6.11a — References and const

BY ALEX ON JUNE 7TH, 2017 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

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
```

A reference to a const value is often called a **const 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-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
```

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 stat
```

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
    // 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

C++ TUTORIAL | PRINT THIS POST

52 comments to 6.11a — References and const

hellmet
January 18, 2020 at 6:30 am · Reply

I recently came across an issue. This was the problematic code piece.

```
// Problematic code
const auto& some_OBJ_create_info = Some::Namespace::StructNameOBJCreateInfo()
    .setAttribute1()... // more set stuff (not relevant to the issue)

const auto& obj_instance = Some::Namespace::CreateOBJ()
    .setPCreateInfo(&some_OBJ_create_info)
    .set // more attributes
```

On the other hand, this works in all cases

```
// The right way, works in all compilers and all compile modes
const auto some_OBJ_create_info { Some::Namespace::StructNameOBJCreateInfo()
    .setAttribute1()...
    // more set stuff (not relevant to the issue)
};

const auto obj_instance { Some::Namespace::CreateOBJ()
    .setPCreateInfo(&some_OBJ_create_info) // <- this seems to fail, can't pin it down, as
    .set // more attributes
};
```

The problem-code worked fine in debug mode, but broke miserably in release mode (both compile fine, meaning I am allowed to take the address of an 'const lvalue reference' to an rvalue). I understand that this (the first snippet) is not the ideal way to initialize stuff, but I don't see why optimization (no matter what type) would break the code here. Specifically, the 'Some::Namespace::CreateOBJ' throws an exception depending on the compiler.

nascardriver
January 18, 2020 at 7:40 am · Reply

If something works in one compilation mode but not the other, you either have undefined behavior or a compiler bug (unlikely).
Without a full example, I can't tell for certain what's causing the issue.

What I suspect: `Some::Namespace::StructNameOBJCreateInfo()` creates a new object (Either it's a constructor or a function call, doesn't matter). `setAttributeX` returns a reference to `*this`. The created temporary by the call to `Some::Namespace::StructNameOBJCreateInfo` dies at the end of line 2. A const reference can't perform lifetime extension in this case (Returned references can't be extended).
In snippet 1, you're trying to perform lifetime extension, but the temporary gets destroyed. Accessing the reference invokes UB.
In snippet 2, you're copying the temporary before it dies.

Same issue with `obj_instance`.

If I'm right, this should fix your issue and avoid copies:

```
auto some_OBJ_create_info{ Some::Namespace::StructNameOBJCreateInfo() };

some_OBJ_create_info.setAttribute1() // set more stuff (But no ... in code :))

auto obj_instance{ Some::Namespace::CreateOBJ() };

obj_instance.setPCreateInfo(&some_OBJ_create_info) // should be a constructor argument
  .set // more attributes
```

hellmet
January 18, 2020 at 7:58 am · Reply

> [...] creates a new object [...] setAttributeX` returns a reference to `*this` [...] created temporary by the call to `Some::Namespace::StructNameOBJCreateInfo` dies at the end of line 2.

I was expecting the whole object to be constructed (everything on the rhs of = to be evaluated completely) and then me holding a reference to it. But.. but why would the ... Ohhh!
I see! The last .setX() returns a reference and I can't set a const reference to a reference! Holy shit that makes so much sense! On a related note, what about reference collapsing? Does that not come into play here? I read about that here [http://thbecker.net/articles/rvalue_references/section_01.html] in section 8. I can't say I understood everything, I'm still trying to make sense of that.

> should be a constructor argument really
The calls are part of the Vulkan API, but yeah, this '.set' pattern seems helpful in cases like this, but I must be careful of the consequences! I'm glad I made the mistake by failing to follow best practices, as it made me appreciate the issue, understand C++ better. Thank you for your on-point explanation!

I expect this to work though, and it did.

```
// The right way, works in all compilers and all compile modes.
// Also, the 'this seems to fail' comment in my comment above was meant for the f

const auto some_OBJ_create_info { Some::Namespace::StructNameOBJCreateInfo()
    .setAttribute1()...
};

const auto obj_instance { Some::Namespace::CreateOBJ()
    .setPCreateInfo(&some_OBJ_create_info)
    .set // more attributes
};
```

In both of the above cases (in the code snippet right above), shouldn't the code in '{ thing_here }' 'thing_here' be evaluated completely, with 0 copies since all the .setX() return a reference? With optimizations, I imagine the object is constructed in-place.

nascardriver
January 18, 2020 at 8:14 am · Reply

I don't think you're there yet

```
#include <iostream>

struct S
{
  S& setAttribute(void){ return *this; };

  ~S(void){ std::cout << "~S()\n"; }
};

int main(void)
{
  // Perform lifetime extension on our temporary
  const S& s{ S{} };

  std::cout << "--\n";

  return 0;
}
```

Output
--
~S()

Now we add a `setAttribute` call to the temporary

```
#include <iostream>

struct S
{
  S& setAttribute(void){ return *this; };

  ~S(void){ std::cout << "~S()\n"; }
};

int main(void)
{
  // No lifetime extension! Returned references can't be extended.
  const S& s{ S{}.setAttribute() };

  std::cout << "--\n";

  return 0;
}
```

Output
~S()
--

The temporary dies before `s` goes out of scope (ie. before you use it for something else). If we try to access `s`, we get UB.
There are no collapsed references here, that's a template thing.

> I expect this to work though, and it did

It does, because the temporary dies after your copied it. But you're creating a copy of the object referenced by the last `setAttribute` call (That is, the temporary object).
If you create the object first, ie.

```
Some::Namespace::StructNameOBJCreateInfo info{}; // (1) If it's a type
auto info{ Some::Namespace::StructNameOBJCreateInfo() }; // (2) If it's a func
```

you're getting no copies/moves (Guaranteed) in case 1, and a copy/move/nothing in case 2, depending on the function.

hellmet
January 18, 2020 at 8:53 am · Reply

Yes, the first part is clear, no extension possible on a returned reference. Thanks for the clarification on the templates thing!

> copy/move/nothing in case 2

Hmmm... I see. Need to experiment with some cases then.

Holy shit, it does make a copy! No moves even (other than explicitly calling std::move)!

```
struct S
{
    int a;
    std::string str;

    S& IncreaseInt(){
        ++a;
        return *this;
    }

    S& ConcatString(const std::string& other){
        str += other;
        return *this;
    }

    S()         : a(-1), str("Default") {}
    S(int p)    : a(p), str("Default") {}
    S(int p, const std::string& st)     : a(p), str(st) {}
    S(int p, const std::string&& rv)    : a(p), str(std::move(rv)) {}
    S(const std::string& st)            : a(-1), str(st) {}
    S(std::string&& st)                 : a(-1), str(std::move(st)) {}

    S(const S& other) {
        std::cerr << "Copy constructor\n";
        a = other.a;
        str = other.str;
    }

    S(S&& other) : a(other.a), str(std::move(other.str)) {
        std::cerr << "Move constructor\n";
    }

    S& operator=(const S& other) {
        std::cerr << "Copy assignment\n";
        a = other.a;
        str = other.str;
        return *this;
    }
```

```

    S& operator=(S&& other) {
        std::cerr << "Move assignment\n";
        a = other.a;
        str = std::move(other.str);
        return *this;
    }

    ~S(void){
        std::cout << "~S()\n";
    }
};

int main(void)
{
    const S s { S().IncreaseInt().ConcatString("LoL") };
    std::cerr << s.str << '\n';
    std::cerr << "--\n";
    return 0;
}
//I hope the snippet is complete... in the sense that I cover all constru

Output
----------
Copy constructor
~S()
DefaultLoL
--
~S()
```

I guess I'll default to writing it out in performance-critical-paths then. In other cases, copying should be okay I guess. Why isn't the object being constructed in-place? I know it's easy for the naïve-me to handwave and say oh this would be nice, but from what compilers are capable of, one would think this is a nice optimization to have?

nascardriver
January 19, 2020 at 1:46 am · Reply

Your compiler doesn't know what `ConcatString` returns, it might be *this, it might be some other object. The compiler could follow the entire call chain and trace it back to the temporary (if there are no conditional returns), but that's not required. I don't know if this is allowed as an optimization.
If you use `std::move`, even with full optimizations, you still have a move. I can only recommend again to separate the lines or use likely optimizations:

```
// non-const @s
// No copy/move
S s{};
s.IncreaseInt().ConcatString("LoL");

// If you want @a to be const
// Move/[Optimized away]
const S s{ [](){
    S s{};
    s.IncreaseInt().ConcatString("LoL");
    return s;
  }()
};
```

Note that the parameter list can be omitted in certain situations, I don't think this is mentioned in the lambda lessons.

hellmet
January 19, 2020 at 1:54 am · Reply

Yep, it's optimized away, even in debug builds.

I'll stick to writing it out in hot-code-paths. I'm still wrapping my head around lambdas, but seems like a convoluted syntax for this use case. I'll just write it out, I guess :)
Thank you for the insight! This was a really enlightening exchange!

Charan
December 24, 2019 at 8:36 am · Reply

Hey,Are there pointer references as well?Constant pointer references are extremely useful while traversing a linked list.

nascardriver
December 27, 2019 at 5:58 am · Reply

A const reference to a non-const pointer?

```
int* p{};
int* const & p2{p};
const auto& p3{p}; // same as p2

p2 = nullptr; // No
*p2 = 3; // Ok
```

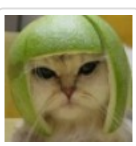
Wallace
October 21, 2019 at 10:47 am · Reply

This page has two sentences that seem so similar that I'm confused. Either I'm not appreciating the difference or they are redundant.

In the first section:
"References to const values are often called 'const references' for short."

In the second section:
"A reference to a const is often called a const reference for short, though this does make for some inconsistent nomenclature with pointers."

Is this an example of a forward declaration of a sentence? ;)



Alex
October 22, 2019 at 6:32 pm · Reply

Lol! More like a violation of "don't repeat yourself". Redundancy removed. Thanks for pointing that out.

helelo

July 22, 2019 at 6:01 am · Reply

If I understand, const in functions are mainly used to print variables?

nascardriver
July 22, 2019 at 6:05 am · Reply

No, that's just what's done here. Whenever you pass a variable by reference and don't modify it, you should mark it as `const`.
If you don't do this, you won't be able to use that function with `const` objects.

RyuuGP
June 4, 2019 at 5:56 pm · Reply

Is there a good reason to assign literal to const reference instead of assign it to actual variable?

nascardriver
June 5, 2019 at 3:42 am · Reply

No

Piyush
January 22, 2019 at 7:12 am · Reply

What is the use of passing structs to the function parameter as const reference if we can do this. ????

```
#include<iostream>
struct employees{
    std::string name;
    int wage;
};
void func(employees emp){
    std::cout<<emp.wage;
}
int main(){
    employees employee={"piyush",260};
    func(employee);
    return 0;
}
```

Thanks a lot.

nascardriver
January 23, 2019 at 9:08 am · Reply

This will create of copy of @employee. Copying is slow. References take up a constant space (4 bytes of 32 bit, 8 bytes on 64 bit). Passing by reference is much faster than passing by value if the value has a size that's bigger than 8 bytes or a size that's not representable by 2^n.

DecSco
July 25, 2019 at 7:31 am · Reply

In addition, say you write a function for giving an employee a raise. If you pass it as a copy, the employee will never get more money, but if you pass it by reference, it works:

```
void giveRaise(Employee e)
{
    e.wage *= 2;
} //here, e goes out of scope

void giveRaise(Employee& e)
{
    e.wage *= 2;
} //e is an alias for what you passed in, so the raise stays.
```

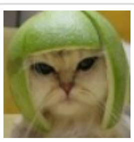
EDIT: ah, saw that you specifically asked for the difference to a const ref. Then this does not apply.

Boteomap2
December 8, 2018 at 7:03 pm · Reply

Hello Alex/nascardriver

->References to r-values extend the lifetime of the referenced value
I Can't see any difference about using reference and non-reference

```
int main()
{
    int x = 3 + 2;
    std::cout << x;
}
```



Alex
December 9, 2018 at 1:46 pm · Reply

You're not using a reference in that example.

```
int main()
{
    std::cout << 3; // the lifetime of 3 is just this statement

    const int &ref = 3; // the lifetime of 3 is now extended to the lifetime of the re
    std::cout << ref; // so we can use it here
}
```

It's kind of silly to do this with a literal -- it's more commonly used with anonymous objects.

Chris
December 8, 2018 at 4:23 am · Reply

There is so many const positions in c++ that's insane to sum it up:

int a {5} = open variable
const int b {5} = const variable
int *c {&a} = open pointer to open variable
const int *d {&a} = open pointer to closed variable
const int const *e {&a} = closed pointer to closed variable
int &f {a} = works

int &g {b} = error because reference not const
const int &h {b} = works because reference now const

But why does a reference work on a literal like 5 when it's const? There is still no memory address for that isn't it?

Alex
December 8, 2018 at 7:37 pm · Reply

When you initialize a const reference with a literal, the compiler likely implicitly defining an anonymous object to hold value 5 and then setting &ref to point at that.

Pikan Ghosh
December 2, 2018 at 11:03 am · Reply

Thank you for the wonderful website and thank you in advance for replying my query.

Scenario 1:

```
#include <iostream>
#include <stdio.h>

int const& retRef() {
    int const &a = 5;
    printf("In Function Address: %p\n", &a);
    return a;
}

int main()
{
    int const& k = retRef();
    printf("Func Address: %p\n", &retRef());
    printf("Address: %p\n", &k);
    printf("Value: %d\n", k);
    return 0;
}
```

Output:

In Function Address: 0x7ffdc9a67d9c
In Function Address: 0x7ffdc9a67d9c
Func Address: 0x7ffdc9a67d9c
Address: 0x7ffdc9a67d9c
Value: 5

==============================
Scenario 2:

```
#include <iostream>
#include <stdio.h>

int const* retRef() {
    int const &a = 5;
    printf("In Function Address: %p\n", &a);
    return &a;
}

int main()
{
    int const* k = retRef();
```

```
    printf("Func Address: %p\n", retRef());
    printf("Address: %p\n", k);
    printf("Value: %d\n", *k);
    return 0;
}
```

Output:

In Function Address: 0x7fffa9df3e9c
In Function Address: 0x7fffa9df3e9c
Func Address: 0x7fffa9df3e9c
Address: 0x7fffa9df3e9c
Value: 5

==================================
Scenario 3:

```
#include <iostream>
#include <stdio.h>

char const* retRef() {
    return "Hello";
}

int main()
{
    char const* k = retRef();
    printf("Func Address: %p\n", retRef());
    printf("Address: %p\n", k);
    printf("Value: %s\n", k);
    return 0;
}
```

output:

Func Address: 0x400845
Address: 0x400845
Value: Hello

==

Scenario 4:

```
#include <iostream>
#include <stdio.h>

int const& retRef() {
    return 5;
}

int main()
{
    int const& k = retRef();
    //printf("Func Address: %p\n",&retRef());
    printf("Address: %p\n", &k);
    //printf("Value: %d\n", k);
    return 0;
}
```

Output:

Address: (nil)

* Uncommenting any of the commented printfs result in segmentation fault.

Question:
How can reference or address of local stack variables can be returned in the scenario 1,2 & 3?
Is there any special treatment of scenario 3, or is it same as scenario 1 & 2?
What is the difference between each of the scenarios with scenario 4?

nascardriver
December 3, 2018 at 4:25 am · Reply

All scenarios except for 3 cause undefined behavior. Never return references or pointers to local variables or temporaries. They die at the end of the function. Any access after that is undefined. It might work, it might crash.

> How can reference or address of local stack variables can be returned in the scenario 1,2 & 3?
You can't.

> Is there any special treatment of scenario 3
Yes, C-style strings are special. This is covered in lesson 6.8b (or 6.6).

> What is the difference between each of the scenarios with scenario 4?
You got lucky.

You're writing C. If you know C and are switching to C++ but don't want to read the first couple of chapters, post your code more frequently so someone can point out what to change.

* <stdio.h> is C, <iostream> and <cstdio> are C++
* Copy initialization is C, uniform initialization is C++ (Lesson 2.1)
* printf is C, @std::cout and @std::printf are C++

Pikan Ghosh
December 3, 2018 at 4:35 am · Reply

Thanks for the clarification. Yes I am switching to C++. The reference thing is very new to me. This is why I was trying different things with it. I will post more codes, once tangled. Thanks again.

Pikan Ghosh
December 2, 2018 at 10:49 am · Reply

Scenario 1:

```
#include <iostream>
#include <stdio.h>

int const& retRef() {
  int const &a = 5;
  printf("In Function Address: %p\n", &a);
  return a;
}

int main()
{
  int const& k = retRef();
  printf("Func Address: %p\n", &retRef());
  printf("Address: %p\n", &k);
  printf("Value: %d\n", k);
  return 0;
}
```

Output:

In Function Address: 0x7ffdc9a67d9c
In Function Address: 0x7ffdc9a67d9c
Func Address: 0x7ffdc9a67d9c
Address: 0x7ffdc9a67d9c
Value: 5

==============================
Scenario 2:

```
#include <iostream>
#include <stdio.h>

int const* retRef() {
   int const &a = 5;
   printf("In Function Address: %p\n", &a);
   return &a;
}

int main()
{
   int const* k = retRef();
   printf("Func Address: %p\n", retRef());
   printf("Address: %p\n", k);
   printf("Value: %d\n", *k);
   return 0;
}
```

Output:

In Function Address: 0x7fffa9df3e9c
In Function Address: 0x7fffa9df3e9c
Func Address: 0x7fffa9df3e9c
Address: 0x7fffa9df3e9c
Value: 5

==================================
Scenario 3:

```
#include <iostream>
#include <stdio.h>

char const* retRef() {
   return "Hello";
}

int main()
{
   char const* k = retRef();
   printf("Func Address: %p\n", retRef());
   printf("Address: %p\n", k);
   printf("Value: %s\n", k);
   return 0;
}
```

output:

Func Address: 0x400845
Address: 0x400845
Value: Hello

==

Scenario 4:

```
#include <iostream>
#include <stdio.h>

int const& retRef() {
   return 5;
}

int main()
{
   int const& k = retRef();
   //printf("Func Address: %p\n",&retRef());
   printf("Address: %p\n", &k);
   //printf("Value: %d\n", k);
   return 0;
}
```

Output:

Address: (nil)

* Uncommenting any of the commented printfs result in segmentation fault.

Question: What is the difference between each of the scenarios with scenario 4?

Luffy
November 14, 2018 at 4:25 am · Reply

Should I use this for arrays, I mean const refernce.

nascardriver
November 14, 2018 at 5:54 am · Reply

const yes, reference no, unless you really want to. You'll learn about better ways to pass arrays (std::array) later.

Hanin
October 15, 2018 at 2:22 am · Reply

Hey, what if I passed a class by const reference and then tried to access to one of its methods that DO NOT MODIFY its state ?
I tried this because I needed using getters of a certain class I passed by const.ref. and I got an error.
Does this mean passing a class in this way prevents us from calling its methods ?

```
 void Point ::positionner(const Point& p)
{
   x = p.getAbscisse();
   y = p.getOrdonne();
}
```

Error : Point.cpp:60:22: error: passing 'const Point' as 'this' argument of 'float Point::getAbscisse()' discards qualifiers [-fpermissive]

Thanks in advance for your answers and hard work.

Hanin
October 15, 2018 at 3:44 am · Reply

I think I found the answer. Apparently we have to declare methods that do not alter the class
as constant methods so that the compiler doesn't get suspicious ..

aman singh
October 13, 2018 at 6:21 am · Reply

hi alex,
pls clear my doubt on this

const int &ref=6;
cout<<&ref;
what will be the output ?
will it print adress of 6
as 6 is r value.

nascardriver
October 13, 2018 at 7:58 am · Reply

Hi Aman!

```
const int &ref=6;
```

creates a temporary int with the same lifetime as @ref.

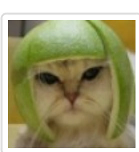
```
&ref
```

return the address of that temporary.

Baschti
September 6, 2018 at 10:44 am · Reply

```
int somefcn()
{
    const int &ref = 2 + 3; // normally the result of 2+3 has expression scope and is destro
    // but because the result is now bound to a reference to a const value...
    std::cout << ref; // we can use it here
}
```

Does ref contain 2 + 3, or 5?



Alex
September 6, 2018 at 3:00 pm · Reply

5

Nigel Booth
August 14, 2018 at 1:50 am · Reply

Hi Alex / Nascardriver,

I just threw together something very simple to help me understand the use of references better (in functions):

```
#include "stdafx.h"
#include <iostream>

void showMe(const int &val)
{
    std::cout << val << std::endl;
}


int main()
{
    int x{ 6 };
    showMe(x);

    int y{ 5 };
    showMe(y);

    int z{ 12 };
    showMe(z);

    return 0;
}
```

This results in the output:

6
5
12

as expected. However, does this mean that instead of using globally initialised variables they can be initialised wherever and accessed through an const reference?

nascardriver
August 14, 2018 at 6:23 am · Reply

Hi Nigel!

You could do so and certainly should in small programs. However, when you're writing bigger projects you'll find yourself with several variables which are used in many places. Passing those around as arguments is tedious, so you'll use global variables or singletons.

Matt
July 20, 2018 at 3:56 pm · Reply

I'm having trouble understanding why this isn't allowed:

```
int &ref3 = 6;
```

but the following IS allowed:

```
const int &ref3 = 6;
```

Could you maybe clarify why this is the case?

How is the following code:

```
int &ref3 = 6;
```

any different from:

```
int x = 6;
int &ref3 = x;
```

?
Thanks for the brilliant website!

nascardriver
July 21, 2018 at 8:08 am · Reply

Hi Matt!

```
int x = 6;
int &ref3 = x;
```

@ref3 is a reference (alias) to @x, when you modify @ref3, you're modifying @x.

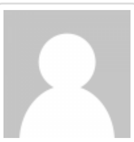
```
int &ref3 = 6;
```

Assuming this worked,
@ref3 is a reference (alias) to 6, when you modify @ref3, you're modifying 6. But 6 cannot be modified.

By saying

```
const int &ref3 = 6;
```

the compiler knows that you're never going to modify @ref3, making the definition legal.

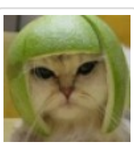


Matt
July 23, 2018 at 5:34 am · Reply

I see, thanks a lot for your help

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

"Rule: Pass non-pointer, non-fundamental data type variables by (const) reference." - u mean non-fundamental data type variables as what variables? Cuz if int, float, char, bool don't match this category, so what is left? Or u mean non-fundamental data type variables as literals?



Alex
October 21, 2017 at 12:14 pm · Reply

Structs (and classes) mainly.

Orfeas
September 23, 2017 at 3:26 am · Reply

Hello Alex,
Thanks for the wonderful tutorials! I came across these two short programs yesterday but I'm not entirely sure how they work.

```
struct A
{
```

```
    int* i;
};

int main()
{
    int i = 2;
    int j = 5;
    A a({ &i });
    cout << i << " " << j << " " << *a.i << endl;
    a.i = &j;
    cout << i << " " << j << " " << *a.i << endl;

    return 0;
}
```

This prints:

2 5 2
5 5 5

Why does the value of variable i change when I point pointer i to variable j's memory adress?

The next one is rather similar:

```
struct A
{
    int& i;
};

int main()
{
    int i = 2;
    int j = 5;
    A a({ i });
    cout << i << " " << j << " " << a.i << endl;
    a.i = j;
    cout << i << " " << j << " " << a.i << endl;

    return 0;
}
```

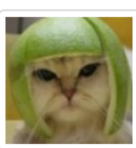
This also prints:

2 5 2
5 5 5

I assume &i is a reference, so does that mean that it doesn't need to be initialized when part of a struct? and then I can't even begin to explain what happens on starting from a.i = j.

Could you help me understand how they work?

Thanks in advance!



Alex
September 25, 2017 at 9:52 pm · Reply

On Visual Studio, I don't get 555 for the second line, I get 255. Sounds like maybe a compiler bug?

Orfeas
September 27, 2017 at 8:08 am · Reply

I read further into the tutorials so I understand pointers and references better, so I understand what happens in program 2, but check this out (I'm also on Visual Studio). Turns out in the first program I actually get an error:

1>c:usersorfeadocumentsvisual studio 2017projectseurydiceeurydiceeurydice.cpp(18): error C2100: illegal indirection (HERE, LINE 10: A a({ i });)
1>Done building project "Eurydice.vcxproj" -- FAILED.

So it was printing the second program instead (as I tried that one first), which still gives me 5 5 5 on the second line.

I tried to think the flow of the second program through:

The program starts executing at main: i = 2, j = 5. i is then sent to a (an A struct) so "a.i" _literally is_ i, since &i is a reference variable. Then i (still 2), j (still 5) and a.i (refers to i, so 2) are printed.

Then I make a.i (literally i) have j's value, so i must have j's value, so cout prints 5 5 5.

So I then tried figuring out what caused the "Illegal indirection" error in the first program.

The program starts executing at main: i = 2 and j = 5, just as before. a (an A struct) contains a pointer, to which &i (i's memory adress) is sent.., meaning a.i points to i.
i (still 2), j (still 5), and *a.i (a dereferenced pointer, pointing to i, so 2).

Then I make a.i point to &j (right?) so i SHOULD stay the same, j as well, but if I were to derefence *a.i, I'll get 5. So I assume this went fine on your machine and the computer correctly printed 2 5 2, 2 5 5. But mine for some reason found an error and ended up reprinting 2 5 2, 5 5 5 from the previous (here the second) program. What should I do about this?

Alex
October 2, 2017 at 9:08 pm · Reply

Well, your first program has a syntax error, in that you're trying to initialize variable a in a way that the compiler doesn't understand. Try this instead:

```
A a = { &i };
```

Then your program should compile and run correctly.

Orfeas
October 2, 2017 at 9:24 pm · Reply

Oh my god, all that for a syntax error XD

Alright thank you so much, you're a great teacher. Have a nice day!

Orfeas

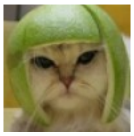
Kumar Santhanam
March 10, 2018 at 9:26 am · Reply

Hi Alex, output is 555 should be second line. because A a({ i }); it means internally i value alias to &a.i, correct?, so second time assigning value j value to it will update in i also. so output is 555 for second line.

Khang

September 13, 2017 at 6:39 am · Reply

Hi, Alex.
At "References to r-values extent the life time of the referenced value"
What is the difference between
[const int &ref=2+3;]
and
[const int value=2+3;].
It seems that they're all extend the life time of "2+3", so is it too obvious
to say the statement above?



Alex
September 13, 2017 at 3:34 pm · Reply

There is no difference. In either case, ref is an const l-value.

FinalDevil
September 3, 2017 at 6:15 am · Reply

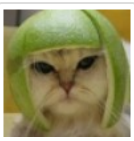
Hi Alex, at the command

```
const int &ref = 2 + 3;
```

, should it be

```
const int&& ref = 2 + 3;
```

? It is reference to rvalue, so we use double ampersand?



Alex
September 5, 2017 at 9:41 am · Reply

There's no need to double-ampersand here. A double-ampersanded reference variable is still treated as an l-value (after all, it has an address). So essentially, in such a case, the second ampersand is ignored.