6.11a — References and const

BY ALEX ON JUNE 7TH, 2017 | LAST MODIFIED BY ALEX ON OCTOBER 21ST, 2017

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 value = 5;
const int &ref = value; // ref is a reference to const value
```

References to const values are often called "const references" for short.

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-value, 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 value = 5;
const int &ref = value; // create const reference to variable value

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

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

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; // 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 e
nd of this statement
    // but because the result is now bound to a reference to a const value...
    std::cout << ref; // 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.

Rule: Pass non-pointer, non-fundamental data type variables (such as structs) by (const) reference.


6.12 -- Member selection with pointers and references


Index


6.11 -- Reference variables

Share this:

C++ TUTORIAL | PRINT THIS POST

13 comments to 6.11a — References and const

Yan
October 20, 2017 at 7:26 am · Reply