10.17 — References and const

ALEX AUGUST 26, 2021

Reference to const value

Just like it's possible to declare a pointer to a const value, it's also possible to declare a reference to a const value. This is done by declaring a reference using the const keyword.

```
const int apples{ 5 };
const int& ref{ apples }; // ref is a reference to a const value
```

A reference to a const value is often called aconst reference for short, though this does make for some inconsistent nomenclature with pointers.

Initializing references to const values

Unlike references to non-const values, which can only be initialized with non-const l-values, references to const values can be initialized with non-const l-values, const l-values, and r-values.

```
int x{ 5 };
const int& ref1{ x }; // okay, x is a non-const l-value

const int y{ 7 };
const int& ref2{ y }; // okay, y is a const l-value

const int& ref3{ 6 }; // okay, 6 is an r-value
```

Much like a pointer to a const value, a reference to a const value can reference a non-const variable. When accessed through a reference to a const value, the value is considered const even if the original variable is not:

```
int apples{ 5 };
const int& ref{ apples }; // create const reference to variable apples

apples = 6; // okay, apples is non-const
ref = 7; // illegal -- ref is const
```

References to r-values extend the lifetime of the referenced value

Normally r-values have expression scope, meaning the values are destroyed at the end of the expression in which they are created.

```
std::cout << 2 + 3 << '\n'; // 2 + 3 evaluates to r-value 5, which is destroyed at the end of this statement
```

However, when a reference to a const value is initialized with an r-value, the lifetime of the r-value is extended to match the lifetime of the reference.

```
int somefcn()
{
    const int& ref{ 2 + 3 }; // normally the result of 2+3 has expression scope and is destroyed at the end of this statement
    // but because the result is now bound to a reference to a const value...
    std::cout << ref << '\n'; // we can use it here
} // and the lifetime of the r-value is extended to here, when the const reference dies
```

Const references as function parameters

References used as function parameters can also be const. This allows us to access the argument without making a copy of it, while guaranteeing that the function will not change the value being referenced.

```
// ref is a const reference to the argument passed in, not a
copy
void changeN(const int& ref)
{
 ref = 6; // not allowed, ref is const
}
```

References to const values are particularly useful as function parameters because of their versatility. A const reference parameter allows you to pass in a non-const l-value argument, a const l-value argument, a literal, or the result of an expression:

```
#include <iostream>

void printIt(const int& x)
{
    std::cout << x;
}

int main()
{
    int a{ 1 };
    printIt(a); // non-const l-value

    const int b{ 2 };
    printIt(b); // const l-value

    printIt(3); // literal r-value

    printIt(2+b); // expression r-
value

    return 0;
}

```

The above prints

```
1234
```

To avoid making unnecessary, potentially expensive copies, variables that are not pointers or fundamental data types (int, double, etc...) should be generally passed by (const) reference. Fundamental data types should be passed by value, unless the function needs to change them. There are a few exceptions to this rule, namely types that are so small that it's faster for the CPU to copy them than having to perform an extra indirection for a reference.

A reminder

References act like pointers. The compiler adds the indirection, which we'd do manually on a pointer using an asterisk, for us.

One of those fast types is `std::string_view`. You'll learn about more exceptions later. If you're uncertain if a non-fundamental type is fast to pass by value, pass it by const reference.

Best practice

Pass non-pointer, non-fundamental data type variables (such as structs) by (const) reference, unless you know that passing it by value is faster.

Quiz time

Question #1

Which of the following types should be passed by value, which by const reference? You can assume the function that takes these types as parameters doesn't modify them.

a) `char`
b) `std::string`
c) `unsigned long`
d) `bool`
e) An enumerator
f)

```
struct
Position
{
  double x{};
  double y{};
  double z{};
};
```

g)

```
struct Player
{
  int health{};
  // The Player struct is still under development. More members will be
added.
};
```

h) `double`
i)

```
struct ArrayView
{
  const int*
array{};
  std::size_t
length{};
};
```

For reference, this is how we'd go about using `ArrayView`:

Show Hint

Show Solution

Next lesson
10.18 Member selection with pointers and references

Back to table of contents

Previous lesson
10.16 Reference variables

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

Name*

Email*

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

Notify me about replies: POST COMMENT

94 COMMENTS

Newest