

KEYWORDS IN C++: A CHATGPT SUMMARY

2-28-2025

by Atomic Badger

"Show a list of all keywords in C++ 17. After the list give an example of each one in use with an explanation and using a snippet of code."

ChatGPT said:

List of All C++17 Keywords:

C++17 includes the following keywords:

alignas, alignof, and, and_eq, asm, atomic_cancel, atomic_commit, atomic_noexcept, auto, bitand, bitor, bool, break, case, catch, char, char16_t, char32_t, class, compl, concept, const, constexpr, const_cast, continue, co_await, co_return, co_yield, 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, return, short, signed, sizeof, static, static_assert, static_cast, struct, switch, synchronized, template, this, thread_local, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while, xor, xor_eq.

Examples and Explanations:

1. alignas

Specifies the alignment requirement for a variable or struct.

```
#include <iostream>
#include <cstdint>

struct alignas(16) AlignedData {
    int data;
};

int main() {
    std::cout << alignof(AlignedData) << std::endl; // Output: 16
}
```

2. alignof

Returns the alignment requirement of a type.

```
#include <iostream>

struct MyStruct {
    double a;
    char b;
};

int main() {
    std::cout << alignof(MyStruct) << std::endl; // Output depends on system
}
```

3. and, and_eq (Alternative to && and &=)

```
#include <iostream>

int main() {
    bool a = true, b = false;
    if (a and not b) { // Equivalent to if (a && !b)
        std::cout << "Condition met\n";
    }
}
```

4. asm (Inline Assembly, Compiler Specific)

```
int main() {
    asm("nop"); // No-operation instruction (GCC/Clang specific)
}
```

5. atomic_cancel, atomic_commit, atomic_noexcept (Used for transactional memory, rarely used)

// These are reserved keywords for future transactional memory extensions.

6. auto

Automatically deduces type.

```
#include <iostream>
```

```
int main() {
    auto x = 42; // x is int
    std::cout << x << std::endl;
}
```

7. bitand, bitor (Alternative for & and |)

```
#include <iostream>
```

```
int main() {
    int x = 5 bitand 3; // Equivalent to 5 & 3
    std::cout << x << std::endl;
}
```

8. bool

Boolean type.

```
#include <iostream>
```

```
int main() {
    bool flag = true;
    std::cout << std::boolalpha << flag << std::endl;
}
```

9. break

Exits a loop.

```
#include <iostream>
```

```
int main() {
    for (int i = 0; i < 10; i++) {
        if (i == 5) break;
        std::cout << i << " ";
    }
}
```

10. case

Used in switch statements.

```
#include <iostream>
```

```
int main() {
```

```
int x = 2;
switch (x) {
    case 1: std::cout << "One"; break;
    case 2: std::cout << "Two"; break;
    default: std::cout << "Other";
}
}
```

11. catch

Handles exceptions.

```
#include <iostream>

int main() {
    try {
        throw std::runtime_error("Error occurred");
    } catch (const std::exception &e) {
        std::cout << e.what() << std::endl;
    }
}
```

12. char, char16_t, char32_t, wchar_t

Character types.

```
#include <iostream>

int main() {
    char c = 'A';
    wchar_t wc = L'Ω';
    std::cout << c << std::endl;
}
```

13. class

Defines a class.

```
#include <iostream>

class MyClass {
public:
    void greet() { std::cout << "Hello, World!" << std::endl; }
};

int main() {
    MyClass obj;
    obj.greet();
}
```

14. constexpr

Evaluates at compile-time.

```
constexpr int square(int x) { return x * x; }

int main() {
    constexpr int val = square(5);
}
```

15. co_await, co_return, co_yield (Used for coroutines)

```
#include <coroutine>

std::suspend_always example() { co_return; }

int main() { example(); }
```

16. decltype

Returns type of expression.

```
#include <iostream>

int main() {
    int x = 5;
    decltype(x) y = 10; // y is also int
}
```

17. default

Provides a default implementation for constructors.

```
class MyClass {
public:
    MyClass() = default;
};
```

18. delete

Deletes objects in heap memory.

```
int* ptr = new int(5);
```

```
delete ptr;
```

19. do

Executes a loop at least once.

```
int i = 0;
do {
    std::cout << i << " ";
} while (++i < 5);
```

20. double

Double-precision floating-point number.

```
double pi = 3.14159;
```

21. dynamic_cast

Casts a base class pointer to derived class.

```
class Base { virtual void foo() {} };
class Derived : public Base {};
Base* b = new Derived();
Derived* d = dynamic_cast<Derived*>(b);
```

22. else

Conditional statement.

```
if (false) {} else { std::cout << "Else executed"; }
```

23. enum

Enumerations.

```
enum Color { RED, GREEN, BLUE };
Color c = GREEN;
```

24. explicit

Prevents implicit conversion.

```
class MyClass {
public:
    explicit MyClass(int x) {}
};
MyClass obj(5); // Works
// MyClass obj = 5; // Error
```

25. export (Not used in standard C++17; reserved for modules in C++20)

// Not used in C++17, but reserved for exporting modules in C++20.

26. extern

Declares external variables or functions.

```
#include <iostream>

extern int globalVar;

int main() {
    std::cout << globalVar << std::endl;
}
```

27. false

Boolean false value.

```
bool flag = false;
if (!flag) std::cout << "False is working!";
```

28. float

Single-precision floating-point type.

```
float pi = 3.14f;
```

29. for

Loop structure.

```
for (int i = 0; i < 5; i++) {  
    std::cout << i << " ";  
}
```

30. friend

Allows non-member functions or classes to access private members.

```
class MyClass {  
    int secret = 42;  
    friend void revealSecret(MyClass);  
};  
  
void revealSecret(MyClass obj) {  
    std::cout << obj.secret;  
}
```

31. goto

Jumps to a labeled statement.

```
#include <iostream>  
  
int main() {  
    goto skip;  
    std::cout << "This won't print.";  
skip:  
    std::cout << "Jumped here!";  
}
```

32. if

Conditional statement.

```
if (true) {  
    std::cout << "Condition met!";  
}
```

33. inline

Suggests inline expansion of functions.

```
inline int square(int x) { return x * x; }
```

34. int

Integer data type.

```
int number = 10;
```

35. long

Long integer type.

```
long bigNumber = 1000000L;
```

36. mutable

Allows modification of a member in a const object.

```
class MyClass {  
public:  
    mutable int counter = 0;  
};  
  
void modify(const MyClass& obj) {  
    obj.counter++; // Allowed due to mutable  
}
```

37. namespace

Defines a scope for identifiers.

```
namespace MyNamespace {  
    int value = 10;  
}
```

38. new

Allocates memory dynamically.

```
int* ptr = new int(5);  
delete ptr;
```

39. noexcept

Indicates a function does not throw exceptions.

```
void myFunction() noexcept {}
```

40. not, not_eq (Alternative for ! and !=)

```
bool x = not false; // true
int a = 5, b = 3;
bool result = (a not_eq b); // true
```

41. nullptr

Represents a null pointer.

```
int* ptr = nullptr;
```

42. operator

Overloads operators.

```
class MyClass {
public:
    int value;
    MyClass operator+(const MyClass& other) {
        return MyClass{value + other.value};
    }
};
```

43. or, or_eq (Alternative for || and |=)

```
bool result = true or false; // true
int x = 5;
x or_eq 2; // Equivalent to x |= 2;
```

44. private, protected, public

Access specifiers in classes.

```
class MyClass {
private:
    int secret;
protected:
    int semiSecret;
public:
    int open;
};
```

45. register (Deprecated; hint to store variable in a register)

`register int x = 10; // Has no effect in modern compilers`

46. reinterpret_cast

Converts between pointer types unsafely.

```
int a = 65;
char* p = reinterpret_cast<char*>(&a);
```

47. requires (Reserved for concepts in C++20)

// Not used in C++17, used in C++20 for constraints.

48. return

Exits from a function.

```
int sum(int a, int b) {
    return a + b;
}
```

49. short

Short integer type.

```
short s = 32767;
```

50. signed

Signed integer type.

```
signed int x = -10;
```

51. sizeof

Gets size of a type.

```
std::cout << sizeof(int);
```

52. static

Defines static storage duration.

```
void func() {  
    static int counter = 0; // Persists across function calls  
    counter++;  
}
```

53. static_assert

Compile-time assertion.

```
static_assert(sizeof(int) == 4, "Unexpected int size");
```

54. static_cast

Safely casts between types.

```
double d = 3.14;  
int i = static_cast<int>(d);
```

55. struct

Defines a structure.

```
struct Point {  
    int x, y;  
};
```

56. switch

Multi-way conditional branching.

```
switch (2) {  
    case 1: std::cout << "One"; break;  
    case 2: std::cout << "Two"; break;  
}
```

57. synchronized (Reserved for future concurrency features)

// Reserved keyword in C++17

58. template

Defines templates.

```
template <typename T>
T square(T x) { return x * x; }
```

59. this

Pointer to the current instance.

```
class MyClass {
public:
    void show() { std::cout << this; }
};
```

60. thread_local

Declares a thread-local variable.

```
thread_local int counter = 0;
```

61. throw

Throws an exception.

```
throw std::runtime_error("Error");
```

62. true

Boolean true value.

```
bool flag = true;
```

63. try

Starts an exception-handling block.

```
try {  
    throw std::runtime_error("Oops");  
} catch (...) {  
    std::cout << "Caught exception";  
}
```

64. typedef

Defines an alias for a type.

```
typedef unsigned int uint;
```

65. typeid

Gets type information.

```
#include <typeinfo>  
std::cout << typeid(int).name();
```

66. typename

Used in templates to indicate a type.

```
template <typename T> class MyClass {};
```

67. union

Defines a union.

```
union Data {  
    int i;  
    float f;  
};
```

68. unsigned

Unsigned integer type.

```
unsigned int x = 10;
```

69. using

Creates a type alias.

```
using uint = unsigned int;
```

70. virtual

Defines virtual functions.

```
class Base { virtual void foo() {} };
```

71. void

Indicates no return value.

```
void function() {}
```

72. volatile

Prevents compiler optimizations.

```
volatile int flag;
```

END OF LIST