

# C++ Reference Book

---

## Contents

### 1. Language Basics

- [Compilation Steps](#)
- [Tokens](#)
- [Comments](#)
- [Character Sets](#)
- [Alternative Tokens](#)
- [Expression Rules](#)

### 2. Declarations

- [Declarations and Definitions](#)
- [Scope](#)
- [Name Lookup](#)
- [Linkage](#)
- [Type Declarations](#)
- [Object Declarations](#)
- [Namespaces](#)

### 3. Expressions

- [Lvalues and Rvalues](#)
- [Type Conversions](#)
- [Constant Expressions](#)
- [Expression Evaluation](#)

### 4. Statements

- [Expression Statements](#)
- [Declarations](#)
- [Compound Statements](#)
- [Selections](#)
- [Loops](#)
- [Control Statements](#)
- [Handling Exceptions](#)

### • 5. Functions ..... 83

- [Function Declarations](#)
- [Function Definitions](#)
- [Function Overloading](#)
- [Operator Overloading](#)
- [The main Function](#)

### • 6. Classes ..... 132

- [Class Definitions](#)

- Data Members
- Member Functions
- Inheritance
- Access Specifiers
- Friends
- Nested Types
- **7. Templates** ..... 177
  - Overview of Templates
  - Template Declarations
  - Function Templates
  - Class Templates
  - Specialization
  - Partial Specialization
  - Instantiation
  - Name Lookup
  - Tricks with Templates
  - Compiling Templates
- **8. Standard Library** ..... 211
  - Overview of the Standard Library
  - C Library Wrappers
  - Wide and Multibyte Characters
  - Traits and Policies
  - Allocators
  - Numerics
- **9. Input and Output** ..... 229
  - Introduction to I/O Streams
  - Text I/O
  - Binary I/O
  - Stream Buffers
  - Manipulators
  - Errors and Exceptions
- **10. Containers, Iterators, and Algorithms** ..... 246
  - Containers
  - Iterators
  - Algorithms
- **11. Preprocessor Reference** ..... 276
- **12. Language Reference** ..... 290
- **13. Library Reference** ..... 328
  - `<algorithm>`
  - 
  - 
  - 
  - 
  - 
  - 
  - 
  -

- [illegible]

# Language Basics

## Compilation Steps

A C++ source file transforms through several stages to become an executable program:

1. **Character Translation:** Source characters are translated to the source character set. Trigraph sequences are converted, and end-of-line sequences are replaced by newlines.
2. **Backslash Handling:** Backslashes followed by newlines are removed, except in universal characters or at the file's end. This can extend preprocessor directives or comments.
3. **Tokenization:** The source is divided into preprocessor tokens, including header names, identifiers, and literals.
4. **Preprocessing:** Macros are expanded, and `#include` files are processed.
5. **Literal Conversion:** Character and string literals are converted to the execution character set.
6. **String Concatenation:** Adjacent string literals are concatenated, with errors for mixing narrow and wide types.
7. **Main Compilation:** The source undergoes the primary compilation process.
8. **File Combination:** Compiled files are combined, with template instantiations identified and compiled.
9. **Linking:** External references are resolved, and compiled files are linked to create an executable.

## Tokens

Source code is divided into a stream of tokens, adhering to the "max munch" rule, where the compiler collects as many characters as possible for a valid token. Tokens include identifiers, keywords, literals, and symbols.

The differences between a preprocessor token and a compiler token are small:

- The preprocessor and the compiler might use different encodings for character and string literals.
- The compiler treats integer and floating-point literals differently; the preprocessor does not.
- The preprocessor recognizes `<header>` as a single token (for `#include` directives); the compiler does not.

## Identifiers

Identifiers are names defined by the programmer or in libraries. They begin with a nondigit (letter, underscore, or universal character) and are followed by any combination of digits and nondigits.

Restrictions include:

- Double underscores (`__`) or an underscore followed by an uppercase letter are reserved.
- Identifiers starting with an underscore are reserved in the global namespace.

## Keywords

Keywords are reserved identifiers for language use, such as `int`, `return`, `class`, etc. Some compilers may allow certain keywords as identifiers.

## C++ Keywords

This table lists some of the keywords in the C++ programming language.

Keyword	Description
<code>and</code>	Logical AND operator
<code>and_eq</code>	Bitwise AND assignment operator
<code>asm</code>	Keyword for inline assembly code
<code>auto</code>	Keyword to declare a variable with automatic storage duration
<code>bitand</code>	Bitwise AND operator
<code>bitor</code>	Bitwise OR operator
<code>bool</code>	Keyword for boolean data type
<code>break</code>	Keyword to exit a loop or switch statement
<code>case</code>	Keyword used in switch statements
<code>catch</code>	Keyword for exception handling (catch block)
<code>char</code>	Keyword for character data type
<code>class</code>	Keyword to define a class (user-defined data type)
<code>compl</code>	Bitwise complement operator
<code>const</code>	Keyword to declare a constant variable
<code>const_cast</code>	Cast that removes const-ness
<code>continue</code>	Keyword to skip to the next iteration in a loop
<code>default</code>	Keyword used in switch statements (default case)
<code>delete</code>	Keyword to deallocate memory
<code>do</code>	Keyword to introduce a do-while loop
<code>double</code>	Keyword for double-precision floating-point data type
<code>dynamic_cast</code>	Runtime cast operator
<code>else</code>	Keyword used for conditional statements
<code>enum</code>	Keyword to define an enumerated data type
<code>explicit</code>	Keyword to specify a constructor must be called explicitly
<code>export</code>	Keyword used for exporting symbols (C++)
<code>extern</code>	Keyword to declare a variable or function defined elsewhere

Keyword	Description
false	Keyword for boolean false value
float	Keyword for single-precision floating-point data type
for	Keyword to introduce a for loop
friend	Keyword to declare a friend function or class
goto	Keyword for jumping to a labeled statement (not recommended)
if	Keyword for conditional statements
inline	Keyword to suggest inlining a function
int	Keyword for integer data type
long	Keyword for long integer data type
mutable	Keyword to declare a member of a const class that can be modified
namespace	Keyword to define a namespace
new	Keyword to allocate memory
not	Logical NOT operator
not_eq	Bitwise NOT-equal operator
operator	Keyword to define overloaded operators
or	Logical OR operator
or_eq	Bitwise OR assignment operator
private	Access specifier for class members (private access)
protected	Access specifier for class members (protected access)
public	Access specifier for class members (public access)
register	Keyword to suggest keeping a variable in a register (compiler-dependent)
reinterpret_cast	Cast that can perform any type conversion
return	Keyword to return a value from a function
short	Keyword for short integer data type
signed	Keyword to declare a signed integer
sizeof	Keyword to get the size of a variable or type
static	Keyword to declare a variable with static storage duration
static_cast	Cast that performs safe type conversions
struct	Keyword to define a structure data type
switch	Keyword to introduce a switch statement

Keyword	Description
template	Keyword to define function or class templates (C++)
this	Keyword to refer to the current object
throw	Keyword to throw an exception
true	Keyword for boolean true value
try	Keyword for exception handling (try block)
typedef	Keyword to create an alias for an existing type
typeid	Keyword to get the type information of an expression
typename	Keyword used in template metaprogramming
union	Keyword to define a union data type
unsigned	Keyword to declare an unsigned integer
using	Keyword to introduce a using declaration
virtual	Keyword to declare virtual functions (C++)
void	Keyword for the void data type (no value)
volatile	Keyword to declare
wchar_t	Keyword for wide character type
while	Keyword for while loops
xor	Keyword for bitwise XOR operation
xor_eq	Keyword for bitwise XOR assignment

Literals

- **Integer Literals:** Can be decimal, octal, or hexadecimal with optional suffixes (U, L). For example, 314, 0xFEeL.
- **Floating-point Literals:** Have an integer part, decimal point, and exponent. Types are double, float (F), or long double (L).
- **Boolean Literals:** true and false.
- **Character Literals:** Enclosed in single quotes, prefixed with L for wide characters (e.g., L'x').

Table: Character escape sequences

Escape Sequence	Meaning
\\	\ character
\'	' character
\"	" character

Escape Sequence	Meaning
\?	? character (used to avoid creating a trigraph)
\a	Alert or bell
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\ooo	Octal number of one to three digits
\xhh...	Hexadecimal number of one or more digits

- **String Literals:** Enclosed in double quotes, with `L` for wide strings. Adjacent strings are concatenated at compile time.

Escape sequences are used in character and string literals, such as `\n` for newline or `\\` for a backslash.

String literals have a type of `const char[]` or `const wchar_t[]` and can be converted to pointers.

Symbols

Nonalphabetic symbols are used as operators and as punctuation (e.g., statement terminators). Some symbols are made of multiple adjacent characters. The following are all the symbols used for operators and punctuation:

{	(	?:	.	^	.	=	!=	--	&=
}	)	:%:	+	&	.*	==	<<	+=	\ =
[	<:	;	-	\	->	<	>>	*=	^=
]	:>	:	*	?	->*	>	<<=	/=	++
#	<%	...	/	:	~	<=	>>=	%=	--
##	%>	,	%	::	!	>=			

- **No Whitespace in Symbols:** You cannot insert whitespace between characters that make up a symbol. C++ will collect as many characters as it can to form a symbol before interpreting it. For example, `x++++y` is read as `x ++ + y`.
- **Template Instantiation:** A common error is omitting a space between closing angle brackets in nested template instantiation. Example:



```
std::list<std::vector<int> > list;
```

Note: The space between > is crucial. Without it, >> would be interpreted as a right-shift operator, not two separate closing angle brackets

```
::std::list< ::std::list<int> > list;
```

Note: Spaces are needed between the angle bracket < and the scope operator ::. This prevents the compiler from misinterpreting <: as an alternative token.

## Comments

Comments start with `/*` and end with `*/`. These comments do not nest. For example:

```
/* this is a comment /* still a comment */  
int not_in_a_comment;
```

A comment can also start with `//`, extending to the end of the line. For example:

```
const int max_widget = 42; // Largest size of a widget
```

Within a `/*` and `/* comment`, `//` characters have no special meaning. Within a `//` comment, `/` and `*/` have no special meaning. Thus, you can “nest” one kind of comment within the other. For example:

```
/* Comment out a block of code:  
const int max_widget = 42; // Largest size of a widget  
*/  
/** Inhibit the start of a block comment  
const int max_widget = 10; // Testing smaller widget limit  
**/
```

A comment is treated as whitespace. For example, `str/comment/ing` describes two separate tokens, `str` and `ing`.