Constants

C++ supports two different kinds of constants:

- **Named constants** are constant values that are associated with an identifier. These are also sometimes called symbolic constants, or occasionally just constants.
- **Literal constants** are constant values that are not associated with an identifier.

Naming your const variables

There are a number of different naming conventions that are used for const variables.

Programmers who have transitioned from C often prefer underscored, upper-case names for const variables (e.g. **EARTH_GRAVITY**). More common in C++ is to use intercapped names with a ‘k’ prefix (e.g. **kEarthGravity**).

Don’t use const when returning by value!!

Prefer constant variables over object-like macros with substitution text.!!

A type qualifier (sometimes called a qualifier for short) is a keyword that is applied to a type that modifies how that type behaves. The const used to declare a constant variable is called a const type qualifier (or const qualifier for short).

As of C++23, C++ only has two type qualifiers: **const** and **volatile**.

Literals

Literal value	Examples	Default literal type	Note
integer value	5, 0, -3	int	
boolean value	true, false	bool	
floating point value	1.2, 0.0, 3.4	double (not float!)	
character	'a', '\n'	char	
C-style string	"Hello, world!"	const char[14]	see C-style string literals section below

Data type	Suffix	Meaning
integral	u or U	unsigned int
integral	l or L	long
integral	ul, uL, Ul, UL, lu, lU, Lu, LU	unsigned long
integral	ll or LL	long long
integral	ull, uLL, Ull, ULL, llu, llU, LLu, LLU	unsigned long long
integral	z or Z	The signed version of std::size_t (C++23)
integral	uz, uZ, Uz, UZ, zu, zU, Zu, ZU	std::size_t (C++23)
floating point	f or F	float
floating point	l or L	long double
string	s	std::string
string	sv	std::string_view

Ranking variables by the likelihood of the compiler being able to optimize them:

- Compile-time constant variables (always eligible to be optimized)
- Runtime constant variables
- Non-const variables (likely optimized in simple cases only)

The conditional operator

? :

Operator	Symbol	Form	Meaning
Conditional	?:	c ? x : y	If conditional `c` is `true` then evaluate `x`, otherwise evaluate `y`

Inline functions and variables

A constexpr function is a function whose return value may be

computed at compile-time.

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

C++20 introduces the keyword **consteval**, which is used to indicate that a function must evaluate at compile-time, otherwise a compile error will result. Such functions are called **immediate functions**.

std::string

To read a full line of input into a string, you're better off using the std::getline() function instead

std::ws

The std::ws input manipulator tells std::cin to **ignore any leading whitespace** before extraction. Leading whitespace is any whitespace character (spaces, tabs, newlines) that occur at the start of the string.

If using **std::getline()** to read strings, use **std::cin >> std::ws** input manipulator to ignore leading whitespace. This needs to be done for each std::getline() call, as std::ws is not preserved across calls.

`std::getline(std::cin >> std::ws, name);`

Do not pass std::string by value, as it makes an expensive copy!! Use std::string_view. Ok to return std::string.

Prefer std::string_view over std::string when you need a read-only string, especially for function parameters.

how to convert std::string_view to std::string

1. Explicitly create a std::string with a std::string_view initializer (which is allowed, since this will rarely be done unintentionally)
2. Convert an existing std::string_view to a std::string using static_cast

Literals for std::string_view

#include
#include // for std::string
#include <string_view> // for std::string_view

#include
#include // for std::string
#include <string_view> // for std::string_view

```
int main()
{
    using namespace std::string_literals;      // access the s suffix
    using namespace std::string_view_literals; // access the sv suffix

    std::cout << "foo\n";   // no suffix is a C-style string literal
    std::cout << "goo\n"s;  // s suffix is a std::string literal
    std::cout << "moo\n"sv; // sv suffix is a std::string_view literal

    return 0;
}
```

- The remove_prefix() member function removes characters from the left side of the view.
- The remove_suffix() member function removes characters from the right side of the view.

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

int main()
{
    std::string_view str{ "Peach" };
    std::cout << str << '\n';

    // Remove 1 character from the left side of the view
    str.remove_prefix(1);
    std::cout << str << '\n';

    // Remove 2 characters from the right side of the view
    str.remove_suffix(2);
    std::cout << str << '\n';

    str = "Peach"; // reset the view
    std::cout << str << '\n';

    return 0;
}
```

A quick guide on when to use std::string vs std::string_view

Use a std::string variable when:

- You need a string that you can modify.
- You need to store user-inputted text.
- You need to store the return value of a function that returns a std::string.

Use a std::string_view variable when:

- You need read-only access to part or all of a string that already exists elsewhere and will not be modified or destroyed before use of the std::string_view is complete.
- You need a symbolic constant for a C-style string.
- You need to continue viewing the return value of a function that returns a C-style string or a non-dangling std::string_view.

Use a std::string function parameter when:

- The function needs to modify the string passed in as an argument without affecting the caller. This is rare.
- You are using a language standard older than C++17.
- You meet the criteria of the pass-by-reference cases covered in lesson 12.5 -- Pass by lvalue reference.
  Use a std::string_view function parameter when:
- The function needs a read-only string.

Use a std::string return type when:

- The return value is a std::string local variable.
- The return value is a function call or operator that returns a std::string by value.
- You meet the criteria of the return-by-reference cases covered in lesson 12.12 -- Return by reference and return by address.
  Use a std::string_view return type when:
- The function returns a C-style string literal or local std::string_view that has been initialized with a C-style string literal.
- The function returns a std::string_view parameter.

Things to remember about std::string:

- Initializing and copying std::string is expensive, so avoid this as much as possible.
- Avoid passing std::string by value, as this makes a copy.
- If possible, avoid creating short-lived std::string objects.
- Modifying a std::string will invalidate any views to that string.

Things to remember about std::string_view:

- std::string_view is typically used for passing string function parameters and returning string literals.
- Because C-style string literals exist for the entire program, it is always okay to set a std::string_view to a C-style string literal.
- When a string is destroyed, all views to that string are invalidated.
- Using an invalidated view (other than using assignment to revalidate the view) will cause undefined behavior.
- A std::string_view may or may not be null-terminated.