# 9.1 — Using a language reference

NASCARDRIVER   AUGUST 30, 2021

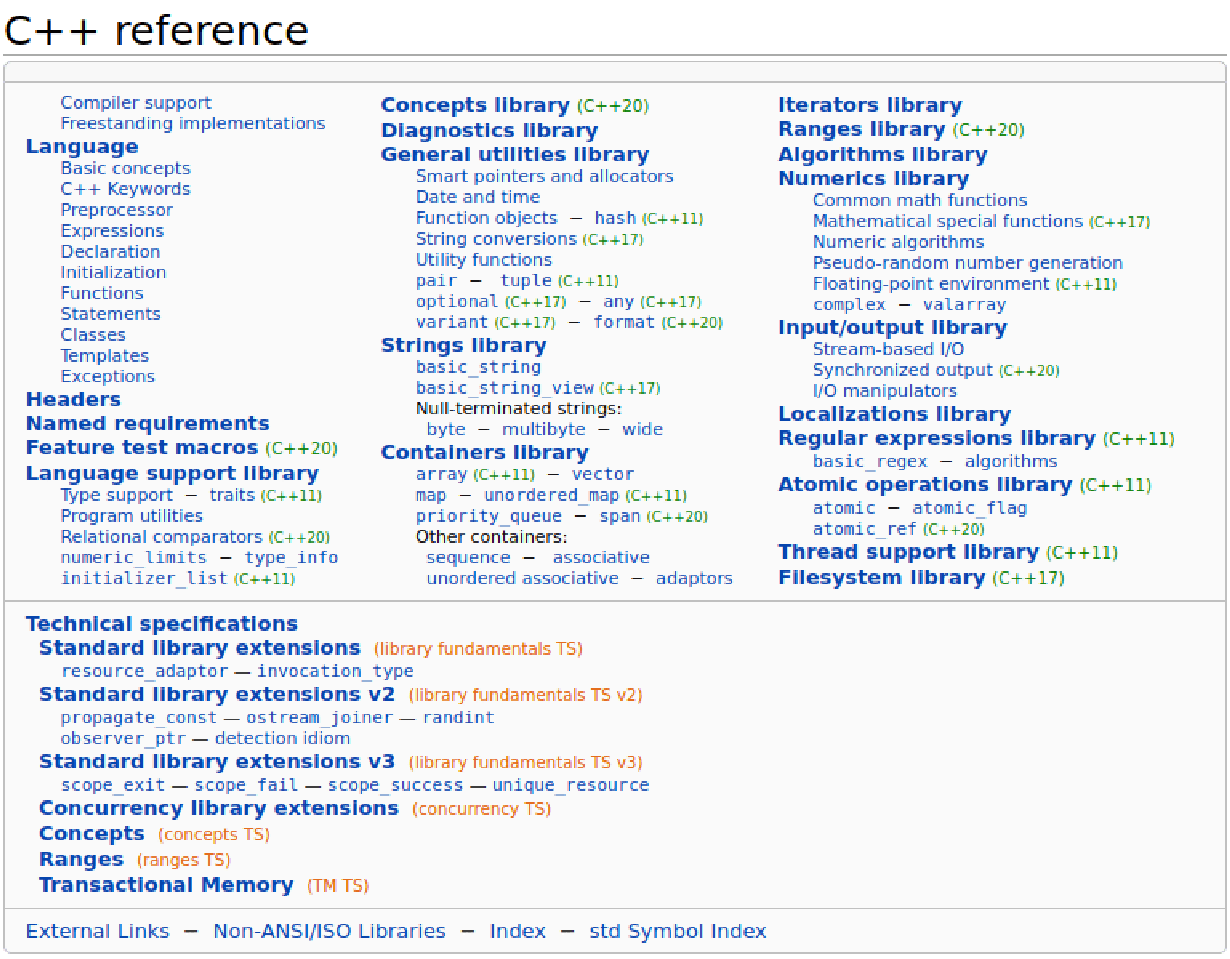
Depending on where you're at in your journey with learning programming languages (and specifically, C++), LearnCpp.com might be the only resource you're using to learn C++ or to look something up. LearnCpp.com is designed to explain concepts in a beginner-friendly fashion, but it simply can't cover every aspect of the language. As you begin to explore outside the topics that these tutorials cover, you'll inevitably run into questions that these tutorials don't answer. In that case, you'll need to leverage outside resources.

One such resource is Stack Overflow, where you can ask questions (or better, read the answer to the same question someone before you asked). But sometimes a better first stop is a reference guide. Unlike tutorials, which are designed to describe part of the language using informal language, reference guides describe the language using formal language. Because of this, reference material tends to be comprehensive, accurate, and… hard to understand.

In this lesson, we'll show how to use cppreference, a popular standard reference that we refer to throughout the lessons, by researching 3 examples.

## Overview

Cppreference greets you with an overview of the core language and libraries:




From here, you can get to everything cppreference has to offer, but it's easier to use the search function, or a search engine. The overview is a great place to visit once you've finished the tutorials on LearnCpp.com, to delve deeper into the libraries, and to see what else the language has to offer that you might not be aware of.

The upper half of the table shows features currently in the language, while the bottom half shows technical specifications, which are features that may or may not be added to C++ in a future version, or have already been partially accepted into the language. This can be useful if you want to see what new capabilities are coming soon.

Starting with C++11, cppreference marks all features with the language standard version they've been added in. The standard version is the little green number you can see next to some of the links in the above image. Features without a version number have been available since C++98/03. The version numbers are not only in the overview, but everywhere on cppreference, letting you know exactly what you can or cannot use in a specific C++ version.

### A reminder

The C++ versions are C++98, C++03, C++11, C++14, C++17, C++20. C++23 is the informal name for the next planned iteration.

### Warning

If you use a search engine and a technical specification has just been accepted into the standard, you might get linked to a technical specification rather than the official reference, which can differ.

### Tip

Cppreference is not only a reference for C++, but also for C. Since C++ shares some functions with C (which can differ), you may find yourself in the C reference after searching for something. The URL and the navigation bar at the top of cppreference always show you if you're browsing the C or C++ reference.

---

## std::string::length

We'll start by researching a function that you know from a previous lesson, `std::string::length`, which returns the length of a string.

On the top right of cppreference, search for "string". Doing so shows a long list of types and functions, of which only the top is relevant for now.

string site:cppreference.com

All Images Videos News Maps Shopping Settings

All regions ▾ Safe search: moderate ▾ Any time ▾

Showing results from: cppreference.com All Results

Strings library - cppreference.com
https://en.cppreference.com/w/cpp/string
The C++ **strings** library includes support for three general types of **strings**: std::basic_string - a templated class designed to manipulate **strings** of any character type.; std::basic_string_view (C++17) - a lightweight non-owning read-only view into a subsequence of a **string**.; Null-terminated **strings** - arrays of characters terminated by a special null character.

std::basic_string - cppreference.com
https://en.cppreference.com/w/cpp/string/basic_string
The class template basic_string stores and manipulates sequences of char-like objects, which are non-array objects of trivial standard-layout type. The class is dependent neither on the character type nor on the nature of operations on that type. The definitions of the operations are supplied via the Traits template parameter - a specialization of std::char_traits or a compatible traits class.

**We could have searched for "string length" right away, but for the purpose of showing as much as possible in this lesson, we're taking the long route. Clicking on "Strings library" takes us to a page talking about the various kinds of strings that C++ supports.**

# Strings library

The C++ strings library includes support for three general types of strings:

- std::basic_string - a templated class designed to manipulate strings of any character type.
- std::basic_string_view (C++17) - a lightweight non-owning read-only view into a subsequence of a string.
- Null-terminated strings - arrays of characters terminated by a special *null* character.

## std::basic_string

The templated class std::basic_string generalizes how sequences of characters are manipulated and stored. String creation, manipulation, and destruction are all handled by a convenient set of class methods and related functions.

Several specializations of std::basic_string are provided for commonly-used types:

Defined in header <string>

| Type | Definition |
|---|---|
| std::string | std::basic_string<char> |
| std::wstring | std::basic_string<wchar_t> |
| std::u8string (since C++20) | std::basic_string<char8_t> |
| std::u16string (since C++11) | std::basic_string<char16_t> |
| std::u32string (since C++11) | std::basic_string<char32_t> |

**If we look under the "std::basic_string" section, we can see a list of typedefs, and within that list is std::string.**

**Clicking on "std::string" leads to the page for std::basic_string. There is no page for std::string, because std::string is a typedef for std::basic_string<char>, which again can be seen in the typedef list:**

| Type | Definition |
|---|---|
| **std::string** | std::basic_string<char> |
| **std::wstring** | std::basic_string<wchar_t> |
| std::u8string (C++20) | std::basic_string<char8_t> |
| **std::u16string** (C++11) | std::basic_string<char16_t> |
| **std::u32string** (C++11) | std::basic_string<char32_t> |
| **std::pmr::string** (C++17) | std::pmr::basic_string<char> |
| **std::pmr::wstring** (C++17) | std::pmr::basic_string<wchar_t> |
| std::pmr::u8string (C++20) | std::pmr::basic_string<char8_t> |
| **std::pmr::u16string** (C++17) | std::pmr::basic_string<char16_t> |
| **std::pmr::u32string** (C++17) | std::pmr::basic_string<char32_t> |

The `<char>` means that each character of the string is of type `char`. You'll note that C++ offers other strings that use different character types. These can be useful when using Unicode instead of ASCII.

Further down the same page, there's a list of member functions (the behaviors that a type has). If you want to know what you can do with a type, this list is very convenient. In this list, you'll find a row for `length` (and `size`).

Following the link brings us to the detailed function description of `length` and `size`, which both do the same thing.

The top of each page starts with a short summary of the feature and syntax, overloads, or declarations:

## std::basic_string<CharT,Traits,Allocator>::size, std::basic_string<CharT,Traits,Allocator>::length

| | |
|---|---|
| `size_type size() const;` | (until C++11) |
| `size_type size() const noexcept;` | (since C++11) |
| `size_type length() const;` | (until C++11) |
| `size_type length() const noexcept;` | (since C++11) |

Returns the number of `CharT` elements in the string, i.e. `std::distance(begin(), end())`.

The title of the page shows the name of the class and function with all template parameters. We can ignore this part. Below the title, we see all of the different function overloads (different versions of the function that share the same name) and which language standard they apply to.

Below that, we can see the parameters that the function takes, and what the return value means.

Because `std::string::length` is a simple function, there's not a lot of content on this page. Many pages show example uses of the feature they're documenting, as does this one:

### Example

Run this code

```
#include <cassert>
#include <iterator>
#include <string>

int main()
{
    std::string s("Exemplar");
    assert(8 == s.size());
    assert(s.size() == s.length());
    assert(s.size() == static_cast<std::string::size_type>(
        std::distance(s.begin(), s.end())));

    std::u32string a(U"ハロー・ワールド"); // 8 code points
    assert(8 == a.size()); // 8 code units in UTF-32

    std::u16string b(u"ハロー・ワールド"); // 8 code points
    assert(8 == b.size()); // 8 code units in UTF-16

    std::string c(u8"ハロー・ワールド"); // 8 code points
    assert(24 == c.size()); // 24 code units in UTF-8
}
```

Until you're done learning C++, there will be features in the examples that you haven't seen before. If there are enough examples, you're probably able to understand a sufficient amount of it to get an idea of how the function is used and what it does. If the example is too complicated, you can search for an example somewhere else or read the reference of the parts you don't understand (you can click on functions and types in the examples to see what they do).

**Now we know what `std::string::length` does, but we knew that before. Let's have a look at something new!**

---

## std::cin.ignore

**In lesson 4.12 -- An introduction to std::string, we talked about `std::cin.ignore`, which is used to ignore everything up to a line break. One of the parameters of this function is some long and verbose value. What was that again? Can't you just use a big number? What does this argument do anyway? Let's figure it out!**

**Typing "std::cin.ignore" into the cppreference search yields the following results:**

std::cin, std::wcin - cppreference.com
https://en.cppreference.com/w/cpp/io/cin
The global objects std::cin and std::wcin control input from a stream buffer of implementation-defined type (derived from std::streambuf), associated with the standard C input stream stdin.. These objects are guaranteed to be initialized during or before the first time an object of type std::ios_base::Init is constructed and are available for use in the constructors and destructors of static ...

std::basic_istream<CharT,Traits>::ignore - cppreference.com
https://en.cppreference.com/w/cpp/io/basic_istream/ignore
Extracts and discards characters from the input stream until and including delim.. ignore behaves as an UnformattedInputFunction.After constructing and checking the sentry object, it extracts characters from the stream and discards them until any of the following conditions occurs:

std::ignore - cppreference.com
https://en.cppreference.com/w/cpp/utility/tuple/ignore
An object of unspecified type such that any value can be assigned to it with no effect. Intended for use with std::tie when unpacking a std::tuple, as a placeholder for the arguments that are not used. [] Possible implementatio

std::basic_istream - cppreference.com
https://en.cppreference.com/w/cpp/io/basic_istream
The class template basic_istream provides support for high level input operations on character streams. The supported operations include formatted input (e.g. integer values or whitespace-separated characters and characters strings) and unformatted input (e.g. raw characters and character arrays).

- `std::cin, std::wcin` **- We want `.ignore`, not plain `std::cin`.**
- `std::basic_istream<CharT,Traits>::ignore` **- Eew, what is this? Let's skip for now.**
- `std::ignore` **- No, that's not it.**
- `std::basic_istream` **- That's not it either.**

**It's not there, what now? Let's go to `std::cin` and work our way from there. There's nothing immediately obvious on that page. On the top, we can see the declaration of `std::cin` and `std::wcin`, and it tells us which header we need to include to use `std::cin`:**

std::cin, std::wcin

Defined in header <iostream>

| | |
|---|---|
| `extern std::istream cin;` | (1) |
| `extern std::wistream wcin;` | (2) |

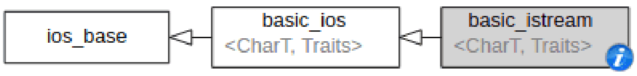
**We can see that `std::cin` is an object of type `std::istream`. Let's follow the link to `std::istream`:**

# std::basic_istream

Defined in header <istream>

```
template<
    class CharT,
    class Traits = std::char_traits<CharT>
> class basic_istream : virtual public std::basic_ios<CharT, Traits>
```

The class template `basic_istream` provides support for high level input operations on character streams. The supported operations include formatted input (e.g. integer values or whitespace-separated characters and characters strings) and unformatted input (e.g. raw characters and character arrays). This functionality is implemented in terms of the interface provided by the underlying `basic_streambuf` class, accessed through the `basic_ios` base class. The only non-inherited data member of `basic_istream`, in most implementations, is the value returned by `basic_istream::gcount()`.



Inheritance diagram

Two specializations for common character types are defined:

Defined in header <istream>

| Type | Definition |
|---|---|
| `istream` | `basic_istream<char>` |
| `wistream` | `basic_istream<wchar_t>` |

## Global objects

Two global basic_istream objects are provided by the standard library.

Defined in header <iostream>

`cin` reads from the standard C input stream `stdin`
`wcin` (global object)

**Hold up! We've seen `std::basic_istream` before when we searched for "std::cin.ignore" in our search engine. It turns out that `istream` is a typedef for `basic_istream`, so maybe our search wasn't so wrong after all.**

**Scrolling down on that page, we're greeted with familiar functions:**

## Member functions

| | |
|---|---|
| (constructor) | constructs the object<br>(public member function) |
| (destructor) [virtual] | destructs the object<br>(virtual public member function) |
| **operator=** (C++11) | move-assigns from another `basic_istream`<br>(protected member function) |
| **Formatted input** | |
| **operator>>** | extracts formatted data<br>(public member function) |
| **Unformatted input** | |
| **get** | extracts characters<br>(public member function) |
| **peek** | reads the next character without extracting it<br>(public member function) |
| **unget** | unextracts a character<br>(public member function) |
| **putback** | puts character into input stream<br>(public member function) |
| **getline** | extracts characters until the given character is found<br>(public member function) |
| **ignore** | extracts and discards characters until the given character is found<br>(public member function) |
| **read** | extracts blocks of characters<br>(public member function) |
| **readsome** | extracts already available blocks of characters<br>(public member function) |
| **gcount** | returns number of characters extracted by last unformatted input operation<br>(public member function) |
| **Positioning** | |
| **tellg** | returns the input position indicator<br>(public member function) |
| **seekg** | sets the input position indicator<br>(public member function) |
| **Miscellaneous** | |
| **sync** | synchronizes with the underlying storage device<br>(public member function) |
| **swap** (C++11) | swaps stream objects, except for the associated buffer<br>(protected member function) |

## Member classes

| | |
|---|---|
| **sentry** | implements basic logic for preparation of the stream for input operations<br>(public member class) |

## Non-member functions

| | |
|---|---|
| **operator>>** (std::basic_istream) | extracts characters and character arrays<br>(function template) |

**We've used many of these functions already: `operator>>`, `get`, `getline`, `ignore`. Scroll around on that page to get an idea of what else there is in `std::cin`. Then click `ignore`, since that's what we're interested in.**

# std::basic_istream<CharT,Traits>::ignore

```
basic_istream& ignore( std::streamsize count = 1, int_type delim = Traits::eof() );
```

Extracts and discards characters from the input stream until and including `delim`.

`ignore` behaves as an *UnformattedInputFunction*. After constructing and checking the sentry object, it extracts characters from the stream and discards them until any one of the following conditions occurs:

- `count` characters were extracted. This test is disabled in the special case when `count` equals `std::numeric_limits<std::streamsize>::max()`
- end of file conditions occurs in the input sequence, in which case the function calls `setstate(eofbit)`
- the next available character `c` in the input sequence is `delim`, as determined by `Traits::eq_int_type(Traits::to_int_type(c), delim)`. The delimiter character is extracted and discarded. This test is disabled if `delim` is `Traits::eof()`

**On the top of the page there's the function signature and a description of what the function and its two parameters do. The `=` signs after the parameters indicate a default argument (we cover this in lesson 8.12 -- Default arguments). If we don't provide an argument for a parameter that has a default value, the default value is used.**

**The first bullet point answers all of our questions. We can see that `std::numeric_limits<std::streamsize>::max()` has special meaning to `std::cin.ignore`, in that it disables the character count check. This means `std::cin.ignore` will continue ignoring characters until it finds the delimiter, or until it runs out of characters to look at.**

**Many times, you don't need to read the entire description of a function if you already know it but forgot what the parameters or return value mean. In such situations, reading the parameter or return value description suffices.**

### Parameters

**`count`** - number of characters to extract
**`delim`** - delimiting character to stop the extraction at. It is also extracted.

### Return value

`*this`

**The parameter description is brief. It doesn't contain the special handling of `std::numeric_limits<std::streamsize>::max()` or the other stop conditions, but serves as a good reminder.**

---

## A language grammar example

**Alongside the standard library, cppreference also documents the language grammar. Here's a valid program:**

```
#include <iostream>

int getUserInput()
{
  int i{};
  std::cin >> i;
  return i;
}

int main()
{
  std::cout << "How many bananas did you eat today? \n";

  if (int iBananasEaten{ getUserInput() }; iBananasEaten <=
2)
  {
    std::cout << "Yummy\n";
  }
  else
  {
    std::cout << iBananasEaten << " is a lot!\n";
  }

  return 0;
}
```

**Why is there a variable definition inside the condition of the `if-statement`? Let's use cppreference to figure out what it does by searching for "cppreference if statement" in our favorite search engine. Doing so leads us to if statements. At the top, there's a syntax reference.**

## Syntax

| | |
|---|---|
| *attr*(optional) **if** **(** *condition* **)** *statement-true* | (until C++17) |
| *attr*(optional) **if** **(** *condition* **)** *statement-true* **else** *statement-false* | (until C++17) |
| *attr*(optional) **if constexpr**(optional) **(** *init-statement*(optional) *condition* **)** *statement-true* | (since C++17) |
| *attr*(optional) **if constexpr**(optional) **(** *init-statement*(optional) *condition* **)** *statement-true* **else** *statement-false* | (since C++17) |

*attr*(C++11) - any number of attributes

*condition* - one of

- expression which is contextually convertible to bool
- declaration of a single non-array variable with a brace-or-equals initializer.

*init-statement*(C++17) - either

- an expression statement (which may be a *null statement* "`;`")
- a simple declaration, typically a declaration of a variable with initializer, but it may declare arbitrary many variables or be a decomposition declaration

  Note that any *init-statement* must end with a semicolon `;`, which is why it is often described informally as an expression or a declaration followed by a semicolon.

*statement-true* - any statement (often a compound statement), which is executed if *condition* evaluates to `true`

*statement-false* - any statement (often a compound statement), which is executed if *condition* evaluates to `false`

**On the right, we can again see the version for which this syntax is relevant. Look at the version of the `if-statement` that is relevant since C++17. If you remove all of the optional parts, you get an `if-statement` that you already know. Before the `condition`, there's an optional `init-statement`, that looks like what's happening in the code above.**

```
if ( init-statement condition ) statement-true
if ( init-statement condition ) statement-true else statement-false
```

**Below the syntax reference, there's an explanation of each part of the syntax, including the `init-statement`. It says that the `init-statement` is typically a declaration of a variable with an initializer.**

**Following the syntax is an explanation of `if-statements` and simple examples:**

## Explanation

If the *condition* yields `true` after conversion to `bool`, *statement-true* is executed.

If the else part of the if statement is present and *condition* yields `false` after conversion to `bool`, *statement-false* is executed.

In the second form of if statement (the one including else), if *statement-true* is also an if statement then that inner if statement must contain an else part as well (in other words, in nested if-statements, the else is associated with the closest if that doesn't have an else)

Run this code

```
#include <iostream>

int main() {
    // simple if-statement with an else clause
    int i = 2;
    if (i > 2) {
        std::cout << i << " is greater than 2\n";
    } else {
        std::cout << i << " is not greater than 2\n";
    }

    // nested if-statement
    int j = 1;
    if (i > 1)
        if (j > 2)
            std::cout << i << " > 1 and " << j << " > 2\n";
        else // this else is part of if (j > 2), not of if (i > 1)
            std::cout << i << " > 1 and " << j << " <= 2\n";

   // declarations can be used as conditions with dynamic_cast
   struct Base {
        virtual ~Base() {}
   };
   struct Derived : Base {
       void df() { std::cout << "df()\n"; }
   };
   Base* bp1 = new Base;
   Base* bp2 = new Derived;

   if (Derived* p = dynamic_cast<Derived*>(bp1)) // cast fails, returns nullptr
       p->df();  // not executed

   if (auto p = dynamic_cast<Derived*>(bp2)) // cast succeeds
       p->df();  // executed
}
```

Output:

```
2 is not greater than 2
2 > 1 and 1 <= 2
df()
```

**We already know how `if-statements` work, and the examples don't include an `init-statement`, so we scroll down a little to find a**

section dedicated to `if-statements` with initializers:

### If Statements with Initializer

If *init-statement* is used, the if statement is equivalent to

```
{
    init_statement
    if constexpr(optional) ( condition )
        statement-true
}
```

or

```
{
    init_statement
    if constexpr(optional) ( condition )
        statement-true
    else
        statement-false
}
```

(since C++17)

Except that names declared by the *init-statement* (if *init-statement* is a declaration) and names declared by *condition* (if condition is a declaration) are in the same scope, which is also the scope of both *statements*.

```
std::map<int, std::string> m;
std::mutex mx;
extern bool shared_flag; // guarded by mx
int demo() {
   if (auto it = m.find(10); it != m.end()) { return it->second.size(); }
   if (char buf[10]; std::fgets(buf, 10, stdin)) { m[0] += buf; }
   if (std::lock_guard lock(mx); shared_flag) { unsafe_ping(); shared_flag = false; }
   if (int s; int count = ReadBytesWithSignal(&s)) { publish(count); raise(s); }
   if (auto keywords = {"if", "for", "while"};
       std::any_of(keywords.begin(), keywords.end(),
                   [&s](const char* kw) { return s == kw; })) {
     std::cerr << "Token must not be a keyword\n";
   }
 }
```

**First, it is shown how the `init-statement` can be written without actually using an `init-statement`. Now we know what the code in question is doing. It's a normal variable declaration, just merged into the `if-statement`.**

**The sentence after that is interesting, because it lets us know that the names from the `init-statement` are available in *both* statements (`statement-true` and `statement-false`). This may be surprising, since you might otherwise assume the variable is only available in the `statement-true`.**

**The `init-statement` examples use features and types that we haven't covered yet. You don't have to understand everything you see to understand how the `init-statement` works. Let's skip everything that's too confusing until we find something we can work with:**

```
// Iterators, we don't know them. Skip.
if (auto it = m.find(10); it != m.end()) { return it->second.size(); }

// [10], what's that? Skip.
if (char buf[10]; std::fgets(buf, 10, stdin)) { m[0] += buf; }

// std::lock_guard, we don't know that, but it's some type. We know what types
are!
if (std::lock_guard lock(mx); shared_flag) { unsafe_ping(); shared_flag = false;
}

// This is easy, that's an int!
if (int s; int count = ReadBytesWithSignal(&s)) { publish(count); raise(s); }

// Whew, no thanks!
if (auto keywords = {"if", "for", "while"};
    std::any_of(keywords.begin(), keywords.end(),
                [&s](const char* kw) { return s == kw; })) {
  std::cerr << "Token must not be a keyword\n";
}
```

**The easiest example seems to be the one with an `int`. Then we look after the semicolon and there's another definition, odd... Let's go back to the `std::lock_guard` example.**

```
if (std::lock_guard lock(mx);
shared_flag)
{
  unsafe_ping();
  shared_flag = false;
}
```

From this, it's relatively easy to see how an `init-statement` works. Define some variable (`lock`), then a semicolon, then the condition. That's exactly what happened in our example.

## A warning about the accuracy of cppreference

Cppreference is not an official documentation source -- rather, it is a wiki. With wikis, anyone can add and modify content -- the content is sourced from the community. Although this means that it's easy for someone to add wrong information, that misinformation is typically quickly caught and removed, making cppreference a reliable source.

The only official source for C++ isthe standard (Free drafts on github), which is a formal document and not easily usable as a reference.

## Quiz time

### Question #1

What does the following program print? Don't run it, use a reference to figure out what `erase` does.

```
#include <iostream>
#include <string>

int main()
{
  std::string str{ "The rice is cooking"
};

  str.erase(4, 11);

  std::cout << str << '\n';

  return 0;
}
```

> **Tip**
>
> When you find `erase` on cppreference, you can ignore the function signatures (2) and (3).

> **Tip**
>
> Indexes in C++ start at 0. The character at index 0 in the string "House" is 'H', at 1 it's 'o', and so on.

Show Solution

### Question #2

In the following code, modify `str` so that its value is "I saw a blue car yesterday" without repeating the string. ie. don't do this:

```
str = "I saw a blue car
yesterday.";
```

You only need to call one function to *replace* "red" with "blue".

```cpp
#include <iostream>
#include <string>

int main()
{
  std::string str{ "I saw a red car yesterday." };

  // ...

  std::cout << str << '\n'; // I saw a blue car yesterday.

  return 0;
}
```

Show Hint

Show Hint

Show Hint

Show Hint

Show Solution


Next lesson

9.2 Enumerated types

Back to table of contents

Previous lesson

8.x Chapter 8 summary and quiz

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ cod

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies:

POST COMMENT

DP N N FOUT

Newest



X

X