Reading & Tutorials Commentary

Introduction

No textbook is perfect! In this document is my commentary on the assigned reading and tutorials for this course.

How to use this document

After reading a section of the on-line text (https://www.learncpp.com/), one of my documents (https://gats.ca/programming/), or completing a tutorial, read the commentary for additional information.

GATS Documents

Why C & C++? [link]

The web is full of opinions about C & C++, and much of it is ill-informed or just flat out wrong! This document attempts to explain why so many programmers continue to be committed to C and C++.

Enabling C++ 20 [link]

How to enable C++ 20 in Visual C++ 2022 solutions.

You can test your installation with this solution: gats.ca/…/TestCpp20.zip

Programming Paradigms [link]

There are many different types of programming languages and ideas about how they should work.

Java, for example, is an imperative language – its code tells the computer what steps it must take to compute the answer. In contrast to Java, HTML is a descriptive markup language – its code describes what the output should look like, but leaves the steps to achieve the result to your web browser.

Learn C++

Chapter 0 – Introduction / Getting Started

0.3 — Introduction to C/C++ [link]

Disagreement: Overall, this is a nice summary, but it suggests that there is a language called C/C++. Stroustrup says no! C and C++ are two separate but highly compatible languages. C++ has a massive amount of language features that don't exist in C. Modern C++ and C# look more alike than modern C and C++. While C++ can almost always compile C code; C has no hope of ever compiling C++ code. $C \subset C++$. $C \neq C++$.

0.4 — Introduction to C++ development [link]

Note: You do not need to return 0 from the main of a standard C++ program. The compiler with generate that code if you don't write it.

The program could have been as simple as:

```
#include <iostream>
int main() {
    std::cout << "Here is some text.\n";
}
```

I've added a \n to the end of the string – a best practice. Without the \n on the last line of output the command line prompt of many operating system shells would appear on the same line.

0.7 — Compiling your first program [link]

Building and executing your first program.

We'll be using the Visual Studio option, but you are free to try the other ones.

Project submissions will require using Visual Studio.

0.8 — A few common C++ problems [link]

Read! But you don't need to apply these fixes if the problem doesn't emerge.

Note: the code to hold your program console open is only useful if you run your program by double-clicking your program from the GUI. Since console programs are normally run from the command prompt, this code just gets in the way. **Never use it in a finished console project!**

0.12 — Configuring your compiler: Choosing a language standard [link]

[**Optional**] Already covered in the GATS document: Enabling C++ 20 [link].

Commentary: C++ 20 will certainly be the dominant language standard by the time you graduate, so prefer that to earlier versions. We will learn some features in this course that were only introduced with C++ 20.

Chapter 1 – Basics

1.1 — Statements and the structure of a program [link]

The string in the example is lacking a \n which most consoles would require to keep the output and the ensuing prompt on separate lines. Both K&R's (the original) and Stroustrup's own example place a newline character at the end of the output string.

I suspect the tutorial author left this out for now so as to avoid the discussion about escape characters until later.

1.2 — Comments [link]

Commentary: There is a cost to commenting – the time the author spends to write and maintain the comments, and the time others spend reading those comments.

Make the comments worth peoples' time to read.

I like to focus on comments that explain why the code that follows is there, not how the code works – the code itself should do that.

1.3 — Introduction to objects and variables [link]

Commentary: While most languages distinguish between *primitive data types* (types directly understood by the CPU) and *non-primitive data types* (types built up from simpler elements), C++ endeavours allow programmers to handle both in exactly the same way. Languages like Java treat primitives and non-primitives with different and often exclusive rules.

1.4 — Variable assignment and initialization [link]

Commentary: It is considered a best practice to initialize a variable when you define that variable. We call this principle RAII – Resource Allocation Is Initialization. So, assign a value to your variable in the same line/statement in which it is defined.

1.6 — Uninitialized variables and undefined behavior [link]

Commentary: The values of uninitialized variables is security risk as they contain the value of whatever that memory had before. Most modern operating systems will 'zero-the-RAM' before allocating that memory to a new application preventing your program from seeing the preceding program's data. Windows is one such system.

Warning: never count on the operating system zeroing your RAM. A lot happens when your program starts up, and memory is allocated and deallocated before your main() is ever called. ***Always initialize your memory yourself!***

1.7 — Keywords and naming identifiers [link]

Commentary: the ISO style guide prefers ***snake_case*** (or as we old timers called it ***saddle_back*** case). However, you should always follow the conventions established by your company (your boss will let you know).

Warning: never use SCREAMING_SNAKE_CASE in C or C++ for constants. It should only be used for preprocessor identifiers (more on this later). And yes, this does conflict with the convention in languages like Java.

1.8 — Whitespace and basic formatting [link]

Horrible Practice: I once wrote a 10,000+ character program in two lines of code – why? To win a bet! It worked but was completely unreadable.

Obscure code: you can break a long string literal into multiple parts simply by adding whitespace between the literals. The compiler will join them back into one literal during preprocessing.

```
cout << "This is a single line of output!\n";

cout << "This is "
      "a single line "
      "of output!\n";
```

To the compiler these two output statements are exactly the same thing.

1.9 — Introduction to literals and operators [link]

Style: don't overdo spacing, studies have shown that too much white-spacing distracts the reader and slows them down.

Tip: use the same spacing guidelines that mathematicians use when writing expressions. Group terms by using tight spacing with multiplies and divides but broad spacing with addition and subtraction.

e.g., `y = m*x + b;`

Warning: PEMDAS/PEDMAS/BEDMAS/etc. only list a subset of operators and **do not** stress that divide and multiple are in the same precedence *group*. **PE(MD)(AS)** would be a little more accurate.

Don't fool yourself the PEMDAS explains it all! What about the precedence of ++, --, <, ==, =, %, ?:, etc.

1.10 — Introduction to expressions [link]

Concept: in compiler theory we talk about **R-values**, and **L-values**. **R-values** are restricted to the *right-hand-side* of an assignment, **L-values** can be on either side of an assignment.

A lone variable is an **L-value**. An expression is typically an **R-value**.

y = m*x + b;

'y' is an **L-value** and 'm*x + b' produces an **R-value**.

m*x + b = y;

violates the **R-value**/**L-value** rule and will not compile.

Chapter 2 – C++ Basics: Functions and Files

2.1 — Introduction to functions [link]

Commentary

*"Functions that you write yourself are called **user-defined functions**."*

This is not a distinction made by C++. All library functions were once user-defined functions. The compiler handles both with exactly the same mechanism.

"Warning: Don't forget to include parentheses () after the function's name when making a function call."

It isn't incorrect to do so, you'll just be getting a different value.

auto x = sqrt(4.0);

This is a call to the sqrt function passing the argument 4.0. The variable 'x' is defined to be of type 'double' and receives the value 2.0.

auto x = sqrt;

This is an assignment of address of the sqrt function. The variable 'x' is defined to be of type double(*)(double) which is a function pointer and receives the memory location of the sqrt function.

Terminology

Every callable thing in C++ is a function.

- a function that may or may not take a parameter and returns a value through its name is called 'function'
  e.g. sin(), pow(), sqrt()
  The name refers to what you get back (the return value), typically using a **noun**.
- a function that may or may not take a parameter and does not return a value through its name is called 'procedure'
  e.g. print, sort, erase
  The name refers to the action performed, typically using a **verb**.
- a function that is in a class/struct is called a 'member function' but everyone calls them methods.

2.2 — Function return values (value-returning functions) [link]

"Functions can only return a single value."

While literally true, we can effectively subvert the rule and efficiently return multiple values.

The standard means to achieve multiple return values is to create a custom class to carry the multiple values back to the caller. We don't like doing this because of the coding overhead and having to introduce a class to the global scope that has no other purpose than to carry more than one value through a function's return statement (i.e., clutter).

The C++ standards committee recognized that an easier mechanism for multiple return values would be welcome and got to work starting with C++ 11. The first move was to include the boost project's tuple class which would add to the already included pair class. Both classes are templates that allow you to bind several data types together without requiring a custom class.

```
std::pair<int, float> p; // binds an int and float into a single variable
std::tuple<int, double, float, char> t; // binds four variables together
```

Retrieving the individual values directly from the pair or tuple is a discussion for another day. Let's instead look at the *structured binding declaration* feature added in C++ 17. It was added to make extraction of individual elements of from structures, classes, arrays, and in this case – pairs and tuples.

Continuing from the above examples:

auto [a, b] = p;

Assigns the integer of the pair to 'a' and the float to 'b'. We can combine these two features to make a simple multiple value return:

```
#include <iostream>
#include <tuple>

std::tuple<int, int> func() {
	return { 1,2 };
}

int main() {
	auto [x, y] = func();
	std::cout << "x = " << x << ", y = " << y << "\n";
}
```

2.3 — Void functions (non-value returning functions) [link]

Style: void functions (also known as *procedures*) are typically named by joining a verb and a noun. The verb indicating what the procedure will do, and the noun indicating what the procedure will do it to. In languages with parameter overloading like (C++ and Java) the noun is often ignored as the parameter would indicate what the procedure is operating on.

example

```
Table t;          // a table object
print_table(t);
print(t);
```

2.4 — Introduction to function parameters and arguments [link]

Style: I typically name parameters to indicate whether they modify the action – changes what the function does – with a name suggesting an adverb. For parameters that represent the object that the function acts upon, I use a noun.

2.5 — Introduction to local scope [link]

Best Practice: there is another great reason to use the best practice "define your local variables as close to their first use as reasonable."

Many studies have shown that as the number of lines between a variable's definition and its first use increases then so does the rate of errors.

Programming trick: a C++ programmer can limit the scope of a variable to much less than function its defined in. How? Just surround the variable with curly brackets. Example:

```
void func() {
```

```
    int x{42};
    {
        int y{43};
        cout << x + y << endl;
    }
    cout << x * y << endl;           // error 'y' is now out of scope
}
```

2.6 — Why functions are useful, and how to use them [link]

What are the purposes of functions?

Most texts only discuss code reuse as the purpose of functions but there are more. I list them as:

1. Code reuse
2. Encapsulating Expertise
3. Higher-order thinking

Code reuse is the most popular but leads to poor design (which inhibits long-term code reuse).

Encapsulating expertise recognizes that functions are usually researched and thoroughly tested by a domain expert. We can 'trust' a function from a library – particularly the standard library.

Higher-order thinking (don't make me think about the details, just the results). A well-chosen function name will document itself and convey the purpose of and effect of a function, hopefully eliminating the need to examine the content of the function to determine its purpose.

Designing functions

Using the purpose listed above, follow the order 3-2-1: high-order thinking + name first, expertise second, then just let code reuse happen.

2.7 — Forward declarations and definitions [link]

Commentary

Make certain that you understand the "one definition rule". It is the source of many errors.

2.8 — Programs with multiple code files [link]

Commentary

Spend a moment to think about a filename that clearly indicates what you will find in that file. Generally speaking, larger projects will require longer more descriptive filenames.

2.9 — Naming collisions and an introduction to namespaces [link]

Commentary

"When C++ was originally designed, all of the identifiers in the C++ standard library (including std::cin and std::cout)…"

Strictly speaking this isn't true. When C++ was originally designed C++ was not a standardized language, nor did it have namespaces (they arrived in the first standard library in 1998). This means that cin and cout, located in the header file <iostream.h>, were not members of any namespace.

*Commentary – Warning against **using namespace std;***

In practice, many programmers violate this warning. It is always a judgement call as to whether you want your code to be clean and thus easier to read, or safer and more compatible with older code. Pre-standard code (before namespace std) is now over 25 years old and should have been updated by now.

My advice… if you are inclined to **using namespace …** then only use one at a time (i.e., only the standard, and have all other namespaces specified at each identifier).

If you have invoked **using namespace std;** and now want to reference a pre-standard global scope you can do so by using the nameless scope prefix.

Example

```
#include <iostream>      // standard
#include <iostream.h>    // pre-standard

using namespace std;
std::cout << "Invoke the standard cout\n";
cout << "Invoke the standard cout\n";          // defaults to std::
::cout << "Invoke the standard cout\n";        // looks in global namespace
```

2.10 — Introduction to the preprocessor [link]

Commentary

Almost every compiler has a preprocessor. What makes C and C++ special is that their preprocessors are programable.

Best practices

Do not use ALL_CAPS for constants in C or C++ but restrict their use to preprocessor macros only. Macro substitutions can occur in the middle of an identifier, so it is best to keep the names of macros completely separate from the names of all other things in C and C++.

Do not use the preprocessor to do anything that can be done within the language. Many of the uses of macros are legacy uses from the C language that didn't (and often still doesn't) have the rich set of tools provided by C++. Macros don't care about parameter types, nor do the optimizing. Only the post-substituted code does.

2.11 — Header files [link]

Best practices

"Prefer a .h suffix when naming your header files" – NO!

C uses this convention, and many header files can be compiled by both C and C++. But many header files can only be compiled by C++. A C++ only header with the .h extension is misleading.

I follow these rules:

- If the header file can be compiled by both C and C++ use .h
- If the header file can only be compiled by C++ use .hpp

Commentary – Key insight…

"Unfortunately, and stupidly, these header files may optionally declare those names in the global namespace as well."

This is wrong, there is no such rule. The reason why the standard uses extensionless header filenames was to stop the standard from instantly breaking pre-standard code when we all upgrade in the late 1990s. We had to maintain much of pre-standard code as we *gradually* transitioned to standard code (we had already written billions of lines of C++ code prior to the 1998 standard).

```
#include <iostream.h>    // for pre-standard ::cout
#include <iostream>      // for post-standard std::cout
```

These two header files could be used simultaneously (although not recommended).

2.12 — Header guards [link]

Commentary

The whole header guard debate (traditional vs #pragma once) will soon be moot as C++ 20/23 is replacing the entire library linking system with **modules**. Modules will provide the benefit of header files and multi-file projects without the strange side effects of multiple references, strange inclusion orders, etc.

The new syntax for including the standard library is:

import std;

This command is the equivalent of including every C++ standard library header file, yet it processes the command in less time than including only <iostream>.

Unfortunately, support for this feature is still migrating into current compilers and IDEs.

Chapter 3 – Debugging C++ Programs

3.6 — Using an integrated debugger: Stepping [link]

Best Practice: Don't write your loops and conditional statements on one line – it makes debugging more difficult.

The compiler has no problem with the following code:

```
for (int c{0}; c < 10; ++c) cout << "c = " << c << "\n";
```

The problem occurs when you try to step through it with your debugger. One step will execute the entire loop!

The conventional style:

```
for (int c{0}; c < 10; ++c)
      cout << "c = " << c << "\n";
```

Allows you to step one line at a time. First the for-loop line, then the output line (the loop body), then back to the for-loop line. This will repeat until the loop has *iterated* 10 times.

3.8 — Using an integrated debugger: Watching variables [link]

Best Practice: make use of limited scope temporary variables.

If the line of code:

```
print(x+y*5);
```

produces an unexpected output you must ask yourself whether the formula or the function is at fault.

While most integrated debuggers can *watch* an expression, the time to set the result can be prohibitive and it can also be difficult to set a conditional breakpoint if there is no variable.

Write the line as:

```
auto res = x + y*5;
print(res);
```

Now we have a variable to watch…

If you don't want the variable to *leak forward* (continuing to exist beyond its use) you can limit the temporary variable's scope by writing the code as:

```
{
    auto res = x + y*5;
```

```
    print(res);
}
```

Chapter 4 – Fundamental Data Types

4.1 – Introduction to fundamental data types [link]

Commentary: The statement, *"The modern standard is that a byte is comprised of 8 sequential bits"* should more likely be, *"The most common definition is that a byte is comprised of 8 sequential bits."*

Both the C and C++ languages define a byte as being able to hold at least 256 different values (necessitating a minimum of 8 bits) and that there be no gaps between two bytes. Therefore, every bit in memory is part of a byte. C and C++ compilers exist that reserve any of 8, 9, 16, 32, or even 36 bits for a single byte.

Bytes is data transmission systems typically contain a start bit, the data bits, one or two stop bits, a possible a parity bit. This means that a single transmitted ASCII character (7 bits) may require bytes of 9 to 11 bits to transmit.

Some CPUs don't have bytes. The CDC 6600 (the first successful supercomputer) had 60-bit machine words (the smallest addressable package of data). On this system characters were stored by packing 10 6-bit characters into a single 60-bit word. Any string operations would have to handle the strings six characters at a time.

Some CPU even had flexible or programmable byte sizes. The PDP-10 had 36-bit words and a byte counter that could be set to whatever the programmer desired. On PDP-10 systems ASCII characters where usually handled as 7-bit value packed 5 to a word (35x7-bits=35-bits leaving 1-bit unused).

There are even 1-bit computers such as the WDR 1 and the Connection Machine (1985) but they are now mostly novelties.

4.3 — Object sizes and the sizeof operator [link]

Style: the chapter uses a C-style formatting in its sizeof examples. Surrounding the sizeof expression with parentheses is a C-style of coding. While parentheses are required when the parameter to sizeof is a type, the parentheses are optional when it is an expression.

C-style:

```
int x{};
std::cout << "x is " << sizeof(x) << " bytes\n";
```

C++-style:

```
int x{};
std::cout << "x is " << sizeof x << " bytes\n";
```

Both are equally correct and generate the same machine code. The C++-style is simply less cluttered and less typing.

Commentary: the text is a bit misleading when it says, "you can also use the sizeof operator on a variable name". The actual rule is, "you can also use the sizeof operator on an expression." For example,

```
sizeof (2 + 3.3)    // yields 8, the size of a double
sizeof 'a'          // yields 1, the size of a char
sizeof -'a'         // yields 4, the size of an int
```

In the first example the parentheses are required to yield the size of result of adding an integer to a double. Without the parentheses,

```
sizeof 2 + 3.3      // yields 7.3, the size of an int + 3.3
```

would yield the size of an int added to the literal value 3.3 as the sizeof operator has a higher precedence than addition and therefore is evaluated before the addition.

Such strange uses for the sizeof operator is most commonly found in low-level programming, where memory would be allocated in bulk by RAM size, and not by element count we typically use on high-level containers.

4.4 — Signed integers [link]

One more warning about using integers – division by zero.

Real numbers can safely be divided by zero. Consider the following code:

```
double zero{};
cout << 4.0 / zero << endl;
cout << 0.0 / zero << endl;
```

It produces the output:

```
Inf
-nan(ind)
```

Which represent infinity and not-a-number. These are not errors, they are *values*. Change the expressions to integers and you have undefined behavior.

```
int zero{};
cout << 4.0 / zero << endl;
cout << 0.0 / zero << endl;
```

In C and in C++, undefined behaviour can lead to a CTD or *crash-to-desktop*.

4.5 — Unsigned integers, and why to avoid them [link]

Commentary: Regarding the author's note about the C++ standard claiming that unsigned operands can never overflow, the standard is correct, and the author is completely wrong.

The C++ standard states that unsigned arithmetic is performed *modulo* 2^n where n is the number of bits in that particular integer. This means that arithmetic on an unsigned char is performed *modulo 256*, and on a 16-bit unsigned short int is performed *modulo 65536*. This guarantees that the results never fall outside the range of the unsigned value, hence, they can never overflow.

I think of overflow as an error state which indicates an illegal or useless result, which is true of signed integers and floating-point arithmetic. Unsigned integers I think of like a clock that rolls over each day starting anew.

Commentary: "Many developers believe that developers should generally avoid unsigned integers."

This statement basically says, "don't use unsigned integers because you probably will get it wrong…". It always bothers me when someone appeals to "avoid the problem" instead of "solve the problem" or "guarantee there is no problem." I'm worried about any code that a programmer writes where the programmer doesn't know whether values will fall outside the range of the variable. "Make the variable bigger, just in case" is a cop-out.

The underlying mathematics will usually tell you whether you are safe to use unsigned values.

My rule of thumb: if the mathematics is defined on the set of integers ($\mathbb{Z}$) use a signed integer. If the mathematics is defined on the set of natural numbers ($\mathbb{N}$) use an unsigned integer. For example, the set of Fibonacci numbers is defined as being natural numbers, so negative values are impossible. The same is true for permutations and combinations – negative values are impossible.

Principle of Programming: don't guess – seek to understand!

4.6 — Fixed-width integers and size_t [link]

Commentary: I agree with pretty much every said except "avoid unsigned". See my comments in 4.5 for an explanation.

Use of the **int_fast#_t** types inform the compiler that the programmer wants to optimize for speed. Use of the **int_least#_t** types inform the compiler that the programmer wants to optimize for size.

4.9 — Boolean values [link]

Commentary: the foundations of systematic logic can be found in the works of Aristotle (384-322 BCE). George Boole's contribution (1815-1864 CE) was to expand the earlier work of Aristotle, De Morgan and other mathematicians and wrap it up into a single formal algebraic mathematical system. I don't intend to diminish Boole's work at all. I have him in my list of 100 greatest geniuses of all time and he did lay the foundations of the information age. I just want to take note that even the product of geniuses stand on the work of earlier… geniuses.

Todo: Chapter 5.1 – 5.8

Chapter 6 – Operators

6.2 — Arithmetic operators [link]

Worst practice: sometimes programmers outthink themselves or just don't think and grab some ready-made code from a website. One such example of bad code that I too often see is this formula for negating a numeric variable:

```
x = x - 2*x;
```

where:

```
x = -x;
```

could have been written.

The first formula generates the machine code:

```
      x = x - 2 * x;
00007FF626105915  mov         eax,dword ptr [x]
00007FF626105918  shl         eax,1
00007FF62610591A  mov         ecx,dword ptr [x]
00007FF62610591D  sub         ecx,eax
00007FF62610591F  mov         eax,ecx
00007FF626105921  mov         dword ptr [x],eax
```

The second formula generates the machine code:

```
      x = -x;
00007FF626105924  mov         eax,dword ptr [x]
00007FF626105927  neg         eax
00007FF626105929  mov         dword ptr [x],eax
```

Half the size, twice the speed!

Ironically, the Microsoft C++ compile when optimizing the code will actually change the first block of code into the second. The compiler analyzes the formula and reduces it to its simplest form before converting to machine code.

Above all, the second solution is simply more clearly understandable than the first.

6.3 — Remainder and Exponentiation [link]

Warning: not every language treat modulo the same way.

C, C++, and Java are consistent in the behaviour of their modulo operators (%). Unfortunately, python is different.

C, C++, and Java use truncation and deliver the answer with the same sign as the left-hand-side operand. Python follows the traditional mathematics rules where modulo arithmetic always returns a non-negative result.

Hence, -6 % 5 evaluates to:

- -1 in C, C++, and Java
- 4 in Python

Similarly, integer division also produces different results. C, C++, and Java round the results towards zero, Python rounds down or 'floors' the result.

Hence:

- In C, C++, and Java, **-6 / 5** evaluates to **-1**.
- In Python, **-6 // 5** evaluates to **-2**.

Todo: Chapter 10

Todo: Chapter 12

Todo 12.1 – 12.9

Todo: Chapter 13

Todo: Chapter 17.7-17.10

Todo: Chapter 19

Chapter 20 – Functions

20.3 — Recursion [link]

Commentary: I am often asked how I choose between recursion and iteration for a particular solution. I'll try to encapsulate my thought process.

1. Collect all the common solutions to the problem (Wikipedia, textbooks, etc.)
    a. If the problem hasn't yet been solved, solve it on *paper* (not literally, but the point is to avoid a programming language because that tends to pick a solution for you – we risk making the solution like our code, and not our code like our solution).
2. Thoroughly understand how the solutions work. You can use physical devices like a deck of cards to help you with searching and sorting algorithms for example. *Paper, whiteboarding* – try to break down the steps to a level that an elementary school child to follow.
3. Pseudo code each solution, or if time permits write a simple implementation of each.
4. Rate the solutions by the following criteria:
    a. Which approach – iterative or recursive – is most like the clearest solution?
    b. How many operations are performed by each solution? Be certain to include the recursive calls in your assessment (they include copying data to the stack, stack safety checks, the call, the return, and finally the stack cleanup – after each call!)
    c. How much memory is used by each solution? The variables: locally declared, dynamic data, and the parameters passed into the function. The overhead of single level function call can often be optimized away, but this is generally not the case with a recursive function call as all the previous local states must be maintained.
    d. What is the maximum number of recursive function-calls before the end condition is reached? The concern is the literal **stack overflow**.

5. Decide based on the following criteria:
    a. If a stack overflow is possible, the recursive solution is a failure. **Use the iterative solution**.
    b. Compare the number of operations to the number of each of solutions and the amount of memory used by each of the solutions. Favour the solution with the most pronounced advantage.
    c. Avoid non-intuitive solutions. If you don't fully understand how a solution works you run the risk of an incorrect implementation (fundamentally flawed or more likely not taking account of boundary or error conditions) which could lead to an error down the road.

Todo: 20.4

Chapter 28 - Input and Output (I/O)

28.1 — Input and output (I/O) streams [link]

Concept: The *stream* concept is often elusive in computing texts and manuals. Let's clarify the concept with examples.

According to Wikipedia, "a stream is a sequence of potentially unlimited data elements made available over time."

What does this mean?

1. A stream contains data elements that are provided one after another over time. (e.g., a keyboard provides characters as the user types).
2. A stream is potentially unlimited (codata). (e.g., we can't know how long a user will type at the keyboard).
3. While not universally true, data elements in a stream are normally accessed in chronological order (e.g., you can not read the fifth character typed until you have read the previous four characters.)

28.2 — Input with istream [link]

C++ 20: Much of the discussion in the chapter has been made moot by C++ 20. In C++ 20, the first code sample:

```
char buf[10]{};
std::cin >> buf;
```

is safe! C++ 20 makes use of developments in template programming and has changed the behavior of reading into an intrinsic array. This code now automatically stops input at nine characters and copies a terminating null into the last position in the array.

The designers of the C++ 20 standard library made this change due to programmers simply ignoring the majority of the safety steps outlined in this chapter.

So, which should I use? In the general case I always start with the std::string input solution.

```
std::string buf;
std::cin >> buf;
```

If I'm reading into a fixed-sized array (when optimizing for size or speed), then I use the first solution.

28.3 — Output with ostream and ios [link]

What about the thousand separators? By default, C++ prints numbers without thousands separators – and this is a good thing!

Why? Because we like output that can be easily digested by other programs. An integer like "123456789" can be directly read by "cin >> n", but "123,456,789" cannot. We'd have to add logic to skip the commas.

But if you want to provide an option for easy-to-read output you can instruct C++ to output numbers with thousands separators by imbuing the output stream with a locale object.

```
#include <iostream>
#include <locale>
int main() {
    int n{};
    std::cin >> n;
    std::cout.imbue(std::locale(""));
    std::cout << "n = " << n << endl;
}
```

28.4 — Stream classes for strings [link]

Commentary: all of the formatting options that can be applied to std::cin and std::cout can also be applied to std::istringstream and std::ostringstream.

If you want to format a number to a string with thousands separators you can imbue the string stream with a locale. For example, the following prints thousands separated real numbers rounded to 3 decimal places.

```
    std::ostringstream formattedNumber;
    formattedNumber << setprecision(3) << fixed;
    formattedNumber.imbue(std::locale(""));
    formattedNumber << "n = " << n;
```

28.5 — Stream states and input validation [link]

Clarification: There is a lot of misunderstanding around the concept of the stream states: fail, and eof. The problem is understanding when they happen and exactly what they mean.

Meaning of fail(): failbit indicates that the last operation on the stream failed to complete. For example:

- Following an element extraction, fail indicates that an element of the expected type was not extracted. The input attempt stops at the first character that would trigger a failure.
- Following an call to .open() on a file stream, fail() would indicate that the file was not opened or created.

Meaning of eof(): eofbit indicates that during the last operation the stream came to an end. This usually occurs when a file-based stream attempts to read past the last character in the file but could also **Note:** eofbit will still be false when the last character has been read. It will only flip to true when we attempt to read another character.

This distinction is important because the read of an individual character will usually trigger both **failbit** and **eofbit** at the same time. However, if you are reading a string and the last data element in the stream is that string, then **failbit** would be **false** (the read of the string succeeded) and **eofbit** would be **true** (the read encountered the end of the stream).

28.6 — Basic file I/O [link]

Commentary: the second program that reads a file and displays the contents of that file to the console has a tiny little issue. It sometimes displays extra newline character at the end of the output.

Why does this happen? When the function std::getline() returns it could be because it acquired a line of text or that it ran out of input, **or** both could be true. The code presented assumes that either one or the other has happened.

A fix to the program that would not introduce a newline at the end if the source file doesn't end in a newline would be:

```
    std::string strInput{};
    while (std::getline(inf, strInput)) {
        std::cout << strInput;
        if (not inf.eof())
            std::cout << '\n';
    }
```

Changelog

Version	Date	Change
0.0.0	2024-01-03	Document creation.
0.1.0	2024-05-05	Added commentary from winter 2024 term
0.2.0	2024-05-08	Added commentary for Chapters 4.1 – 4.11, 6.2 – 6.3
0.3.0	2024-05-14	Added commentary for Chapters 28.1 – 28.6
0.4.0	2024-06-02	Completed commentary for 2.1 through 2.12