makes a new copy of all data members (including pointers), known as a deep copy.
shallow copy
Creating a copy of an object by copying only the data members' values creates a shallow copy of the object.
Animation
10.10.1 Copying an object without a copy constructor.
Question set
10.10.2 Copying an object without a copy constructor.

Construct
10.10.1 Copy constructor.

Figure
10.10.1 Problem solved by creating a copy constructor that does a deep copy.

Aside
Copy constructors in more complicated situations
Question set
10.10.3 Determining which constructor will be called.
Progression
10.10.1 Enter the output of the copy constructors.
Coding challenge
10.10.2 Write a copy constructor.

10.11 Copy assignment operator

deep copy
Allocating and copying data for pointer members is known as a deep copy.
Animation
10.11.1 Basic assignment operation fails when pointer member involved.
Question set
10.11.2 Default assignment operator behavior.

Figure
10.11.1 Assignment operator performs a deep copy.
Question set
10.11.3 Assignment operator.
Progression
10.11.1 Enter the output of the program with an overloaded assignment operator.
Coding challenge
10.11.2 Write a copy assignment.

10.12 Rule of three

rule of three

The rule of three describes a practice that if a programmer explicitly defines any one of those three special member functions (destructor, copy constructor, copy assignment operator), then the programmer should explicitly define all three.
the big three
Animation
10.12.1 Rule of three.

Aside
Default destructors, copy constructors, and assignment operators
Question set
10.12.2 Rule of three.

10.14 Lab 1 - Pointer Introduction

Lab activity
10.14.1 Lab 1 - Pointer Introduction

10.15 Lab 2 - Dynamic Memory

Lab activity
10.15.1 Lab 2 - Dynamic Memory

10.16 Practicum 1

Lab activity
10.16.1 Practicum 1

10.17 Practicum 1 - Online students only

Lab activity
10.17.1 Practicum 1 - Online students only

10.18 Lab 3 - C-string functions

Lab activity
10.18.1 Lab 3 - C-string functions

11. Inheritance

11.1 Derived classes

derived class
A derived class (or subclass) is a class that is derived from another class, called a base class (or superclass).
subclass
A derived class (or subclass) is a class that is derived from another class, called a base class (or superclass).
base class
A derived class (or subclass) is a class that is derived from another class, called a base class (or superclass).
superclass
A derived class (or subclass) is a class that is derived from another class, called a base class (or superclass).
inheritance
The derived class is said to inherit the properties of the base class, a concept called inheritance.

Animation
11.1.1 Creating a ProduceItem from GenericItem.
Question set
11.1.2 Derived class concept.

Figure
11.1.1 Class ProduceItem is derived from class GenericItem.
Animation
11.1.3 Using GenericItem and ProduceItem objects.
Question set
11.1.4 Derived classes.
Learning tool
11.1.5 Interactive inheritance tree.
Question set
11.1.6 Inheritance scenarios.

Figure
11.1.2 Inheritance example: Business and Restaurant classes.
Question set
11.1.7 Inheritance example.
Progression
11.1.1 Derived classes.
Coding challenge
11.1.2 Basic inheritance.

11.2 Access by members of derived classes

protected

Figure
11.2.1 Member functions of a derived class cannot access private members of the base class.
Question set
11.2.1 Access by derived class members.

Figure
11.2.2 Access specifiers -- Protected allows access by derived classes but not by others.

Table
11.2.1 Access specifiers.
Question set
11.2.2 Protected access specifier.
Question set
11.2.3 Access specifiers for class definitions.

11.3 Overriding member functions

override
When a derived class defines a member function that has the same name and parameters as a base class's function, the member function is said to override the base class's function.
Animation

11.3.1 Overriding member function example.

Aside
Overriding vs. overloading
Question set
11.3.2 Overriding.

Figure
11.3.1 Function calling overridden function of base class.
Question set
11.3.3 Override example.
Progression
11.3.1 Overriding member function.
Coding challenge
11.3.2 Basic derived class member override.

11.4 Polymorphism and virtual member functions

Polymorphism
Polymorphism refers to determining which program behavior to execute depending on data types.
Compile-time polymorphism
Compile-time polymorphism is when the compiler determines which function to call at compile-time.
Runtime polymorphism
Runtime polymorphism is when the compiler is unable to determine which function to call at compile-time, so the determination is made while the program is running.
derived/base class pointer conversion
The program above uses a C++ feature called derived/base class pointer conversion, where a pointer to a derived class is converted to a pointer to the base class without explicit casting.
virtual function
A virtual function is a member function that may be overridden in a derived class and is used for runtime polymorphism.
override
The override keyword is an optional keyword used to indicate that a virtual function is overridden in a derived class.
virtual table
To implement virtual functions, the compiler creates a virtual table that allows the computer to quickly lookup which function to call at runtime.
pure virtual function
A pure virtual function is a virtual function that provides no definition in the base class, and all derived classes must override the function.
abstract base class
A class that has at least one pure virtual function is known as an abstract base class.
Animation
11.4.1 Compile-time polymorphism vs. runtime polymorphism.
Question set
11.4.2 Polymorphism.

Aside
Polymorphism word origin
Animation
11.4.3 Adding a virtual function to the base class.

Aside
Virtual table

Figure
11.4.1 Runtime polymorphism via a virtual function.
Question set
11.4.4 Polymorphism with virtual functions.

Figure
11.4.2 Business is an abstract base class.
Question set
11.4.5 Pure virtual functions.

Aside
Possible warning messages when using virtual functions
Progression
11.4.1 Polymorphism and virtual member functions.
Coding challenge
11.4.2 Basic polymorphism.

11.5 Abstract classes: Introduction (generic)

abstract class
An abstract class is a class that guides the design of subclasses but cannot itself be instantiated as an object.
Animation
11.5.1 Classes, inheritance, and abstract classes.
Question set
11.5.2 Classes, inheritance, and abstract classes.
Animation
11.5.3 Biological classification uses abstract classes.
Question set
11.5.4 Abstract classes.

11.6 Abstract classes

pure virtual function
A pure virtual function is a virtual function that is not implemented in the base class, thus all derived classes must override the function.
abstract class
An abstract class is a class that cannot be instantiated as an object, but is the superclass for a subclass and specifies how the subclass must be implemented. Any class with one or more pure virtual functions is abstract.
concrete class
A concrete class is a class that is not abstract, and hence *can* be instantiated.
Animation
11.6.1 A Shape class with a pure virtual function is an abstract class.
Question set
11.6.2 Shape class.

Figure

11.6.1 Shape is an abstract class. Circle and Rectangle are concrete classes that extend the Shape class.
Question set
11.6.3 Shape classes.
Progression
11.6.1 Abstract classes.

11.7 Is-a versus has-a relationships

Unified Modeling Language (UML)
Programmers commonly draw class inheritance relationships using Unified Modeling Language (UML) notation.

Figure
11.7.1 Composition.

Figure
11.7.2 Inheritance.
Question set
11.7.1 Is-a vs. has-a relationships.
Animation
11.7.2 UML derived class example: ProduceItem derived from GenericItem.

11.8 UML

Universal Modeling Language
The Universal Modeling Language (UML) is a language for software design that uses different types of diagrams to visualize the structure and behavior of programs.
UML
The Universal Modeling Language (UML) is a language for software design that uses different types of diagrams to visualize the structure and behavior of programs.
structural diagram
A structural diagram visualizes static elements of software, such as the variables and functions used in the program.
behavioral diagram
A behavioral diagram visualizes dynamic behavior of software, such as the flow of an algorithm.
class diagram
A UML class diagram is a structural diagram that can be used to visually model the classes of a computer program, including member variables and functions.
Animation
11.8.1 UML class diagrams show class names, members, types, and access.
Question set
11.8.2 UML class diagrams.
Animation
11.8.3 UML uses italics for abstract classes and methods.
Question set
11.8.4 UML for inheritance.

11.11 LAB: Pet information (derived classes)

Lab activity
11.11.1 LAB: Pet information (derived classes)

11.12 LAB: Instrument information

Lab activity
11.12.1 LAB: Instrument information

11.13 LAB: Course information (derived classes)

Lab activity
11.13.1 LAB: Course information (derived classes)

11.14 LAB: Book information (overriding member functions)

Lab activity
11.14.1 LAB: Book information (overriding member functions)

11.15 LAB: Plant information (vector)

Lab activity
11.15.1 LAB: Plant information (vector)

11.16 LAB: Library book sorting

Lab activity
11.16.1 LAB: Library book sorting

11.17 Lab 4 - The Big Four, Inheritance, and Dynamically Resizing Arrays

Lab activity
11.17.1 Lab 4 - The Big Four, Inheritance, and Dynamically Resizing Arrays

11.18 Practicum 2

Lab activity
11.18.1 Practicum 2

11.19 Practicum 2 - alt

Lab activity
11.19.1 Practicum 2 - alt

11.20 Lab 5 - Doggies

Lab activity
11.20.1 Lab 5 - Doggies

12. Recursion

12.1 Recursion: Introduction

algorithm
An algorithm is a sequence of steps for solving a problem.
recursive algorithm

A recursive algorithm is an algorithm that breaks the problem into smaller subproblems and applies the same algorithm to solve the smaller subproblems.
base case
At some point, a recursive algorithm must describe how to actually do something, known as the base case.

Figure
12.1.1 Algorithms are like recipes.

Figure
12.1.2 Mowing the lawn can be broken down into a recursive process.
Question set
12.1.1 Recursion.

12.2 Recursive functions

recursive function
A function that calls itself is a recursive function.
Animation
12.2.1 A recursive function example.
Question set
12.2.2 Thinking about recursion.
Coding challenge
12.2.1 Calling a recursive function.

12.3 Recursive algorithm: Search

binary search
A binary search algorithm begins at the midpoint of the range and halves the range after each guess.
base case
Animation
12.3.1 Binary search: A well-known recursive algorithm.

Figure
12.3.1 A recursive function carrying out a binary search algorithm.

Aside
Calculating the middle value
Learning tool
12.3.2 Binary search tree tool.

Figure
12.3.2 Recursively searching a sorted list.
Question set
12.3.3 Recursive search algorithm.
Question set
12.3.4 Recursive calls.
Progression
12.3.1 Enter the output of binary search.

12.4 Adding output statements for debugging

conditional compilation

Figure
12.4.1 Output statements can help debug recursive functions, especially if indented based on recursion depth.
Question set
12.4.1 Recursive debug statements.

12.5 Creating a recursive function

base case
Animation
12.5.1 Writing a recursive function for factorial: First write the base case, then add the recursive case.
Question set
12.5.2 Creating recursion.

Figure
12.5.1 Non-recursive solution to compute N!
Question set
12.5.3 When recursion is appropriate.
Coding challenge
12.5.1 Recursive function: Writing the base case.
Coding challenge
12.5.2 Recursive function: Writing the recursive case.

12.6 Recursive math functions

Fibonacci sequence
The Fibonacci sequence is 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, etc.; starting with 0, 1, the pattern is to compute the next number by adding the previous two numbers.
greatest common divisor
The greatest common divisor (GCD) is the largest number that divides evenly into two numbers.

Figure
12.6.1 Fibonacci sequence step-by-step.

Figure
12.6.2 Calculate greatest common divisor of two numbers.
Question set
12.6.1 Recursive GCD example.
Coding challenge
12.6.1 Writing a recursive math function.

12.7 Recursive exploration of all possibilities

Animation
12.7.1 Exploring all possibilities viewed as a tree of choices.

Figure
12.7.1 Scramble a word's letters in every possible way.
Question set

12.7.2 Letter scramble.

Figure
12.7.2 Shopping spree in which a user can fit 3 items in a shopping bag.
Question set
12.7.3 All letter combinations.

Figure
12.7.3 Find distance of traveling to 3 cities.
Question set
12.7.4 Recursive exploration.
Progression
12.7.1 Enter the output of recursive exploration.

12.8 Stack overflow

stack frame
Each function call places a new stack frame on the stack, for local parameters, local variables, and more function items.
stack overflow
Deep recursion could fill the stack region and cause a stack overflow, meaning a stack frame extends beyond the memory region allocated for stack, .
Animation
12.8.1 Recursion causing stack overflow.
Question set
12.8.2 Stack overflow.

12.9 Quicksort

Quicksort
Quicksort is a sorting algorithm that repeatedly partitions the input into low and high parts (each part unsorted), and then recursively sorts each of those parts.
pivot
The pivot can be any value within the array being sorted, commonly the value of the middle array element.
Animation
12.9.1 Quicksort partitions data into a low part with data less than/equal to a pivot value and a high part with data greater than/equal to a pivot value.
Question set
12.9.2 Quicksort pivot location and value.
Question set
12.9.3 Low and high partitions.
Animation
12.9.4 Quicksort.

Figure
12.9.1 Quicksort algorithm.
Learning tool
12.9.5 Quicksort tool.
Question set
12.9.6 Quicksort runtime.
Question set
12.9.7 Worst case quicksort runtime.

12.10 Merge sort

Merge sort
Merge sort is a sorting algorithm that divides a list into two halves, recursively sorts each half, and then merges the sorted halves to produce a sorted list.
Animation
12.10.1 Merge sort recursively divides the input into two halves, sorts each half, and merges the lists together.
Question set
12.10.2 Merge sort partitioning.
Animation
12.10.3 Merging partitions: Smallest element from left or right partition is added one at a time to a temporary merged list. Once merged, temporary list is copied back to the original list.
Question set
12.10.4 Tracing merge operation.

Figure
12.10.1 Merge sort algorithm.
Question set
12.10.5 Merge sort runtime and memory complexity.

12.12 LAB: Descending selection sort with output during execution

Lab activity
12.12.1 LAB: Descending selection sort with output during execution

12.13 LAB: Sorting user IDs

Lab activity
12.13.1 LAB: Sorting user IDs

12.14 LAB: All permutations of names

Lab activity
12.14.1 LAB: All permutations of names

12.15 LAB: Number pattern

Lab activity
12.15.1 LAB: Number pattern

13. Exceptions

13.1 Exception basics

Error-checking code
Error-checking code is code a programmer writes to detect and handle errors that occur during program execution.
exception
An exception is a circumstance that a program was not designed to handle, .
exception-handling constructs
The language has special constructs, try, throw, and catch, known as exception-handling constructs, to keep error-checking code separate and to reduce redundant checks.

try
A try block surrounds normal code, which is exited immediately if a throw statement executes.
throw
A throw statement appears within a try block; if reached, execution jumps immediately to the end of the try block.
stdexcept library
catch
A catch clause immediately follows a try block; if the catch was reached due to an exception thrown of the catch clause's parameter type, the clause executes.
handler
A catch block is called a handler because it handles an exception.

Figure
13.1.1 BMI example without error checking.

Figure
13.1.2 BMI example with error-checking code but without using exception-handling constructs.

Construct
13.1.1 Exception-handling constructs.
Animation
13.1.1 How try, throw, and catch handle exceptions.

Figure
13.1.3 BMI example with error-checking code using exception-handling constructs.
Question set
13.1.2 Exceptions.
Question set
13.1.3 Exception basics.
Progression
13.1.1 Exception handling.

13.2 Exceptions with functions

Figure
13.2.1 BMI example using exception-handling constructs along with functions.
Question set
13.2.1 Exceptions.

13.3 Multiple handlers

catch(...)
Catch(...) is a catch-all handler that catches any type, which is useful when listed as the last handler.

Construct
13.3.1 Exception-handling: multiple handlers.
Animation
13.3.1 Multiple handlers.
Question set
13.3.2 Exceptions with multiple handlers.

13.5 LAB: Exception handling to detect input string vs. int

Lab activity
13.5.1 LAB: Exception handling to detect input string vs. int

14. Templates

14.1 Function templates

function template
A function template is a function definition having a special type parameter that may be used in place of types in the function.
type parameter

Figure
14.1.1 Functions may have identical behavior, differing only in data types.

Figure
14.1.2 A function template enables a function to handle various data types.
Question set
14.1.1 Function templates.

Construct
14.1.1 Function template with multiple parameters.

14.2 Class templates

class template
A class template is a class definition having a special type parameter that may be used in place of types in the class.
template parameter

Figure
14.2.1 Classes may be nearly identical, differing only in data type.

Figure
14.2.2 A class template enables one class to handle various data types.
Question set
14.2.1 Class templates.

Construct
14.2.1 Class template with multiple parameters.

14.4 Practice Practicum

Lab activity
14.4.1 Practice Practicum

14.5 Practice Practicum

Lab activity
14.5.1 Practice Practicum

14.6 Practicum 3

Lab activity
14.6.1 Practicum 3

14.7 Practicum 3 - Thursday

Lab activity
14.7.1 Practicum 3 - Thursday

14.8 LAB: What order? (function templates)

Lab activity
14.8.1 LAB: What order? (function templates)

14.9 LAB: Zip code and population (class templates)

Lab activity
14.9.1 LAB: Zip code and population (class templates)

14.10 Lab 6 - Class Templates

Lab activity
14.10.1 Lab 6 - Class Templates

14.11 Lab 7 - Templates, Binary Files, and Exceptions

Lab activity
14.11.1 Lab 7 - Templates, Binary Files, and Exceptions

15. Lists, Stacks, and Queues

15.1 Abstract data types

abstract data type
An abstract data type (ADT) is a data type described by predefined user operations, such as "insert data at rear," without indicating how each operation is implemented.
ADT
An abstract data type (ADT) is a data type described by predefined user operations, such as "insert data at rear," without indicating how each operation is implemented.
list
A list is an ADT for holding ordered data.
dynamic array
A dynamic array is an ADT for holding ordered data and allowing indexed access.
stack
A stack is an ADT in which items are only inserted on or removed from the top of a stack.
queue
A queue is an ADT in which items are inserted at the end of the queue and removed from the front of the queue.

deque
A deque (pronounced "deck" and short for double-ended queue) is an ADT in which items can be inserted and removed at both the front and back.
bag
A bag is an ADT for storing items in which the order does not matter and duplicate items are allowed.
set
A set is an ADT for a collection of distinct items.
priority queue
A priority queue is a queue where each item has a priority, and items with higher priority are closer to the front of the queue than items with lower priority.
dictionary
A dictionary is an ADT that associates (or maps) keys with values.
Animation
15.1.1 List ADT using array and linked lists data structures.
Question set
15.1.2 Abstract data types.

Table
15.1.1 Common ADTs.
Question set
15.1.3 Common ADTs.

15.2 Applications of ADTs

Animation
15.2.1 Programming using ADTs.
Question set
15.2.2 Programming with ADTs.

Table
15.2.1 Standard libraries in various programming languages.
Question set
15.2.3 ADTs in standard libraries.

15.3 Algorithm efficiency

Algorithm efficiency
Algorithm efficiency is typically measured by the algorithm's computational complexity.
Computational complexity
Computational complexity is the amount of resources used by the algorithm.
runtime complexity
An algorithm's runtime complexity is a function, T(N), that represents the number of constant time operations performed by the algorithm on an input of size N.
best case
An algorithm's best case is the scenario where the algorithm does the minimum possible number of operations.
worst case
An algorithm's worst case is the scenario where the algorithm does the maximum possible number of operations.
space complexity
An algorithm's space complexity is a function, S(N), that represents the number of fixed-size memory units used by the algorithm for an input of size N.
auxiliary space complexity
An algorithm's auxiliary space complexity is the space complexity not including the input data.

Animation
15.3.1 Computational complexity.
Question set
15.3.2 Algorithm efficiency and computational complexity.

Aside
Input data size must remain a variable
Animation
15.3.3 Linear search best and worst cases.
Question set
15.3.4 FindFirstLessThan algorithm best and worst case.
Question set
15.3.5 Best and worst case concepts.
Animation
15.3.6 FindMax space complexity and auxiliary space complexity.
Question set
15.3.7 Space complexity of GetEvens function.

15.4 List abstract data type (ADT)

list
A list is a common ADT for holding ordered data, having operations like append a data item, remove a data item, search whether a data item exists, and print the list.
Animation
15.4.1 List ADT.
Question set
15.4.2 List ADT.

Table
15.4.1 Some common operations for a list ADT.
Question set
15.4.3 List ADT common operations.

15.5 Singly-linked lists

singly-linked list
A singly-linked list is a data structure for implementing a list ADT, where each node has data and a pointer to the next node.
head
A singly-linked list's first node is called the head, and the last node the tail.
tail
A singly-linked list's first node is called the head, and the last node the tail.
positional list
A singly-linked list is a type of positional list: A list where elements contain pointers to the next and/or previous elements in the list.
null
Null is a special value indicating a pointer points to nothing.
Append
Given a new node, the Append operation for a singly-linked list inserts the new node after the list's tail node.
Prepend
Given a new node, the Prepend operation for a singly-linked list inserts the new node before the list's head node.

Aside
null
Animation
15.5.1 Singly-linked list: Each node points to the next node.
Question set
15.5.2 Singly-linked list data structure.
Animation
15.5.3 Singly-linked list: Appending a node.
Question set
15.5.4 Appending a node to a singly-linked list.
Animation
15.5.5 Singly-linked list: Prepending a node.
Question set
15.5.6 Prepending a node in a singly-linked list.
Progression
15.5.1 Singly-linked lists.

15.6 Singly-linked lists: Insert

InsertAfter
Given a new node, the InsertAfter operation for a singly-linked list inserts the new node after a provided existing list node.
Animation
15.6.1 Singly-linked list: Insert nodes.
Question set
15.6.2 Inserting nodes in a singly-linked list.
Question set
15.6.3 Singly-linked list insert-after algorithm.
Progression
15.6.1 Singly-linked lists: Insert.

15.7 Singly-linked lists: Remove

RemoveAfter
Given a specified existing node in a singly-linked list, the RemoveAfter operation removes the node after the specified list node.
Animation
15.7.1 Singly-linked list: Node removal.
Question set
15.7.2 Removing nodes from a singly-linked list.
Question set
15.7.3 ListRemoveAfter algorithm execution: Intermediate node.
Question set
15.7.4 ListRemoveAfter algorithm execution: List head node.
Progression
15.7.1 Singly-linked lists: Remove.

15.8 Linked list search

search
Given a key, a search algorithm returns the first node whose data matches that key, or returns null if a matching node was not found.
Animation

15.8.1 Singly-linked list: Searching.
Question set
15.8.2 ListSearch algorithm execution.
Question set
15.8.3 Searching a linked-list.
Progression
15.8.1 Linked list search.

15.9 Doubly-linked lists

doubly-linked list
A doubly-linked list is a data structure for implementing a list ADT, where each node has data, a pointer to the next node, and a pointer to the previous node.
positional list
A doubly-linked list is a type of positional list: A list where elements contain pointers to the next and/or previous elements in the list.
Append
Given a new node, the Append operation for a doubly-linked list inserts the new node after the list's tail node.
Prepend
Given a new node, the Prepend operation of a doubly-linked list inserts the new node before the list's head node and points the head pointer to the new node.
Question set
15.9.1 Doubly-linked list data structure.
Animation
15.9.2 Doubly-linked list: Appending a node.
Question set
15.9.3 Doubly-linked list data structure.
Animation
15.9.4 Doubly-linked list: Prepending a node.
Question set
15.9.5 Prepending a node in a doubly-linked list.
Progression
15.9.1 Doubly-linked lists.

15.10 Doubly-linked lists: Insert

InsertAfter
Given a new node, the InsertAfter operation for a doubly-linked list inserts the new node after a provided existing list node.
Animation
15.10.1 Doubly-linked list: Inserting nodes.
Question set
15.10.2 Inserting nodes in a doubly-linked list.
Progression
15.10.1 Doubly-linked lists: Insert.

15.11 Doubly-linked lists: Remove

Remove
The Remove operation for a doubly-linked list removes a provided existing list node.
Animation
15.11.1 Doubly-linked list: Node removal.
Question set

15.11.2 Deleting nodes from a doubly-linked list.
Question set
15.11.3 ListRemove algorithm execution: Intermediate node.
Question set
15.11.4 ListRemove algorithm execution: List head node.
Progression
15.11.1 Doubly-linked lists: Remove.

15.12 Linked list traversal

list traversal
A list traversal algorithm visits all nodes in the list once and performs an operation on each node.
reverse traversal
A reverse traversal visits all nodes starting with the list's tail node and ending after visiting the list's head node.

Figure
15.12.1 Linked list traversal algorithm.
Animation
15.12.1 Singly-linked list: List traversal.
Question set
15.12.2 List traversal.

Figure
15.12.2 Reverse traversal algorithm.
Question set
15.12.3 Reverse traversal algorithm execution.

15.13 Sorting linked lists

Animation
15.13.1 Sorting a doubly-linked list with insertion sort.
Question set
15.13.2 Insertion sort for doubly-linked lists.

Aside
Algorithm efficiency
Animation
15.13.3 Sorting a singly-linked list with insertion sort.

Figure
15.13.1 ListFindInsertionPosition algorithm.
Question set
15.13.4 Sorting singly-linked lists with insertion sort.

Aside
Algorithm efficiency

Table
15.13.1 Sorting algorithms easily adapted to efficiently sort linked lists.

Table
15.13.2 Sorting algorithms that cannot as efficiently sort linked lists.
Question set
15.13.5 Sorting linked-lists vs. sorting arrays.

15.14 Linked lists: Recursion

Animation
15.14.1 Recursive forward traversal.
Question set
15.14.2 Forward traversal in a linked list with 10 nodes.
Question set
15.14.3 Forward traversal concepts.

Figure
15.14.1 ListSearch and ListSearchRecursive functions.
Question set
15.14.4 Searching a linked list with 10 nodes.
Animation
15.14.5 Recursive reverse traversal.
Question set
15.14.6 Reverse traversal concepts.

15.15 Stack abstract data type (ADT)

stack
A stack is an ADT in which items are only inserted on or removed from the top of a stack.
push
The stack push operation inserts an item on the top of the stack.
pop
The stack pop operation removes and returns the item at the top of the stack.
last-in first-out
A stack is referred to as a last-in first-out ADT.
Animation
15.15.1 Stack ADT.
Question set
15.15.2 Stack ADT: Push and pop operations.

Table
15.15.1 Common stack ADT operations.
Question set
15.15.3 Common stack ADT operations.
Progression
15.15.1 Stack ADT.

15.16 Stacks using linked lists

Animation
15.16.1 Stack implementation using a linked list.
Question set

15.16.2 Stack push and pop operations with a linked list.
Progression
15.16.1 Stacks using linked lists.

15.17 Queue abstract data type (ADT)

queue
A queue is an ADT in which items are inserted at the end of the queue and removed from the front of the queue.
enqueue
The queue enqueue operation inserts an item at the end of the queue.
dequeue
The queue dequeue operation removes and returns the item at the front of the queue.
first-in first-out
A queue is referred to as a first-in first-out ADT.
Animation
15.17.1 Queue ADT.
Question set
15.17.2 Queue ADT.

Table
15.17.1 Some common operations for a queue ADT.
Question set
15.17.3 Common queue ADT operations.
Progression
15.17.1 Queue ADT.

15.18 Queues using linked lists

Animation
15.18.1 Queue implemented using a linked list.
Question set
15.18.2 Queue push and pop operations with a linked list.
Progression
15.18.1 Queues using linked lists.

15.19 Deque abstract data type (ADT)

deque
A deque (pronounced "deck" and short for double-ended queue) is an ADT in which items can be inserted and removed at both the front and back.
peek
A peek operation returns an item in the deque without removing the item.
Animation
15.19.1 Deque ADT.
Question set
15.19.2 Deque ADT.

Table
15.19.1 Common deque ADT operations.
Question set
15.19.3 Common queue ADT operations.
Progression

15.19.1 Deque ADT.

15.20 Lab 7 - C Linked List

Lab activity
15.20.1 Lab 7 - C Linked List

15.21 Lab 8 - Circular Linked List

Lab activity
15.21.1 Lab 8 - Circular Linked List

15.22 Practicum 4

Lab activity
15.22.1 Practicum 4

16. Containers

16.1 Range-based for loop

range-based for loop
The range-based for loop is a for loop that iterates through each element in a vector or container.
for each loop
A range-based for loop is also known as a for each loop.
Animation
16.1.1 The range-based for loop declares a new variable and assigns the variable with each successive element of a container.
Question set
16.1.2 Range-based for loop.

Figure
16.1.1 Range-based for loop decreases the amount of code needed to iterate through a vector.
Question set
16.1.3 Using range-based for loops.

Figure
16.1.2 Modifying a vector using a range-based for loop.
Question set
16.1.4 Range-based for loop modifying loops vector.
Progression
16.1.1 Range-based for loop.

Figure
16.1.3 Range-based for loop with auto.
Question set
16.1.5 Using auto with range-based for loops.

16.2 List

list
The list class defined within the C++ Standard Template Library (STL) defines a container of ordered elements, i.e., a sequence.
front()
The front() function returns the first element of a list.
back()
The back() function returns the last element of a list.
pop_front()
The pop_front() and pop_back() functions remove the first and last elements of a list, respectively.
pop_back()
The pop_front() and pop_back() functions remove the first and last elements of a list, respectively.
remove()
The remove() function removes specific existing elements from the list.
front()
back()
push_back()
push_front()
size()
pop_back()
pop_front()
remove()
iterator
An iterator is an object that points to a location in a list and can be used to traverse the list bidirectionally.
begin()
The begin() function returns an iterator to the first element in the list.
end()
The end() function returns an iterator to a position after the last element in the list.
begin()
end()
insert()
The insert() function adds one or more elements before the specified iterator position.
erase()
The erase() function can remove one or multiple elements starting at the specified iterator position.
insert()
erase()
Animation
16.2.1 push_back() adds elements to end of list and push_front() add element to front of the list.
Question set
16.2.2 list's push_back() and push_front() functions.
Animation
16.2.3 pop_front() removes the first element and remove() removes specific elements from the list.

Table
16.2.1 Common list functions.
Question set
16.2.4 Using list's front(), back(), pop_front(), pop_back(), and remove() functions.
Animation
16.2.5 The list class provides functions to access elements of a list using an iterator.

Table
16.2.2 Common list functions that use an iterator.
Question set
16.2.6 Using list's begin() and end() functions to traverse and modify a list.

Table
16.2.3 list functions for inserting and removing elements using iterators.
Question set
16.2.7 Using list's insert() and erase() functions to modify a list.

Figure
16.2.1 Using a range-based for loop with a list.
Question set
16.2.8 Range-based for loop list traversal.
Progression
16.2.1 List.

16.3 Pair

pair
The pair class in the C++ Standard Template Library (STL) defines a container that consists of two data elements.
make_pair()
The make_pair() function creates a pair with the specified first and second values, with which a pair variable can be assigned.

Figure
16.3.1 Creating a pair and accessing the pair's elements.
Question set
16.3.1 STL pair objects.

16.4 Map

map
The map class within the C++ Standard Template Library (STL) defines a container that associates (or maps) keys to values.
emplace()
at()
count()
erase()
clear()
Animation
16.4.1 A map allows a programmer to map keys to values.
Animation
16.4.2 Updating the value associated with a key.
Question set
16.4.3 Basic map operations: emplace() and at().
Question set
16.4.4 Determining if map contains a key.

Table
16.4.1 Common map functions.
Question set
16.4.5 Common map functions.
Progression
16.4.1 Map functions.

Aside
[ ] operator

Aside
emplace()

16.5 Set

set
The set class defined within the C++ Standard Template Library (STL) defines a collection of unique elements.
insert()
erase()
count()
size()
Animation
16.5.1 A set's insert(), erase(), and count() functions add an item, remove an item, and check if an item exists within the set.
Question set
16.5.2 set operations: insert(), erase(), and count().
Animation
16.5.3 A set's insert() and erase() functions return a value to indicate the operation's failure or success.
Question set
16.5.4 Return value of set's insert() and erase() functions.

Table
16.5.1 Common set functions.
Progression
16.5.1 Using a set to define unique elements.

16.6 Queue

queue
The queue class defined within the C++ Standard Template Library (STL) defines a container of ordered elements that supports element insertion at the tail and element retrieval from the head.
push()
A queue's push() function adds an element to the tail of the queue and increases the queue's size by one.
front()
A queue's front() function returns the element at the head of the queue.
pop()
A queue's pop() function removes the element at the head of the queue.
push()
pop()
front()
back()
size()
Animation
16.6.1 queue: push() adds an element to the tail of the queue, front() returns element at the head of the queue, and pop() removes the element at the head of the queue.
Question set
16.6.2 Using queue's push(), front(), and pop() functions to insert and retrieve elements.

Table
16.6.1 Common queue functions.
Progression
16.6.1 Queue.

16.7 Deque

deque
The deque (pronounced "deck") class defined within the C++ Standard Template Library (STL) defines a container of ordered elements that supports element insertion and removal at both ends (i.e., at the head and tail of the deque).
push_front()
Deque's push_front() function adds an element at the head of the deque and increases the deque's size by one.
front()
Deque's front() function returns the element at the head of the deque.
pop_front()
And, deque's pop_front() function removes the element at the head of the deque.
stack
A stack is an ADT in which elements are only added or removed from the top of a stack.
push_front()
push_back()
pop_front()
pop_back()
front()
back()
size()
Animation
16.7.1 deque: push_front() function adds an element at the head, front() function returns the element at the head, and pop_front() function removes the element at the head.
Question set
16.7.2 Use deque's push_front(), front(), and pop_front() functions to insert and retrieve elements.

Table
16.7.1 Common deque functions.
Progression
16.7.1 Enter the output for deque.

16.8 find() function

find()
The find() function seeks a specific value in a range of elements. find() is defined in the algorithms library of the Standard Template Library (STL) .
find_if()
The find_if() function is a variation of the find() function, defined in the STL algorithms library, that searches a range for an element that satisfies a boolean condition.
Animation
16.8.1 The find() function.
Question set
16.8.2 Using find() to locate elements in a list.

Figure

16.8.1 Using the find_if() function to find the first string formatted like ##.#.
Question set
16.8.3 The find_if() function.

16.9 sort() function

sort()
The sort() function arranges a container's elements in ascending order.
Animation
16.9.1 Using sort().
Question set
16.9.2 Using sort() to sort a vector's elements.
Animation
16.9.3 Using sort() with a custom comparison function to sort in descending order.
Question set
16.9.4 sort() with a custom comparison function.

Figure
16.9.1 Using the sort() function and custom comparison functions to sort a vector of students in different ways.
Question set
16.9.5 Using sort() with custom data types.

16.10 LAB: Grocery shopping list (list)

Lab activity
16.10.1 LAB: Grocery shopping list (list)

16.11 LAB: Student grades (map)

Lab activity
16.11.1 LAB: Student grades (map)

16.12 LAB: Ticketing service (queue)

Lab activity
16.12.1 LAB: Ticketing service (queue)

17. Trees

17.1 Binary trees

binary tree
In a binary tree, each node has up to two children, known as a *left child* and a *right child*.
Leaf
Leaf: A tree node with no children.
Internal node
Internal node: A node with at least one child.
Parent
Parent: A node with a child is said to be that child's parent.
ancestors
A node's ancestors include the node's parent, the parent's parent, etc., up to the tree's root.

Root
Root: The one tree node with no parent (the "top" node).
edge
The link from a node to a child is called an edge.
depth
A node's depth is the number of edges on the path from the root to the node.
level
All nodes with the same depth form a tree level.
height
A tree's height is the largest depth of any node.
full
A binary tree is full if every node contains 0 or 2 children.
complete
A binary tree is complete if all levels, except possibly the last level, contain all possible nodes and all nodes in the last level are as far left as possible.
perfect
A binary tree is perfect, if all internal nodes have 2 children and all leaf nodes are at the same level.
Animation
17.1.1 Binary tree basics.
Question set
17.1.2 Binary tree basics.
Animation
17.1.3 Binary tree terminology: height, depth, and level.
Question set
17.1.4 Binary tree height, depth, and level.

Figure
17.1.1 Special types of binary trees: full, complete, perfect.
Question set
17.1.5 Identifying special types of binary trees.
Progression
17.1.1 Binary trees.

17.2 Applications of trees

Binary space partitioning
Binary space partitioning (BSP) is a technique of repeatedly separating a region of space into 2 parts and cataloging objects contained within the regions.
BSP
Binary space partitioning (BSP) is a technique of repeatedly separating a region of space into 2 parts and cataloging objects contained within the regions.
BSP tree
A BSP tree is a binary tree used to store information for binary space partitioning.
Animation
17.2.1 A file system is a hierarchy that can be represented by a tree.
Question set
17.2.2 Analyzing a file system tree.
Question set
17.2.3 File system trees.
Animation
17.2.4 A BSP tree is used to quickly determine which objects do not need to be rendered.
Question set
17.2.5 Binary space partitioning.

Aside
Using trees to store collections

17.3 Binary search trees

binary search tree
A binary search tree (BST), which has an ordering property that any node's left subtree keys ≤ the node's key, and the right subtree's keys ≥ the node's key. That property enables fast searching for an item.
search
To search nodes means to find a node with a desired key, if such a node exists.
successor
A BST node's successor is the node that comes after in the BST ordering, so in A B C, A's successor is B, and B's successor is C.
predecessor
A BST node's predecessor is the node that comes before in the BST ordering.

Figure
17.3.1 BST ordering property: For three nodes, left child is less-than-or-equal-to parent, parent is less-than-or-equal-to right child. For more nodes, all keys in subtrees must satisfy the property, for every node.
Animation
17.3.1 BST ordering properties.
Question set
17.3.2 Binary search tree: Basic ordering property.

Figure
17.3.2 Searching a BST.
Animation
17.3.3 A BST may yield faster searches than a list.
Question set
17.3.4 Searching a BST.

Table
17.3.1 Minimum binary tree heights for N nodes are equivalent to $\lfloor log_2 N \rfloor$.
Animation
17.3.5 Searching a perfect BST with N nodes requires only O($logN$) comparisons.
Question set
17.3.6 Searching BSTs with N nodes.

Figure
17.3.3 A BST defines an ordering among nodes.
Question set
17.3.7 Binary search tree: Defined ordering.
Progression
17.3.1 Binary search trees.

17.4 BST search algorithm

search

Given a key, a search algorithm returns the first node found matching that key, or returns null if a matching node is not found.
Animation
17.4.1 BST search algorithm.
Question set
17.4.2 BST search algorithm.
Question set
17.4.3 BST search algorithm decisions.
Question set
17.4.4 Tracing a BST search.
Progression
17.4.1 BST search algorithm.

17.5 BST insert algorithm

insert
Given a new node, a BST insert operation inserts the new node in a proper location obeying the BST ordering property.
Animation
17.5.1 Binary search tree insertions.
Question set
17.5.2 BST insert algorithm.
Question set
17.5.3 BST insert algorithm decisions.
Question set
17.5.4 Tracing BST insertions.

Aside
BST insert algorithm complexity
Progression
17.5.1 BST insert algorithm.

17.6 BST remove algorithm

remove
Given a key, a BST remove operation removes the first-found matching node, restructuring the tree to preserve the BST ordering property.
Animation
17.6.1 BST remove: Removing a leaf, or an internal node with a single child.
Animation
17.6.2 BST remove: Removing internal node with two children.

Figure
17.6.1 BST remove algorithm.

Aside
BST remove algorithm complexity
Question set
17.6.3 BST remove algorithm.
Progression
17.6.1 BST remove algorithm.

17.7 BST inorder traversal

tree traversal
A tree traversal algorithm visits all nodes in the tree once and performs an operation on each node.
inorder traversal
An inorder traversal visits all nodes in a BST from smallest to largest.

Figure
17.7.1 BST inorder traversal algorithm.
Animation
17.7.1 BST inorder print algorithm.
Question set
17.7.2 Inorder traversal of a BST.

17.8 BST height and insertion order

height
A tree's height is the maximum edges from the root to any leaf.
Animation
17.8.1 Inserting in random order keeps tree height near the minimum. Inserting in sorted order yields the maximum.
Question set
17.8.2 BST height.
Animation
17.8.3 BSTGetHeight algorithm.
Question set
17.8.4 BSTGetHeight algorithm.

17.9 BST parent node pointers

Figure
17.9.1 BSTInsert algorithm for BSTs with nodes containing parent pointers.

Figure
17.9.2 BSTReplaceChild algorithm.

Figure
17.9.3 BSTRemoveKey and BSTRemoveNode algorithms for BSTs with nodes containing parent pointers.
Question set
17.9.1 BST parent node pointers.

17.10 BST: Recursion

Animation
17.10.1 BST recursive search algorithm.
Question set
17.10.2 BST recursive search algorithm.

Figure

17.10.1 BST get parent algorithm.
Question set
17.10.3 BST get parent algorithm.

Figure
17.10.2 Recursive BST insertion and removal.
Question set
17.10.4 Recursive BST insertion and removal.

17.11 Tries

trie
A trie (or prefix tree) is a tree representing a set of strings.
prefix tree
A trie (or prefix tree) is a tree representing a set of strings.
terminal node
A terminal node is a node that represents a terminating character, which is the end of a string in the trie.
trie insert
Given a string, a trie insert operation creates a path from the root to a terminal node that visits all the string's characters in sequence.
trie search
Given a string, a trie search operation returns the terminal node corresponding to that string, or null if the string is not in the trie.
trie remove
Given a string, a trie remove operation removes the string's corresponding terminal node and all non-root ancestors with 0 children.
Animation
17.11.1 Trie representing the set of strings: bat, cat, and cats.
Question set
17.11.2 Trie representing the set of strings: bat, cat, and cats.
Animation
17.11.3 Trie insert algorithm.
Question set
17.11.4 Trie insert algorithm.
Animation
17.11.5 Trie search algorithm.
Question set
17.11.6 Trie search algorithm.
Animation
17.11.7 Trie remove algorithm.
Question set
17.11.8 Trie remove algorithm.

Aside
Trie time complexities

18. Additional Labs: Variables / Assignments

18.1 LAB: Pizza Party

Lab activity
18.1.1 LAB: Pizza Party

18.2 LAB: Square Root

Lab activity
18.2.1 LAB: Square Root

18.3 LAB: Volume and area of cylinder

Lab activity
18.3.1 LAB: Volume and area of cylinder

18.4 LAB: Ordering pizza

Lab activity
18.4.1 LAB: Ordering pizza

18.5 LAB: Postfix of 3

Lab activity
18.5.1 LAB: Postfix of 3

18.6 LAB: Prefix of 3

Lab activity
18.6.1 LAB: Prefix of 3

18.7 LAB: Convert from seconds

Lab activity
18.7.1 LAB: Convert from seconds

18.8 LAB: Convert to seconds

Lab activity
18.8.1 LAB: Convert to seconds

18.9 LAB: Hypotenuse

Lab activity
18.9.1 LAB: Hypotenuse

18.10 LAB: Midfix of 3

Lab activity
18.10.1 LAB: Midfix of 3

18.11 LAB: Area of a triangle

Lab activity
18.11.1 LAB: Area of a triangle

19. Additional Labs: Branches

19.1 LAB: Max of 2

Lab activity
19.1.1 LAB: Max of 2

19.2 LAB: Max of 3

Lab activity
19.2.1 LAB: Max of 3

19.3 LAB: Longest string

Lab activity
19.3.1 LAB: Longest string

19.4 LAB: Golf scores

Lab activity
19.4.1 LAB: Golf scores

20. Additional Labs: Loops

20.1 LAB: Count multiples

Lab activity
20.1.1 LAB: Count multiples

20.2 LAB: Find largest number

Lab activity
20.2.1 LAB: Find largest number

20.3 LAB: Hailstone sequence

Lab activity
20.3.1 LAB: Hailstone sequence

21. Additional Labs: Arrays / Vectors

21.1 LAB: Print 2D array in reverse

Lab activity
21.1.1 LAB: Print 2D array in reverse

22. Additional Labs: User-Defined Functions

22.1 LAB: Toll calculations

Lab activity
22.1.1 LAB: Toll calculations

22.2 LAB: Calculate average

Lab activity
22.2.1 LAB: Calculate average

22.3 LAB: Count evens

Lab activity
22.3.1 LAB: Count evens

22.4 LAB: Reverse vector

Lab activity
22.4.1 LAB: Reverse vector

22.5 LAB: Check if vector is sorted

Lab activity
22.5.1 LAB: Check if vector is sorted

22.6 LAB: Remove all even numbers from a vector

Lab activity
22.6.1 LAB: Remove all even numbers from a vector

22.7 LAB: Fun with characters

Lab activity
22.7.1 LAB: Fun with characters

22.8 LAB: Write Convert() function to cast double to int

Lab activity
22.8.1 LAB: Write Convert() function to cast double to int

23. Additional Labs: Objects and Classes

23.1 LAB: BankAccount class

Lab activity
23.1.1 LAB: BankAccount class

23.2 LAB: Swap two numbers

Lab activity
23.2.1 LAB: Swap two numbers

23.3 LAB: Rolling for a pair

Lab activity
23.3.1 LAB: Rolling for a pair

23.4 LAB: Rolling for X

Lab activity
23.4.1 LAB: Rolling for X

23.5 LAB: Calculator class

Lab activity
23.5.1 LAB: Calculator class

23.6 LAB: Count probations

Lab activity
23.6.1 LAB: Count probations

23.7 LAB: Flipping for heads

Lab activity
23.7.1 LAB: Flipping for heads

23.8 LAB: Simple car

Lab activity
23.8.1 LAB: Simple car

23.9 LAB: Student class

Lab activity
23.9.1 LAB: Student class

23.10 LAB: Product class

Lab activity
23.10.1 LAB: Product class

23.11 LAB: Course size

Lab activity
23.11.1 LAB: Course size

23.12 LAB: Dean's list

Lab activity
23.12.1 LAB: Dean's list

23.13 LAB: How many dice rolls?

Lab activity
23.13.1 LAB: How many dice rolls?

23.14 LAB: Random values

Lab activity
23.14.1 LAB: Random values

23.15 LAB: Drop student

Lab activity
23.15.1 LAB: Drop student

23.16 LAB: Find student with highest GPA

Lab activity
23.16.1 LAB: Find student with highest GPA

23.17 LAB: Print student roster

Lab activity
23.17.1 LAB: Print student roster

23.18 LAB: Flipping for tails

Lab activity
23.18.1 LAB: Flipping for tails

23.19 LAB: Find the maximum in an vector

Lab activity
23.19.1 LAB: Find the maximum in an vector

23.20 LAB: Consecutive heads

Lab activity
23.20.1 LAB: Consecutive heads