C++ Programming

The Basics

C++ : almost a
super subset
of C

The Basics

- Statements and expressions
- Control flow
- Structures

Statements

- As in C, a statement in C++ is a command in a program to direct the program to take a particular action
- Likely the simplest statement in C++ is an expression statement
  - This is simply an expression (which is optional) followed by a semicolon(;)
  - This does mean that the following (the null statement) is a valid statement:

this is a legal statement. -> ; <= do nothing i.e. for a cycle.

- Usually though, an expression is given and that expression is evaluated when the statement is executed

Expressions and Types

- Every expression has a type that determines the operations that may be performed on it
- A declaration is a statement that introduces a name into a program, specifying a type for the named entity
- For example:

```
int foo;
```

introduces a named variable foo that is of type integer

Expressions and Types

- C++ has a similar set of basic types to C
- This includes things like bool, char, int, and double
- Each fundamental type corresponds directly to hardware facilities and has a fixed size that dictates the range of values that it can represent
- C++ also has a collection of other types that C traditionally does not; we will touch on some of those later on

Expressions and Types

- C++11 introduced the concept of auto typing
- In this, the keyword auto can be used to avoid explicitly declaring the type of an entity (like a variable)
- Instead, the type is inferred by the context, such as the type of the expression used to initialize a variable
- For example:

```
auto n = 0;     // Create an integer variable n and initialize it to 0
```

Expressions and Types

- The use of auto in C++ is divisive in the community, however
  - Purists say it should be avoided and types should always be explicitly declared for readability and to maintain compatibility with older C++ code and compilers
  - Others, however, find the shorthand useful and a step towards languages without explicit declarations (like Python)
- Either way, you will see auto in the wild, so it is good to know what it does and how to use it

Constant Expressions

in pre-compile stage, search and replace these value explicitely. however, if you have #define pi 3.14, then pie would be 3.14e, that's bad.

- C++ has multiple concepts of constants:
  - `#define` constants – preprocessor definitions that do a simple replacement during preprocessing and before compiling
  - `const` constants – named constant declarations where the programmer commits to not changing a value, and this promise is enforced by the compiler
  - `constexpr` constants – constant expressions evaluated at compile time, allowing them to be placed in read-only memory to improve performance (new in C++11)

any attempt that tries to change the value would fail.

at compile time

Constant Expressions

if you have an uninitialized variable, the value is unknown and is depend on the initial value in memory, just like an uninitialize pointer.

Debugger initialize the memory to zero at first.

it could be calculate during compile time.

```
#define PI 3.14
constexpr double circleArea(double x) { return PI*x*x; }

int main() {
    const double pi = 3.14;
    constexpr double radius1 = 5.0;
    const double radius2 = 10.0;

    constexpr double area1 = circleArea(radius1);
    constexpr double area2_error = circleArea(radius2);
    const double area2_good = circleArea(radius2);
}
```

case-sensitive

that is runtime

time x would not known during compile time

const only works in const func.

setup at runtime. the compiler make sure it does not change

safe.

also safe since r1 is also constexpr

it is const

it set up at compile time.

the compiler would not set up during program run.

it works since it is also const.

constexpr: everything it touch must be constexpr

same for const.

constexpr is not recommend in coding unless u know what u doing.

Expressions and Operators

both const and constexpr would not modify
however, constexpr is a compile time const, i.e shall be initialize during compile time, const could be set up at compile time or run time

- C++ also uses a similar set of operators to C
- Arithmetic operators like +, -, *, /, and %
- Comparison operators like ==, !=, <, >, <=, and >=
- Assignment and initialization operator as =
- In assignments and arithmetic operations, C++ does do conversions between basic types as in C i.e. int => float.

Expressions and Operators

```
#include <iostream>
using namespace std;

int main()
{
    int sum;
    bool bigEnough;
    sum = 50 + 50;
    bigEnough = (sum >= 100);
    cout << "Sum is " << sum << ". Is it big
      enough?  " << bigEnough << endl;
}
```

the initlize value is unknown

now we initialize it =

comparison operation.

endline code, kinda like \n

Statements and Blocks

in C++, now u could introduce new variable in block of statement.

- A block, or compound statement, is the term given to a collection of statements enclosed in {…} i.e. if statement
  - This is the way that a program can group multiple statements together into a single statement, often for the purposes of control flow
  - A block also defines a scope for variables declared in it; local variables are put on the stack for the duration of the execution of the block statement
  - Such variable declarations are optional, and no new variables need be introduced in a block
  - Originally all such declarations had to be at the beginning of the block, but they can now generally be interwoven with statements through the code

in C++, you could create a compound statement without any reason (i.e. if, then, for ...) as u like.

it is usually for defining variable and control its scope.

Control Flow

- In C++, the flow of control in a program behaves very similarly to how things are done in C
  - Sequence: The normal stepping through from one statement to the next
  - Selection: To enable selection for the next statement or block from amongst a number of possibilities
  - Repetition: Repetition of a single statement or of a block; repetition can take many forms depending on the needs of the program, including for(...), while(...), and do...while loops.

Control Flow – Selection

- One-way selection is accomplished using a simple if

```
if (<logical expression>)
    <statement1> //performed if expression is true
```

Control Flow – Selection

- Two-way selection is accomplished using if-else

```
if (<logical expression>)
     <statement1> //performed if expression is true
else
     <statement2> //performed if false
```

Control Flow – Selection

- Multi-way selection using the else if

```
if(<expression1>)
   <statement1>
else if (<expression2>)
   <statement2>
else if (<expression3>)
   <statement3>
else
   <statement4> // otherwise -- default
```

Control Flow – Selection

- Multi-way selection using the switch / case statement

input from user

if there is no [break;], it would
fall along cases, Like 1->2->3 ...

```
switch (<expression>){
case <constant-exp1>: <statement(s)1> [break;]
case <constant-exp2>: <statement(s)2> [break;]
case <constant-exp3>: <statement(s)3> [break;]
default: <statement(s)d>; [break;]
}
```

Control Flow – Repetition

- while loops:

```
while(<expr>) <statement>


int count = 0;
while (count < 10) {
     cout << "Count is: " << count << endl;
     count++;
}
```

Control Flow – Repetition

- for loops:

```
for(<expr-init>;<condition expr>;<expr-iter>)
     <statement>
```

initialization expression iteration .

```
for(int count = 0; count < 10; count++) {
     cout << "Count is: " << count << endl;
}
```

C++ also allows for iterators.

Control Flow – Repetition

- do-while loops:

```
do
    <statement>
while (<expr>);
```

```
int count = 0;
do {
    cout << "Count is: " << count << endl;
    count++;
} while (count < 10);
```

Structures

it is the only way in C to store multiple types of data.

- In C++, a user-defined record (aggregate type) is called a structure and is referred to as a `struct` in a program
- They are handled in much the same way as they are in C, with slight differences (that ultimately make them easier to use and refer to)
- Interestingly in C++, structures and classes are more-or-less equivalent except for their default visibility:
  - In a structure, members default to public; whereas, i.e. globally visibility.
  - In a class, members default to private.
- Much more on classes shortly!

Structures

- As noted previously, structures are defined using the `struct` keyword

```
struct Point {
 int x;
 int y;
};
Point p1, p2;
```

- Notice the difference between how C++ and C would handle declaring things using our new `Point` structure?

Structures

point p1; error
struct point p1

- C would require us to refer to our structure as `struct Point` or else it would generate an error during compilation
- As this would get tedious, you could use the `typedef` directive to name a new type to avoid having to do this
- In C++, however, this is not necessary; you can do things the C way, but you are not required to do so

Structures

- Accessing fields: Individual fields of a structure can be accessed using the "." syntax

```
Point p1;
p1.x = 3;
p1.y = 2;
```

When we have a pointer to a structure (more on pointers in a minute), we can instead access fields using "->"

Structures

- Assignment: A structure can be assigned as a complete unit:

```
Point p1, p2;
p1.x = 3;
p1.y = 2;
p2 = p1;
```

But for most time, we use class instead of struct in defining an object.

class is better since it is private by default

This last statement is equivalent to:

```
p2.x = p1.x;
p2.y = p1.y;
```

p2 get a full copy of p1.

What's the difference between constexpr and const?

Asked 10 years, 8 months ago Modified 21 days ago Viewed 331k times

825

What's the difference between `constexpr` and `const` ?

- When can I use only one of them?
- When can I use both and how should I choose one?

c++ c++11 constants constexpr

Share Edit Follow

edited Apr 17, 2022 at 13:59
nbro
15.4k 32 113 197

asked Jan 2, 2013 at 1:42
MBZ
26.1k 47 114 191

131 `constexpr` creates a compile-time constant; `const` simply means that value cannot be changed. – David G Jan 2, 2013 at 1:44

1 Also see When should you use constexpr capability in C++11? – jww Sep 3, 2016 at 3:53

May be this article from `boost/hana` library can enlight some `constexpr` issues where you can use `constexpr` and where you can't: boost.org/doc/libs/1_69_0/libs/hana/doc/html/... – Andry Feb 7, 2019 at 8:38

1 @0x499602D2 "*simply means that value cannot be changed*" For a scalar initialized with a literal, a value that cannot be change *is* also a compile time constant. – curiousguy Jan 20, 2020 at 23:09

@curiousguy Yeah my comment was very oversimplified. Admittedly I was new to `constexpr` back then too :) – David G Jan 20, 2020 at 23:12

10 Answers

Sorted by:
Highest score (default)

777

✓

Basic meaning and syntax

Both keywords can be used in the declaration of objects as well as functions. The basic difference when applied to *objects* is this:

- `const` declares an object as *constant*. This implies a guarantee that once initialized, the value of that object won't change, and the compiler can make use of this fact for optimizations. It also helps prevent the programmer from writing code that modifies objects that were not meant to be modified after initialization.
- `constexpr` declares an object as fit for use in what the Standard calls *constant expressions*. But note that `constexpr` is not the only way to do this.

When applied to *functions* the basic difference is this:

- `const` can only be used for non-static member functions, not functions in general. It gives a guarantee that the member function does not modify any of the non-static data members (except for mutable data members, which can be modified anyway).
- `constexpr` can be used with both member and non-member functions, as well as constructors. It declares the function fit for use in *constant expressions*. The compiler will only accept it if the function meets certain criteria (7.1.5/3,4), most importantly (†):
  - The function body must be non-virtual and extremely simple: Apart from typedefs and static asserts, only a single `return` statement is allowed. In the case of a constructor, only an initialization list, typedefs, and static assert are allowed. (`= default` and `= delete` are allowed, too, though.)
  - As of C++14, the rules are more relaxed, what is allowed since then inside a constexpr function: `asm` declaration, a `goto` statement, a statement with a label other than `case` and `default`, try-block, the definition of a variable of non-literal type, definition of a variable of static or thread storage duration, the definition of a variable for which no initialization is performed.
  - The arguments and the return type must be *literal types* (i.e., generally speaking, very simple types, typically scalars or aggregates)

Constant expressions

As said above, `constexpr` declares both objects as well as functions as fit for use in constant expressions. A constant expression is more than merely constant:

- It can be used in places that require compile-time evaluation, for example, template parameters and array-size specifiers:

  ```
  template<int N>
  class fixed_size_list
  { /*...*/ };

  fixed_size_list<X> mylist;  // X must be an integer constant expression

  int numbers[X];  // X must be an integer constant expression
  ```

- But note:
- Declaring something as `constexpr` does not necessarily guarantee that it will be evaluated at compile time. It *can be used* for such, but it can be used in other places that are evaluated at run-time, as well.
- An object *may* be fit for use in constant expressions *without* being declared `constexpr`. Example:

  ```
    int main()
    {
      const int N = 3;
      int numbers[N] = {1, 2, 3};  // N is constant expression
    }
  ```

This is possible because `N`, being constant and initialized at declaration time with a literal, satisfies the criteria for a constant expression, even if it isn't declared `constexpr`.

So when do I actually have to use `constexpr`?

- An **object** like `N` above can be used as constant expression *without* being declared `constexpr`. This is true for all objects that are:
  - `const` *and*
  - of integral or enumeration type *and*
  - initialized at declaration time with an expression that is itself a constant expression.

[This is due to §5.19/2: A constant expression must not include a subexpression that involves "an lvalue-to-rvalue modification unless […] a glvalue of integral or enumeration type […]" Thanks to Richard Smith for correcting my earlier claim that this was true for all literal types.]

- For a **function** to be fit for use in constant expressions, it **must** be explicitly declared `constexpr`; it is not sufficient for it merely to satisfy the criteria for constant-expression functions. Example:

```
template<int N>
class list
{ };

constexpr int sqr1(int arg)
{ return arg * arg; }

int sqr2(int arg)
{ return arg * arg; }

int main()
{
  const int X = 2;
  list<sqr1(X)> mylist1;  // OK: sqr1 is constexpr
  list<sqr2(X)> mylist2;  // wrong: sqr2 is not constexpr
}
```

When can I / should I use both, `const` and `constexpr` *together?*

A. In object declarations. This is never necessary when both keywords refer to the same object to be declared. `constexpr` implies `const`.

```
constexpr const int N = 5;
```

is the same as

```
constexpr int N = 5;
```

However, note that there may be situations when the keywords each refer to different parts of the declaration:

```
static constexpr int N = 3;

int main()
{
  constexpr const int *NP = &N;
}
```

Here, `NP` is declared as an address constant-expression, i.e. a pointer that is itself a constant expression. (This is possible when the address is generated by applying the address operator to a static/global constant expression.) Here, both `constexpr` and `const` are required: `constexpr` always refers to the expression being declared (here `NP`), while `const` refers to `int` (it declares a pointer-to-const). Removing the `const` would render the expression illegal (because (a) a pointer to a non-const object cannot be a constant expression, and (b) `&N` is in-fact a pointer-to-constant).

B. In member function declarations. In C++11, `constexpr` implies `const`, while in C++14 and C++17 that is not the case. A member function declared under C++11 as

```
constexpr void f();
```

needs to be declared as

```
constexpr void f() const;
```

under C++14 in order to still be usable as a `const` function.

Share Edit Follow

edited Aug 21 at 19:13
Benjamin Wang
226 1 9

answered Jan 2, 2013 at 5:10
jogojapan
68.4k 11 101 131

14 IMO the "not necessarily evaluated at compile time" is less helpful than thinking of them as "evaluated at compile time". The constraints on a constant expression mean that it would be relatively easy for a compiler to evaluate it. A compiler must complain if those constraints are not satisfied. Since there are no side effects, you can never tell a difference whether a compiler "evaluated" it or not. – aschepler Jan 2, 2013 at 5:27

21 @aschepler Sure. My main point there is that if you call a `constexpr` function on a non-constant expression, e.g. an ordinary variable, this is perfectly legal and the function will be used like any other function. It will not be evaluated at compile time (because it can't). Perhaps you think that's obvious -- but if I stated that a function declared as `constexpr` will always be evaluated at compile-time, it could be interpreted in the wrong way. – jogojapan Jan 2, 2013 at 5:34

8 Yes, I was talking about `constexpr` objects, not functions. I like to think of `constexpr` on objects as forcing compile time evaluation of values, and `constexpr` on functions as allowing the function to be evaluated at compile time or run time as appropriate. – aschepler Jan 2, 2013 at 5:38

5 A correction: 'const' is only a restriction that YOU cant change the value of a variable; it does not make any promise that the value wont change (ie, by someone else). It's a write property, not a read property. – Jared Grubb Jan 20, 2013 at 17:05

4 This sentence: *It gives a guarantee that the member function does not modify any of the non-static data members.* misses one important detail. Members marked as `mutable` may also be modified by