1.7 — Keywords and naming identifiers

learncpp.com/cpp-tutorial/keywords-and-naming-identifiers/

Keywords

C++ reserves a set of 92 words (as of C++23) for its own use. These words are called **keywords** (or reserved words), and each of these keywords has a special meaning within the C++ language.

Here is a list of all the C++ keywords (through C++23):

- alignas
- alignof
- and
- and_eq
- asm
- auto
- bitand
- bitor
- bool
- break
- case
- catch
- char
- char8_t (since C++20)
- char16_t
- char32_t
- class
- compl
- concept (since C++20)
- const
- consteval (since C++20)
- constexpr
- constinit (since C++20)
- const_cast
- continue
- co_await (since C++20)
- co_return (since C++20)
- co_yield (since C++20)
- decltype
- default

- delete
- do
- double
- dynamic_cast
- else
- enum
- explicit
- export
- extern
- false
- float
- for
- friend
- goto
- if
- inline
- int
- long
- mutable
- namespace
- new
- noexcept
- not
- not_eq
- nullptr
- operator
- or
- or_eq
- private
- protected
- public
- register
- reinterpret_cast
- requires (since C++20)
- return
- short
- signed
- sizeof
- static
- static_assert
- static_cast
- struct

- switch
- template
- this
- thread_local
- throw
- true
- try
- typedef
- typeid
- typename
- union
- unsigned
- using
- virtual
- void
- volatile
- wchar_t
- while
- xor
- xor_eq

The keywords marked (C++20) were added in C++20. If your compiler is not C++20 compliant (or does have C++20 functionality, but it’s turned off by default), these keywords may not be functional.

C++ also defines special identifiers: *override*, *final*, *import*, and *module*. These have a specific meaning when used in certain contexts but are not reserved otherwise.

You have already run across some of these keywords, including *int* and *return*. Along with a set of operators, these keywords and special identifiers define the entire language of C++ (preprocessor commands excluded). Because keywords and special identifiers have special meaning, your IDEs will likely change the text color of these words to make them stand out from other identifiers.

By the time you are done with this tutorial series, you will understand what almost all of these words do!

Identifier naming rules

As a reminder, the name of a variable (or function, type, or other kind of item) is called an identifier. C++ gives you a lot of flexibility to name identifiers as you wish. However, there are a few rules that must be followed when naming identifiers:

- The identifier can not be a keyword. Keywords are reserved.

- The identifier can only be composed of letters (lower or upper case), numbers, and the underscore character. That means the name can not contain symbols (except the underscore) nor whitespace (spaces or tabs).
- The identifier must begin with a letter (lower or upper case) or an underscore. It can not start with a number.
- C++ is case sensitive, and thus distinguishes between lower and upper case letters. `nvalue` is different than `nValue` is different than `NVALUE`.

Identifier naming best practices

Now that you know how you *can* name a variable, let's talk about how you *should* name a variable (or function).

First, it is a convention in C++ that variable names should begin with a lowercase letter. If the variable name is a single word or acronym, the whole thing should be written in lowercase letters.

```
int value; // conventional

int Value; // unconventional (should start with lower case letter)
int VALUE; // unconventional (should start with lower case letter and be in all lower
case)
int VaLuE; // unconventional (see your psychiatrist) ;)
```

Most often, function names are also started with a lowercase letter (though there's some disagreement on this point). We'll follow this convention, since function *main* (which all programs must have) starts with a lowercase letter, as do all of the functions in the C++ standard library.

Identifier names that start with a capital letter are typically used for user-defined types (such as structs, classes, and enumerations, all of which we will cover later).

If the variable or function name is multi-word, there are two common conventions: words separated by underscores (sometimes called snake_case), or intercapped (sometimes called camelCase, since the capital letters stick up like the humps on a camel).

```
int my_variable_name;   // conventional (separated by underscores/snake_case)
int my_function_name(); // conventional (separated by underscores/snake_case)

int myVariableName;     // conventional (intercapped/camelCase)
int myFunctionName();   // conventional (intercapped/camelCase)

int my variable name;   // invalid (whitespace not allowed)
int my function name(); // invalid (whitespace not allowed)

int MyVariableName;     // unconventional (should start with lower case letter)
int MyFunctionName();   // unconventional (should start with lower case letter)
```

In this tutorial, we will typically use the intercapped approach because it's easier to read (it's easy to mistake an underscore for a space in dense blocks of code). But it's common to see either -- the C++ standard library uses the underscore method for both variables and functions. Sometimes you'll see a mix of the two: underscores used for variables and intercaps used for functions.

It's worth noting that if you're working in someone else's code, it's generally considered better to match the style of the code you are working in than to rigidly follow the naming conventions laid out above.

Best practice

When working in an existing program, use the conventions of that program (even if they don't conform to modern best practices). Use modern best practices when you're writing new programs.

Second, you should avoid naming your identifiers starting with an underscore, as these names are typically reserved for OS, library, and/or compiler use.

Third, your identifiers should make clear what the value they are holding means (particularly if the units aren't obvious). Identifiers should be named in a way that would help someone who has no idea what your code does be able to figure it out as quickly as possible. In 3 months, when you look at your program again, you'll have forgotten how it works, and you'll thank yourself for picking variable names that make sense.

However, giving a trivial variable an overly complex name impedes overall understanding of what the program is doing almost as much as giving a widely used identifier an inadequate name. Therefore, a good rule of thumb is to make the length of an identifier proportional to how widely it is used. An identifier with a trivial use can have a short name (e.g. such as *i*). An identifier that is used more broadly (e.g. a function that is called from many different places in a program) should have a longer and more descriptive name (e.g. instead of *open*, try *openFileOnDisk*).

int ccount	Bad	What does the c before "count" stand for?
int customerCount	Good	Clear what we're counting
int i	Either	Okay if use is trivial, bad otherwise
int index	Either	Okay if obvious what we're indexing
int totalScore	Either	Okay if there's only one thing being scored, otherwise too ambiguous
int _count	Bad	Do not start names with underscore

int count	Either	Okay if obvious what we're counting
int data	Bad	What kind of data?
int time	Bad	Is this in seconds, minutes, or hours?
int minutesElapsed	Good	Descriptive
int value1, value2	Either	Can be hard to differentiate between the two
int numApples	Good	Descriptive
int monstersKilled	Good	Descriptive
int x, y	Either	Okay if use is trivial, bad otherwise

In any case, avoid abbreviations (unless they are common/unambiguous). Although they reduce the time you need to write your code, they make your code harder to read. Code is read more often than it is written, the time you saved while writing the code is time that every reader, including the future you, wastes when reading it. If you're looking to write code faster, use your editor's auto-complete feature.

For variable declarations, it is useful to use a comment to describe what a variable is going to be used for, or to explain anything else that might not be obvious. For example, say we've declared a variable named `numberOfChars` that is supposed to store the number of characters in a piece of text. Does the text "Hello World!" have 10, 11, or 12 characters? It depends on whether we're including whitespace or punctuation. Rather than naming the variable `numberOfCharsIncludingWhitespaceAndPunctuation`, which is rather lengthy, a well placed comment on or above the declaration line should help the user figure it out:

```
// a count of the number of chars in a piece of text, including whitespace and
punctuation
int numberOfChars;
```

Quiz time

Question #1

Based on how you *should* name a variable, indicate whether each variable name is conventional (follows best practices), unconventional (compiler will accept but does not follow best practices), or invalid (will not compile), and why.

int sum {};

(Assume it's obvious what we're summing)
Show Solution

int _apples {};
Show Solution

int VALUE {};
Show Solution

int my variable name {};
Show Solution

int TotalCustomers {};
Show Solution

int void {};
Show Solution

int numFruit {};
Show Solution

int 3some {};
Show Solution

int meters_of_pipe {};
Show Solution