Basics of C++

CSI-140 Introduction to Programming

Instructor: Dr. Vikas Thammanna Gowda Semester: Fall 2025

Module Overview

This module introduces the foundational concepts of programming in C++ by focusing on variables, data types, and input/output operations. Students learn how data is represented, stored, and manipulated within a program, and how programs interact with users through input and output. The module emphasizes proper use of identifiers, scope, and constants; explores primitive and derived data types; and demonstrates C++'s stream-based I/O system for reading from and writing to the console. These concepts form the backbone of structured programming and are essential for building more complex logic in later modules.

Learning Objectives

- *Define variables and constants, and apply rules for naming identifiers.*
- *Differentiate between declaration, initialization, and assignment.*
- *Explain scope and lifetime of variables (local vs. global).*
- *Identify and use fundamental C++ data types: int, float, double, char, bool, and string.*
- *Perform type conversion and casting, distinguishing implicit and explicit conversions.*
- *Explain the concept of streams and their role in C++ I/O.*
- *Use the four standard streams (cin, cout, cerr, clog) appropriately.*
- *Perform basic console I/O operations with insertion («) and extraction (»).*
- *Use escape sequences and formatting to produce well-structured output.*
- *Recognize stream states and handle input/output errors effectively.*

Contents

Chapter 1

Variables

1.1 Introduction

Variables are fundamental building blocks in C++ programming. They act as containers to store data values that can be used and manipulated by the program. In simple terms, a variable has a name and holds a value in memory, and this value can change during program execution. Variables enable programmers to refer to data by name, making code easier to read and maintain. In this section, we will explore what variables are, how to declare and initialize them, rules for naming them, their scope and lifetime, and how to create constants.

A variable in C++ is essentially a named storage location in memory for a value. The name given to the variable is called an **identifier**. You can think of it as a box with a label (the variable's name: identifier) that holds some data (the variable's value). The data stored could be a number, a character, a string, etc., depending on the variable's type. By giving the storage a name, we can easily access or change the value later in our code by referring to that name. You must declare a variable (tell the compiler its name and type) before you can use it. The next subsection covers declarations and initialization in detail.

1.2 Rules for naming variables (identifiers):

Choosing good variable names is important for writing clear code. There are strict rules for what constitutes a valid name in C++, and there are also conventional styles that programmers follow to make code more readable.

1. **Start with a letter or underscore. It cannot start with a digit.**:
   - Valid: `_temp`, `varName`, `tax_1`.
   - Invalid: `1_tax` (starts with digit).
2. **After the first character, identifiers contain only letters, digits, or underscores. No spaces or special characters**:
   - Valid: `user1`, `_data_set2`.
   - Invalid: `first name`, `user-name` , `count$`.
3. **C++ is Case-sensitive**: `Data`, `DATA`, and `data` are distinct.
4. **You cannot use C++ reserved keywords as variable names.** (Keywords are inbuilt identifiers): Reserved words such as `if`, `for`, `while`, `double`.
5. **Unlimited length (but keep names reasonable)**.

Note: A name can begin with underscore, but underscore-starting names are sometimes used for special purposes by libraries. As a beginner, it's safest to start your own names with a letter to avoid any unintended conflicts.

1.3 Conventions for naming variables:

Beyond the basic rules, programmers follow naming conventions for clarity:

- Use meaningful names that describe the purpose of the variable. E.g. use `totalScore` instead of `x`, `playerName` instead of `p`.
- In C++, a common style is **camelCase** for variable names. This means if a name is made of multiple words, you start with lowercase and capitalize the first letter of each subsequent word, e.g. `highScore`, `firstName`. Another style is **snake_case**, where words are all lowercase and separated by underscores (e.g. `high_score`). Either style is acceptable; what's important is consistency.
- Typically, constants (which we'll discuss later) are named in all uppercase with underscores, e.g. `MAX_VALUE`. This makes them visually distinct.
- Avoid extremely short or single-letter names except for very trivial uses (like loop counters `i`, `j`). Clear names make your program easier to understand.

Following naming conventions makes your code more readable and maintainable. For example, `int numberOfStudents;` is much clearer than `int nos;` or `int n;`. Good names along with proper formatting help others (and your future self) understand the code's intent quickly.

1.4 Declaration vs. Initialization

Declaration of a variable means introducing the variable's name and type to the program. When you declare a variable, the compiler allocates memory for it. For example:

Example 1:

```
int highScore; // Declaration of an integer variable named highScore
```

Here, `highScore` is declared as an `int` (integer). After this line, the program knows `highScore` exists and is an integer, but at this point `highScore` contains an undefined value (often just whatever random bits were in memory). You cannot reliably use it until you assign it a value.

Initialization means giving a variable an initial value at the time of declaration. For example:

Example 2:

```
int highScore = 100; // Declaration with initialization
```

This both declares `highScore` as an `int` and initializes it to 100 in one step. Initializing a variable is generally considered good practice, because it ensures the variable starts with a known value rather than indeterminate garbage. In the above example, `highScore` is ready to use immediately, and it holds the value 100. It's important to distinguish initialization from simple assignment. Initialization happens only at the moment of creation of the variable (as part of the declaration). Assignment can happen any time after the variable exists. For example:

Example 3:

```
int value_1; // declaration (value_1 has an undefined value now)
value_1 = 5; // assignment (now value_1 is set to 5)
int num = 10; // declaration with initialization (num starts at 10)
num = 15; // assignment (num's value changes from 10 to 15)
```

In this snippet, `value_1` is declared then later given a value, whereas `num` is declared and initialized in one statement. In both cases you end up with variables that have values, but `num` had an initial value immediately, while `value_1` did not get a valid value until the assignment.

If you try to use a variable's value before initializing it, you will get unpredictable results (the value is whatever was already in memory at that location). Always initialize your variables before use to avoid logic errors or "garbage" values. Modern compilers may also warn you if a variable is used without initialization.

Note: You must declare a variable (tell the compiler its name and type) before you can use it.

1.5 Scope & Lifetime (Local vs. Global)

Scope refers to the region of the program where a variable can be accessed by its name. In C++, a variable's scope is usually limited to the block in which it is declared. A **block** is a section of code enclosed in braces `{ }`, such as the body of a function, loop, or conditional. We commonly categorize scope into two main types: *local scope* and *global scope.*

- **Local variables:** If a variable is declared inside a function or a block, it has local scope. This means it is only visible and can only be used within that function or block. For example, variables declared inside `main()` can't be accessed from other functions. Once the program exits that block (for instance, the function returns), the local variable is no longer accessible and its memory may be reclaimed. The lifetime of a local variable (unless it's declared `static`) is just during the execution of that block: it is created when the block is entered and destroyed when the block is exited.
- **Global variables:** If a variable is declared outside any function (typically at the top of the program file, outside of `main` or other functions), it has global scope. A global variable can be accessed from anywhere in the program after its declaration in all functions and blocks that come after its declaration in the code. Its lifetime is the entire duration of the program (it is created when the program starts and released when the program ends).

Example 4:

```
#include <iostream>
using namespace std;
int globalVar = 5; // global variable
int main() {
  int localVar = 10; // local variable (scope is main function only)
  cout << "Global: " << globalVar << "\n";
              // OK: globalVar is accessible anywhere
  cout << "Local: "  << localVar  << "\n";
            // OK: localVar is accessible here in main()

return 0;
}
// cout << localVar; // ERROR: localVar not in scope here (outside main)
```

Output:

```
Global: 5
Local: 10
```

In this code, `globalVar` is declared outside `main`, so it's global. It can be used both inside `main` and even inside other functions (if we had any). `localVar` is declared inside `main`, so it's a local variable. It exists and is usable only within `main`. Trying to use `localVar` in another function or after `main` ends will cause a compile error, because it's out of scope. Essentially, the braces `{ }` of `main` define the region where `localVar` lives.

Lifetime differences: A local variable's lifetime is limited to its scope. In the above example, when `main()` finishes, `localVar` is destroyed and its memory freed. If `main` were called again (in other programs, functions can be called multiple times), a new `localVar` would be created fresh. In contrast, `globalVar` exists for as long as the program runs; its memory is allocated when the program starts (or when the declaration is encountered) and freed when the program exits. This means global variables retain their values across function calls.

Note: There are also static local variables which, though local in scope, have a lifetime that extends across calls-persisting between function invocations. Those are a special case and use the `static` keyword.

While global variables might seem convenient, excessive use of them is discouraged beyond simple programs, because they can be modified from anywhere, making it harder to track changes and debug. Often, it's better to prefer local variables and pass values to functions as needed. But understanding global scope is important, and small programs often use one or two for simplicity.

Finally, **variable shadowing** is a concept related to scope: if you declare a local variable with the same name as a global variable, the local name "shadows" (hides) the global name within that scope. For instance, if we had a global `int value;` and inside a function we declare another `int value;`, then inside that function any use of `value` refers to the local one, not the global. It's usually best to avoid reusing the same name in nested scopes to prevent confusion.

1.5.1 Constants (`const` vs. `#define`)

Often in programs we have values that should not change once set, such as mathematical constants (pi), configuration values, or other fixed parameters. C++ provides two primary ways to define such constant values: using the keyword `const`, or using the preprocessor directive `#define`. Both can prevent the value from being modified during program execution, but they work very differently.

- `const` variables: In C++, you can declare a variable with the `const` qualifier, which makes that variable read-only after initialization.

  ```
  const double PI = 3.14159;
  ```

  Here `PI` is a constant of type `double`. You must initialize a const variable at the point of declaration (since you can't assign to it later). If you try to write `PI = 2.5;` somewhere else, the compiler will error because `PI` is const. A const variable has a type and obeys scope rules like other variables (e.g., it can be global or local, depending on where you declare it). It's also subject to C++ type checking - meaning you can't accidentally assign a wrong type to it or misuse it in a way inconsistent with its type. Const variables are known to the compiler and can be used in expressions, and the compiler can even optimize them out (in many cases treating them as literal values).

- `#define` macros: The directive `#define` comes from the C preprocessor (which C++ also inherits). It essentially tells the preprocessor to do a textual find-and-replace.

`#define PI 3.14159`

This instructs that wherever the sequence PI appears in code (after this line), it should be substituted with 3.14159 before the actual compilation of C++ code begins. `#define` can be used for constants or even for creating macro functions. However, `#define` does not create a typed variable at all - it's just a blind text substitution. There is no type checking (the compiler never officially knows about a type called PI; it just sees 3.14159 in the code wherever PI was used). Also, `#define` substitutions have no scope; by default they have a global effect from the point of definition onward in the file (you can limit them with undefining or scoping include files, but that's more advanced). This means if you `#define MAX 100` in one place, that macro name MAX will be replaced everywhere after that line until the end of the compilation unit (or until undefined). There's a risk of name collisions with macros because they don't respect namespaces or scopes.

Which to use? In modern C++ programming, prefer const (or newer alternatives like constexpr) to define constant values, rather than #define. Const variables are safer and better integrated into the language:

- They have a specific type and the compiler checks that you use them correctly with respect to that type. For example, if you have const int COUNT = 10; and later accidentally try to do something invalid with COUNT (like treating it as a pointer), the compiler will catch it. With a macro, the compiler just sees the replaced text.
- They obey scope rules. You can have a const inside a function that only exists there. A macro, on the other hand, is simply replaced everywhere, which can lead to unintended interactions if not carefully named (e.g., a macro name might inadvertently match a variable name elsewhere).
- They can be inspected in a debugger since they actually exist as variables (at least until optimized away), whereas macros are gone by compile time - the debugger has no knowledge of a macro.

The only advantage of `#define` for simple constants is that it guarantees a constant value at compile time (and can be used in preprocessor conditionals). But in C++, we have constexpr (constant expressions) and also regular const for most needs, which also can be compile-time. In fact, compilers usually treat const variables of primitive types (when initialized with a literal) similarly to macros by substituting their values at compile time, so performance or memory usage is typically identical. Therefore, use `const` for constants. Reserve `#define` for other purposes like include guards or complex macros (and even then, macros should be used sparingly).

Example 5

```
const int MAX_USERS = 50;
int users = MAX_USERS; // OK: using the const value
// MAX_USERS = 51; // ERROR: cannot modify a const #define LIMIT 100
int vals[LIMIT]; // creates an array of 100 ints
// LIMIT = 200; // this wouldn't make sense - LIMIT is not a variable at
   all
```

In this snippet, `MAX_USERS` is a true constant variable (type int), while `LIMIT` is a macro. Note that we can't do `LIMIT = 200;` because LIMIT isn't an actual variable. Also, if we did something like:

Example 5a

```
#define TEN 10
cout << TEN * 2;
```

the preprocessor would replace it as `cout << 10 * 2;` which is fine. But macros can be tricky in complex expressions - e.g., if you had `\#define SQR(x) (x * x)` and then used `SQR(1+2)`, it expands to `(1+2 * 1+2)` which is not what you might expect due to textual substitution (it becomes `1 + 2 * 1 + 2 = 5`, not 9!). Const variables don't have such surprises because they follow normal expression evaluation rules. In summary, use const whenever possible for constants - it's type-safe, scoped, and checked by the compiler. You will see `\#define` in older C/C++ code, but its role for constants has largely been supplanted by const and constexpr in modern C++.

Chapter 2

Data Types

Every variable in C++ has a data type which determines what kind of data it can hold (e.g. integer, floating-point, character, etc.) and how much memory it occupies. Choosing the correct data type for a variable is important for ensuring the variable can represent the required range of values and for using memory efficiently. C++ has several fundamental data types:

- Integers (whole numbers)
- Floating-point numbers (numbers with fractions/decimals)
- Characters (single letters or symbols)
- Booleans (true/false values)
- And more (like more complex types or user-defined types), but we'll focus on the basics above, plus the C++ string type for text.

Below, we discuss each of these common types in turn, as well as how conversions between types work.

2.1 Integer Types (`int`)

An integer type represents whole numbers (no decimal point). The most commonly used integer type in C++ is `int`. An `int` typically is a signed 32-bit number on modern systems, which means it can store values roughly in the range -2 billion to +2 billion (specifically, -2,147,483,648 to 2,147,483,647, if 32-bit). Some systems or compilers might have `int` as 16-bit or 64-bit, but 32-bit is the usual today. In any case, `int` is sufficient for counting and indexing in most programs, unless you need very large numbers. Examples of integer literals in code: `5`, `0`, `-12`, `2023`. By default these are considered `int` if they are in range. You declare integer variables like:

Example 6:

```
int age = 17;
int count; // declared, not yet initialized
count = 42; // later assignment
```

Here, `age` is initialized to 17. `count` is declared without an initial value, then given 42 later. Besides `int`, C++ includes other integer types with different sizes or properties: `short`, `long`, `long long` are integer types with typically shorter or longer ranges (e.g., `short` often 16-bit, `long long` often 64-bit). You can also have `unsigned int` (and unsigned short/long/long long), which can only represent non-negative numbers but roughly doubles the positive range (since it uses the bit that was for sign to extend magnitude). For example, a 32-bit unsigned int can range 0 to 4.29 billion. There are also exact-width integers in C++

(from `<cstdint>` header) like `int32_t`, `int64_t` if you need a specific size. For beginners, `int` is the go-to for whole numbers unless there's a reason to use a different size. Memory and size: An `int` typically takes 4 bytes of memory (on a 32-bit or 64-bit system). Each byte is 8 bits, so 4 bytes = 32 bits, hence the term "32-bit integer." Example use:

Example 7:

```
int a = 10, b = 25;
int sum = a + b; // sum will be 35
std::cout << sum; // outputs 35
```

Arithmetic with ints is done with the usual mathematical operators (+, -, *, /, % for modulus). One thing to watch out for is that if you divide two ints, the result is an int (fractional part is discarded). For example, `7/2` using int arithmetic yields 3 (not 3.5) because it does integer division.

2.2 Floating-Point Types (`float`, `double`)

Floating-point types represent numbers with a decimal fraction (real numbers). In C++ the standard floating-point types are:

- `float` - typically a 32-bit IEEE 754 single-precision floating point. It can represent about 6-7 decimal digits of precision. This means if you have a number like 12345.678, a float can represent it (with some possible tiny rounding).
- `double` - typically a 64-bit double-precision floating point. It can represent about 15 decimal digits of precision, which is much more precise than float. Most calculations that need decimal or fractional parts use `double` by default in C++ because the precision of float might be insufficient for many tasks (to avoid accumulating rounding errors).
- There's also `long double`, which is often 80-bit or more precision depending on the platform, but that's a more specialized type.

Use `float` or `double` when you need to represent numbers that aren't integers, like 3.14 (pi approximations), 0.005, or 123.456. By default, a literal like `3.14` is actually treated as a double in C++. If you explicitly want a float literal, you can write `3.14f` (adding `f` at the end).
Differences between float and double: Double has roughly double the precision of float and usually a larger range of values. Float takes 4 bytes, double takes 8 bytes in memory. Double's increased precision makes it the default choice in many numerical programs. Float might be used in environments where memory is very constrained or where the extra precision is unnecessary (such as certain graphics calculations or large arrays where precision trade-off is acceptable).

Example 8:

```
float price = 10.99f;
double total = 1234567.89;
std::cout << price << "\n";
std::cout << total << "\n";
```

Here `price` is a float initialized with 10.99. (The `f` after the number forces it to type float; otherwise 10.99 would be a double by default that then gets converted to float.) `total` is a double with a larger number. Both can be printed. Keep in mind that floating-point values may not print exactly as you expect if they can't be represented exactly in binary (for instance, 0.1 is an infinitely repeating binary fraction, so it can't be represented with perfect accuracy; the stored value might be 0.10000000000000001 and that

could show up depending on formatting). One must be cautious when comparing floating-point numbers for equality due to possible tiny differences in representation. But that's a more advanced topic. To summarize:

- Use `int` for whole numbers.
- Use `double` for general floating-point calculations.
- Use `float` if you are sure the precision is enough or you have specific needs (like interfacing with a graphics API using floats).

2.3 Character Type (`char`)

The char type is used to store a single character - such as a letter, digit, punctuation mark, or other symbol. A `char` typically is 1 byte in size (8 bits), and can represent numbers 0-255 (if unsigned) or -128 to 127 (if signed), but more importantly it's usually used to represent ASCII characters. Each character has an integer code (for example, 'A' is 65 in ASCII, 'B' is 66, 'a' is 97, etc.). In C++ code, character constants are enclosed in single quotes, e.g. `'A'`, `'x'`, `'?'`. You can also write a char as a numeric value (its code), but that's less common except for special cases. Example:

Example :

```
char initial = 'J';
char punctuation = '!';
std::cout << initial << punctuation; // prints: J!
```

Here `initial` holds the character 'J', and `punctuation` holds '!'. If you output them, you see "J!" as expected. A `char` is actually an integer type under the hood (essentially an 8-bit integer). This means you can do integer operations on chars (though the result will be an int when doing arithmetic). For instance, you can increment a char to get the next character in ASCII:

Example :

```
char letter = `A';
letter++; // letter becomes 'B' (since `A'+1 = `B' in ASCII)
```

You can also cast chars to int to see their ASCII code:

Example :

```
char grade = `B';
std::cout << grade << " has ASCII code " << int(grade) << "\n";
```

This might output:

```
B has ASCII code 66
```

because 'B' corresponds to 66. We used `int(grade)` to explicitly convert the char to its integer code for printing. Keep in mind that char is for single characters. It's not for strings or words (for that, we use arrays of char or, more conveniently, `std::string` which we will discuss shortly). Also note: char can represent characters in the basic ASCII set easily. For characters beyond that (Unicode, international characters), C++11 introduced a larger char type (char16_t, char32_t for UTF-16/UTF-32). But for simplicity, standard `char` deals with the basic character set.

2.4 Boolean Type (`bool`)

The bool type represents a boolean value - either `true` or `false`. This is useful for logical conditions, flags, and anytime you need to express a yes/no, on/off, or true/false state. In C++, bool is a built-in type. A bool occupies at least 1 byte of memory (typically exactly 1 byte on most systems). Even though logically it only needs 1 bit, for alignment reasons it usually uses a byte. Boolean literals in code are written as `true` and `false` (these are keywords). You can assign or initialize bools with those values or with the result of comparisons. Example:

Example :

```
bool isGameOver = false;
bool hasWon = true;
bool isEqual = (5 == 5); // result of expression (5 == 5) which is true
```

Here, `isGameOver` starts false, `hasWon` starts true. The variable `isEqual` will be true because the comparison `5 == 5` is true. You can output bool values with `std::cout`, but by default they will print as 1 for true and 0 for false (because historically they were treated as 1 or 0 in I/O). If you want to print "true"/"false" as words, you can use `std::boolalpha`:

Example :

```
bool loggedIn = false;
std::cout << loggedIn << "\n"; // prints 0
std::cout << std::boolalpha << loggedIn << "\n"; // prints false
```

The second line sets an output format flag to print bools textually. Booleans in C++ are often a result of comparisons (`==`, `<`, `>`, etc.) or logical operations (`&&`, `||` for AND/OR). They are also used in conditions for `if` statements, loops (while, for), etc., to decide program flow. One more thing: In C++, a bool can be implicitly converted to int or other numeric types: `true` becomes 1, and `false` becomes 0. Conversely, integers can convert to bool: 0 becomes false, and any nonzero value becomes true. This is a source of potential bugs (using an int where bool is expected or vice versa), but it's by design (inherited from C). For clarity, stick to bool for boolean logic. The implicit conversions allow things like:

```
if (someInt) { ... }
```

to treat `someInt` as a boolean condition (false if 0, true if nonzero). But as a beginner, it's clearer to use explicit comparisons (e.g. `if (someInt != 0)`).

2.5 String Type (`std::string`)

Technically, `std::string` is not a primitive built-in type; it is a class (in the C++ standard library) that represents a sequence of characters, i.e., a string of text. However, it behaves in many ways like a built-in type for basic usage, and it's the go-to way to handle text in C++.

To use `std::string`, you need to include the `<string>` header (or ¡iostream¿ also indirectly includes ¡string¿ in many implementations), and either use the `std::` prefix or a using declaration for std::string. Often, if you already have `using namespace std;` in a small program, you can write string.

Example :

```
#include <string>
using namespace std;
string greeting = "Hello, world!";
cout << greeting << "\n"; // prints: Hello, world!
```

Here we created a string variable named `greeting` initialized to "Hello, world!". Notice we use double quotes for string literals (as opposed to single quotes for single char). A string value in C++ must be in double quotes.

The string type allows you to do many useful operations, like concatenation (joining strings with `+` or `+=`), finding substrings, getting length (`greeting.size()` or `greeting.length()`), accessing individual characters (with `greeting[index]`), etc.

Example :

```
string firstName = "John";
string lastName = "Doe";
string fullName = firstName + " " + lastName; // concatenation
cout << fullName; // outputs "John Doe"
```

You can also append: `firstName += "ny";` would change "John" into "Johnny". Reading input into strings: One thing to be aware of: using `std::cin >> someString` will only read a word (stops at whitespace). To read a full line of text (which may include spaces), you should use `std::getline` (covered in I/O section).

Example :

```
string line;
getline(cin, line);
```

This will read an entire line (until a newline character) into the string variable `line`. Memory: A string manages a dynamic array of characters internally. You don't generally need to worry about the size because the string class handles growing the storage as needed when you add text. Just be mindful that strings can potentially use a lot of memory if you store a very large text in them, but for normal usage this is not an issue.

In summary, `std::string` is a convenient and powerful way to work with text in C++. It is part of the C++ standard library (in the std namespace), so remember to include the header and use the namespace appropriately. We will see string usage in examples of input/output as well.

2.6 Type Conversion & Casting (Implicit vs. Explicit)

Sometimes you have values of one type and need to use them as another type. Type conversion is the process of converting a value from one data type to another. C++ does some conversions automatically (this is called implicit conversion or coercion), and other times you must explicitly request a conversion (called explicit casting).

Implicit Conversions: These happen automatically by the compiler in certain situations, usually when it can do so without losing information (or in a well-defined manner if losing some precision) Common examples:

- Promoting int to double in an expression. If you do `int a = 5; double d = a;`, the compiler will convert the int 5 to a double 5.0 to store in `d`. No special syntax needed - this is an implicit conversion.

- In expressions, if you mix types, C++ will usually promote the "smaller" type to the larger. For example, in `a + d` where `a` is int and `d` is double, `a` will be converted to double (5 becomes 5.0) and then the addition happens in double. This is to preserve precision.
- Converting from an integer type to a larger integer type (e.g., int to long long) is implicit.
- Converting from float to double is implicit (widening conversion).
- Converting from double to float is also implicit in assignments, though this is narrowing (double to float may lose precision since double is more precise). If you initialize a float with a double literal, you might need a suffix or cast, but in assignment, the compiler will usually allow it with possibly a warning.

However, not all conversions that lose information are implicit: for example, converting from `double` to `int` (which drops the fractional part) will not happen implicitly in an assignment or initialization without a cast in certain contexts (especially if using brace initialization). In C++ old-style assignment would allow it (with a warning), but it's considered a narrowing conversion. So it's best to do that explicitly. Explicit Conversions (Casting): These are conversions you force in code. C++ has several cast operators (like `static_cast`, `dynamic_cast`, etc.) and also inherits C's style cast. The simplest explicit cast is using parentheses like in C: `(type) value`. For example:

Example :

```
double pi = 3.14159;
int approx = (int) pi; // explicit cast to int, approx becomes 3
```

Here, `pi` was 3.14159 (double). Casting to int truncates the decimal, so `approx` gets 3. This is an explicit conversion because we, the programmers, specified it. We told the compiler "convert pi to int here". You should use explicit casts when you want to make it clear that you intentionally convert types, especially if information might be lost. C++-style, one could write `int approx = static_cast<int>(pi);` which does the same thing. `static_cast` is the modern C++ way; it's more verbose but also more precise about the intent and safer in some contexts. Why and when to cast?

- To prevent implicit conversion ambiguities or to pick a specific conversion when multiple are possible.
- To convert to a type where implicit conversion isn't provided. Example: converting a `char` to an `int` to get its ASCII code can be done implicitly, but we often do `int(code)` for clarity. Or converting a pointer to an `int` (special case, needing reinterpret_cast).
- To perform numeric conversions that involve narrowing (like double to int as above, or long to short).
- In general, use casting sparingly and deliberately, because it can bypass type safety.

Example scenario: Consider integer division vs floating-point division:

Example :

```
int a = 10, b = 3;
double c = a / b; // both a and b are int, so integer division happens
double d = (double) a / b;
        // a is explicitly cast to double, so floating-point division
std::cout << c << "\n" << d << "\n";
```

In this snippet:

- In the calculation of `c`, `a/b` are both ints. Integer division 10/3 yields 3 (fractional part discarded), then that 3 is converted to double (becoming 3.0) to store in `c`. So `c` ends up as 3.0.
- In the calculation of `d`, we explicitly cast `a` to double. Now the expression is double divided by int. The compiler will implicitly convert `b` to double as well (3 becomes 3.0) because one operand is already double So it performs floating-point division 10.0/3.0, which yields approximately 3.33333, and that is stored in `d`. Thus `d` $\approx$ 3.33333.

The output of the above would be:

```
3
3.33333
```

This demonstrates how an explicit cast can change the behavior of an operation by affecting the types involved. Implicit vs Explicit summary: Implicit conversions are convenient, but you must be aware of when they happen to avoid surprises. For example, if you assign a larger type to a smaller type:

Example :

```
int big = 1000000;
short s = big; // this might overflow since short may hold less range
```

Many compilers will either throw a warning or implicitly convert and potentially overflow (`s` might end up not 1000000 but some truncated value mod 65536).

An explicit cast here (`short s = (short) big;`) would say "I know I might lose data, do it anyway." In general, prefer explicit casts for narrowing conversions so that it's clear you've acknowledged the loss of data or precision.

Also, note that not all types can convert to each other. For instance, you cannot implicitly convert a `std::string` to an `int`, or an `int` to a `std::string`. Those require more work (like using functions: e.g., `std::stoi` converts string to int, and for int to string use `std::to_string` or streaming). But that's beyond basic implicit/explicit casting rules.

Finally, be careful with casts - they can lead to undefined behavior if used improperly (especially pointer casts). For basic numeric types, static_cast or C-style casts are fine when needed. The rule of thumb is to rely on implicit conversions when they make sense (like promoting int to double), and use explicit casts when you need to force a conversion that the compiler wouldn't normally do or to make the conversion obvious in your code.

Chapter 3

I/O in C++

3.1 Stream Concepts

3.1.1 What is a Stream?

A stream in C++ is fundamentally a sequence of bytes flowing between your program and the outside world. Imagine a stream as a pipe or channel through which data travels. When you type on your keyboard, the characters flow through an input stream into your program. When your program displays text on the screen, the characters flow through an output stream from your program to the display.

The stream concept is one of the most elegant abstractions in programming. It treats all input and output uniformly, whether the data is coming from a keyboard, going to a screen, being read from a file, or being sent over a network. This uniformity means you can learn one set of concepts and apply them to many different situations.

Think of a stream like a conveyor belt in a factory. Items (data) are placed on one end of the belt and travel to the other end. The belt doesn't care what the items are – it just moves them. Similarly, a stream doesn't initially care what kind of data flows through it – it just facilitates the flow. The beauty is that streams are smart enough to handle different types of data appropriately when asked to do so.

3.1.2 Why Streams Matter

Before streams existed, programmers had to write different code for different types of I/O operations. Reading from the keyboard required one set of functions, writing to the screen required another, and file operations needed yet another. Streams unified all these operations under a single, consistent interface.

Streams also provide a buffer between your program and the actual hardware devices. This buffering improves efficiency dramatically. Instead of sending each character immediately to the screen (which would be slow), the stream collects characters in a buffer and sends them in batches. This is like collecting mail throughout the day and making one trip to the post office, rather than going to the post office for each individual letter.

3.1.3 How Streams Work

When you use a stream, several layers of abstraction are working together:

1. The Physical Device: This is the actual hardware – your keyboard, monitor, hard drive, etc.

2. The Operating System: The OS manages the hardware and provides system calls for reading and writing data.

3. The Stream Buffer: This is a memory area that temporarily holds data being transferred. It acts as a shock absorber between your program and the OS.

4. The Stream Object: This is what you interact with in your C++ code. It provides a high-level interface that hides all the complexity below.

When you type a character on your keyboard, it doesn't immediately appear in your program. First, the keyboard sends it to the operating system. The OS places it in a system buffer. When you press Enter, the OS makes that line of input available to your program's input stream. The stream then provides methods for your program to read this data in various formats.

3.1.4 Types of Streams

Input Streams

An input stream brings data into your program. The data flows from an external source (like the keyboard) into your program's variables. The key characteristic of an input stream is that it reads data and converts it from external representation (usually text characters) into internal representation (like integers or floating-point numbers).

When you type "123" on your keyboard, you're actually typing three separate characters: '1', '2', and '3'. Each character has its own ASCII code (49, 50, and 51 respectively). The input stream's job is to recognize that these three characters represent the integer value 123 and perform the conversion automatically.

Output Streams

An output stream takes data from your program and sends it to an external destination. It performs the opposite conversion from input streams: taking internal representations (like the integer 123 stored in binary in memory) and converting them to external representations (the characters '1', '2', '3' that can be displayed on screen).

This conversion is more complex than it might seem. The integer 123 is stored in memory as a binary pattern (01111011 in 8-bit representation). The output stream must convert this binary pattern into the three separate characters that represent "123" in human-readable form.

Input/Output Streams

Some streams can handle both input and output. These bidirectional streams are most commonly used with files, where you might need to both read existing data and write new data. The stream maintains separate positions for reading and writing, allowing sophisticated data manipulation.

3.1.5 Stream States

Every stream maintains information about its current state. This state information tells you whether operations have succeeded or failed, whether you've reached the end of input, or whether an error has occurred. Understanding stream states is crucial for writing robust programs.

A stream can be in one of several states:

Good State: Everything is working correctly. This is the normal, desired state for a stream. All operations have succeeded, and the stream is ready for more I/O operations.

Fail State: An operation failed, but the stream might be recoverable. This often happens when you try to read the wrong type of data (like trying to read "abc" into an integer variable). The stream can often be restored to a good state after clearing the error.

Bad State: A serious error occurred that probably can't be recovered from. This is rare for console I/O but can happen with file I/O if there's a hardware problem.

EOF State: The stream has reached the end of input. For keyboard input, this happens when the user signals end-of-file (Ctrl+D on Unix/Linux/Mac, Ctrl+Z on Windows).

3.2 Standard I/O Streams

3.2.1 The Four Standard Streams

When a C++ program starts, four stream objects are automatically created and connected to standard devices. These streams are global objects that are immediately available for use without any setup or configuration. They provide the primary interface between your program and the user.

3.2.2 cin - The Standard Input Stream

What is cin?

The name "cin" stands for "character input" (pronounced "see-in"). It's an object of type `istream` that's pre-connected to the standard input device, which is typically the keyboard. When your program starts, `cin` is ready and waiting to receive input from whatever source the operating system has designated as standard input.

`cin` is not just a simple pipe for characters. It's an intelligent object that can interpret the characters it receives and convert them into different data types. When you ask `cin` to read an integer, it knows to look for digit characters and convert them into a binary integer value. When you ask it to read a floating-point number, it knows to look for digits, decimal points, and possibly exponential notation.

How cin Processes Input

Understanding how `cin` processes input is essential for using it effectively. When you type on the keyboard, the characters don't go directly to your program. Instead, they go into a buffer managed by the operating system. This buffer holds the characters until you press Enter. Only then does the entire line become available to `cin`.

This line-buffering behavior has important implications. First, it means the user can edit their input (using backspace, for example) before your program sees it. Second, it means that `cin` often has more input available than you're immediately reading, which can lead to unexpected behavior if you're not careful.

When `cin` reads from its buffer, it follows specific rules: - It skips leading whitespace (spaces, tabs, newlines) when reading most data types - It stops reading when it encounters whitespace after finding valid data - It leaves unread characters in the buffer for future input operations

Example

```
#include <iostream>
using namespace std;

int main() {
    // Demonstrating basic cin behavior
    int number;
    char letter;

    cout << "Enter a number: ";
    cin >> number;
    // User types: 42 and presses Enter
    // cin reads 42, leaves newline in buffer

    cout << "You entered: " << number << endl;

    cout << "Enter a letter: ";
    cin >> letter;
    // cin skips the newline and waits for new input

    cout << "You entered: " << letter << endl;

    // Reading multiple values at once
    int x, y, z;
    cout << "Enter three numbers separated by spaces: ";
    cin >> x >> y >> z;
    // User types: 10 20 30 and presses Enter
    // cin reads all three values

    cout << "The numbers are: " << x << ", " << y << ", " << z << endl;

    return 0;
}
```

3.2.3 cout - The Standard Output Stream

What is cout?

The name "cout" stands for "character output" (pronounced "see-out"). It's an object of type `ostream` that's pre-connected to the standard output device, typically the console or terminal window. `cout` is your program's primary way of communicating results to the user.

Like `cin`, `cout` is more than a simple character pipe. It's an intelligent object that knows how to convert various data types into their textual representations. When you send the integer 42 to `cout`, it converts the binary value into the characters '4' and '2' that can be displayed on screen.

How cout Processes Output

`cout` uses a buffer to improve performance. When you send data to `cout`, it doesn't immediately appear on screen. Instead, it goes into a buffer. The buffer is flushed (sent to the screen) when: - The buffer becomes full - You explicitly flush it (using `flush` or `endl`) - The program reads input (automatic flush before input) - The program ends normally

This buffering is usually invisible to you but can be important in certain situations. For example, if your program crashes, unflushed output might be lost.

Example

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    // Basic cout usage
    cout << "Hello, World!" << endl;

    // Outputting different data types
    int count = 42;
    double pi = 3.14159;
    char grade = 'A';
    string message = "C++ Programming";

    cout << "Integer: " << count << endl;
    cout << "Double: " << pi << endl;
    cout << "Character: " << grade << endl;
    cout << "String: " << message << endl;

    // Combining multiple items in one statement
    cout << "The value of pi is approximately " << pi << endl;

    // Using multiple insertion operators
    cout << "Count: " << count << ", Grade: " << grade << endl;

    // Output without endl (no automatic flush)
    cout << "This is on one line. ";
    cout << "This continues the same line." << endl;

    return 0;
}
```

3.3 Console I/O Operations

3.3.1 Basic Input with cin

The Extraction Operator (»)

The extraction operator >> is the primary tool for reading formatted input. The name ”extraction” comes from the idea that you’re extracting data from the input stream. This operator is overloaded for all fundamental data types, which means it knows how to read integers, floating-point numbers, characters, and strings.

The extraction operator is intelligent about whitespace. It automatically skips leading whitespace (spaces, tabs, newlines) before reading data. This is usually helpful but can sometimes cause confusion when you’re trying to read individual characters or when whitespace is significant.

When the extraction operator reads data, it stops at the first character that doesn’t match the expected format. For example, when reading an integer, it stops at the first non-digit character. This character remains in the input buffer and will be the first character seen by the next input operation.

Example

```cpp
#include <iostream>
using namespace std;

int main() {
    // Reading a single integer
    int number;
    cout << "Enter a number: ";
    cin >> number;
    cout << "You entered: " << number << endl;
    cout << "The double of your number is: " << (number * 2) << endl;

    // Reading multiple values
    int first, second;
    cout << "Enter two numbers separated by space: ";
    cin >> first >> second;
    cout << "First: " << first << ", Second: " << second << endl;
    cout << "Sum: " << (first + second) << endl;

    // Reading different types
    int integer_value;
    double decimal_value;
    cout << "Enter an integer and a decimal number: ";
    cin >> integer_value >> decimal_value;
    cout << "Integer: " << integer_value << endl;
    cout << "Decimal: " << decimal_value << endl;

    return 0;
}
```

Reading Characters

Reading characters requires special attention because the extraction operator's whitespace-skipping behavior might not be what you want. When reading a single character with >>, leading whitespace is skipped. If you need to read whitespace characters, you must use different methods.

Example

```
#include <iostream>
using namespace std;

int main() {
    char ch1, ch2, ch3;

    // Using >> skips whitespace
    cout << "Enter three characters (spaces will be skipped): ";
    cin >> ch1 >> ch2 >> ch3;
    // If user types: "a b c", gets 'a', 'b', 'c'
    // If user types: "  a  b  c  ", still gets 'a', 'b', 'c'

    cout << "Characters read: '" << ch1 << "', '"
         << ch2 << "', '" << ch3 << "'" << endl;

    // Reading a character including whitespace
    char ch4;
    cout << "Enter any character (including space): ";
    cin.get(ch4);  // Reads any character, including whitespace
    cout << "Character read: '" << ch4 << "'" << endl;

    return 0;
}
```


Reading Strings

String input has its own complexities. The extraction operator reads a string until it encounters whitespace, which means it only reads single words. This is often not what you want when reading names or phrases.

Example

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string word;
    string line;

    // Reading a single word
    cout << "Enter a single word: ";
    cin >> word;
    cout << "Word read: \"" << word << "\"" << endl;

    // Attempting to read a full name with >>
    string first_name, last_name;
    cout << "Enter your first and last name: ";
    cin >> first_name >> last_name;
    cout << "First name: \"" << first_name << "\"" << endl;
    cout << "Last name: \"" << last_name << "\"" << endl;

    // Reading an entire line (after clearing the buffer)
    cin.ignore();  // Ignore the newline left in buffer
    cout << "Enter a complete sentence: ";
    getline(cin, line);
    cout << "Line read: \"" << line << "\"" << endl;

    return 0;
}
```

3.3.2 Basic Output with cout

The Insertion Operator («)

The insertion operator << sends data to the output stream. The name comes from the idea that you're inserting data into the stream. Like the extraction operator, it's overloaded for all fundamental data types and many library types.

The insertion operator returns a reference to the stream, which allows you to chain multiple insertions together. This chaining is not just convenient; it's also efficient because it avoids the overhead of multiple function calls.

Example

```cpp
#include <iostream>
using namespace std;

int main() {
    // Simple output
    cout << "This is a simple message" << endl;

    // Outputting variables
    int age = 25;
    cout << "Age: " << age << endl;

    // Chaining multiple insertions
    int x = 10, y = 20;
    cout << "x = " << x << " and y = " << y << endl;

    // Outputting expressions
    cout << "x + y = " << (x + y) << endl;
    cout << "x * y = " << (x * y) << endl;

    // Mixing literals and variables
    double price = 19.99;
    int quantity = 3;
    cout << "Price per item: $" << price << endl;
    cout << "Quantity: " << quantity << endl;
    cout << "Total: $" << (price * quantity) << endl;

    return 0;
}
```

Special Characters and Escape Sequences

C++ uses escape sequences to represent special characters that can't be typed directly or that have special meaning. These sequences start with a backslash and are interpreted specially by the compiler.

Example

```cpp
#include <iostream>
using namespace std;

int main() {
    // Newline character
    cout << "First line\nSecond line\nThird line" << endl;

    // Tab character
    cout << "Name\tAge\tCity" << endl;
    cout << "Alice\t30\tBoston" << endl;
    cout << "Bob\t25\tChicago" << endl;

    // Quotation marks
    cout << "She said, \"Hello, World!\"" << endl;

    // Backslash
    cout << "Path: C:\\Users\\Documents\\" << endl;

    // Carriage return (moves cursor to beginning of line)
    cout << "Progress: 100%\rComplete!" << endl;

    // Alert (bell) - may produce a beep
    cout << "Warning!\a" << endl;

    // Using single quotes in character output
    char quote = '\'';
    cout << "Single quote character: " << quote << endl;

    return 0;
}
```