

What are the differences between a pointer variable and a reference variable?

Asked 16 years, 3 months ago Modified 1 year, 2 months ago

Viewed 1.3m times



What is the difference between a pointer variable and a reference variable?

3933



c++

pointers

reference

c++-faq



Share



Improve this question

Follow

edited Jul 4, 2022 at 20:58



Mateen Ulhaq

27.1k ● 21 ● 117 ● 152

asked Sep 11, 2008 at 20:03



prakash

59.6k ● 25 ● 94 ● 115

44 Answers

Sorted by:

Highest score (default)



1

2

Next



1. A pointer can be re-assigned:

2233



```
int x = 5;
int y = 6;
int *p;
p = &x;
p = &y;
*p = 10;
assert(x == 5);
assert(y == 10);
```

A reference cannot be re-bound, and must be bound at initialization:

```
int x = 5;
int y = 6;
int &q; // error
int &r = x;
```

2. A pointer variable has its own identity: a distinct, visible memory address that can be taken with the unary `&` operator and a certain amount of space that can be measured with the `sizeof` operator. Using those operators on a reference returns a value corresponding to whatever the reference is bound to; the reference's own address and size are invisible. Since the reference assumes the identity of the original variable in this way, it is convenient to think of a reference as another name for the same variable.

```
int x = 0;
int &r = x;
int *p = &x;
int *p2 = &r;

assert(p == p2); // &x == &r
assert(&p != &p2);
```

3. It is possible to create a pointer to a pointer, but not a pointer to a reference.

```
int **pp; // OK, pointer to pointer
int &*pr; // ill-formed, pointer to reference
```

4. It is possible to create an array of pointers, but not an array of references.

```
int *ap[]; // OK, array of pointers
int &ar[]; // ill-formed, array of references
```

5. You can have arbitrarily nested pointers to pointers offering extra levels of indirection. References only offer one level of indirection because references to references [collapse](#).

```
int x = 0;
int y = 0;
int *p = &x;
int *q = &y;
int **pp = &p;

**pp = 2;
pp = &q; // *pp is now q
**pp = 4;

assert(y == 4);
assert(x == 2);
```

6. A pointer can be assigned `nullptr`, whereas a reference must be bound to an existing object. If you try hard enough, you can bind a reference to `nullptr`, but this is [undefined](#) and will not behave consistently.

```

/* the code below is undefined; your compiler may
 * differently, emit warnings, or outright refuse

int &r = *static_cast<int *>(nullptr);

// prints "null" under GCC 10
std::cout
    << (&r != nullptr
        ? "not null" : "null")
    << std::endl;

bool f(int &r) { return &r != nullptr; }

// prints "not null" under GCC 10
std::cout
    << (f(*static_cast<int *>(nullptr))
        ? "not null" : "null")
    << std::endl;

```

You can, however, have a reference to a pointer whose value is `nullptr`.

7. Pointers are [Contiguous Iterators](#) (of an array). You can use `++` to go to the next item that a pointer is pointing to, and `+ 4` to go to the 5th element.
8. A pointer needs to be dereferenced with `*` to access the object it points to, whereas a reference can be used directly. A pointer to a class/struct uses `->` to access its members whereas a reference uses a `.`.
9. Const references and rvalue references can be bound to temporaries (see [temporary materialization](#)). Pointers cannot (not without some indirection):

```

const int &x = int(12); // legal C++
int *y = &int(12); // illegal to take the address

```

This makes `const &` more convenient to use in argument lists and so forth.

Share Improve this answer

edited Oct 5, 2023 at 11:30

Follow

community wiki

29 revs, 15 users 37%

Brian R. Bondy



551

What's a C++ reference (*for C programmers*)



A *reference* can be thought of as a *constant pointer* (not to be confused with a pointer to a constant value!) with automatic indirection, ie the compiler will apply the `*` operator for you.



All references must be initialized with a non-null value or compilation will fail. It's neither possible to get the address of a reference - the address operator will return the address of the referenced value instead - nor is it possible to do arithmetics on references.

C programmers might dislike C++ references as it will no longer be obvious when indirection happens or if an argument gets passed by value or by pointer without looking at function signatures.

C++ programmers might dislike using pointers as they are considered unsafe - although references aren't really any safer than constant pointers except in the most trivial cases - lack the convenience of automatic indirection and carry a different semantic connotation.

Consider the following statement from the [C++ FAQ](#):

Even though a reference is often implemented using an address in the underlying assembly language, please do *not* think of a reference as a funny looking pointer to an object. A reference *is* the object. It is not a pointer to the object, nor a copy of the object. It *is* the object.

But if a reference *really* were the object, how could there be dangling references? In unmanaged languages, it's impossible for references to be any 'safer' than pointers - there generally just isn't a way to reliably alias values across scope boundaries!

Why I consider C++ references useful

Coming from a C background, C++ references may look like a somewhat silly concept, but one should still use them instead of pointers where possible: Automatic indirection *is* convenient, and references become especially useful when dealing with [RAII](#) - but not

because of any perceived safety advantage, but rather because they make writing idiomatic code less awkward.

RAII is one of the central concepts of C++, but it interacts non-trivially with copying semantics. Passing objects by reference avoids these issues as no copying is involved. If references were not present in the language, you'd have to use pointers instead, which are more cumbersome to use, thus violating the language design principle that the best-practice solution should be easier than the alternatives.

Share Improve this answer

Follow

edited Jul 7, 2015 at 21:27



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Feb 27, 2009 at 21:26



Christoph

169k ● 36 ● 186 ● 241

23 @kriss: No, you can also get a dangling reference by returning an automatic variable by reference. – Ben Voigt Nov 2, 2010 at 6:14

20 @kriss: It's virtually impossible for a compiler to detect in the general case. Consider a member function that returns a reference to a class member variable: that's safe and should not be forbidden by the compiler. Then a caller that has an automatic instance of that class, calls that member function, and returns the reference. Presto: dangling reference. And yes, it's going to cause trouble, @kriss: that's my point. Many people claim that an advantage of references over pointers is that references are always valid, but it just isn't so. – Ben Voigt Nov 2, 2010 at 13:15

7 @kriss: No, a reference into an object of automatic storage duration is very different from a temporary object. Anyway, I was just providing a counter-example to your statement that you can only get an invalid reference by dereferencing an invalid pointer. Christoph is correct -- references are not any safer than pointers, a program which uses references exclusively can still break type safety. – [Ben Voigt](#) Nov 2, 2010 at 15:15

11 References are not a kind of pointer. They are a new name for an existing object. – [catphive](#) Jul 20, 2011 at 1:28

29 @catphive: true if you go by language semantics, not true if you actually look at the implementation; C++ is a far more 'magical' language than C, and if you remove the magic from references, you end up with a pointer – [Christoph](#) Jul 23, 2011 at 9:07



232



If you want to be really pedantic, there is one thing you can do with a reference that you can't do with a pointer: extend the lifetime of a temporary object. In C++ if you bind a const reference to a temporary object, the lifetime of that object becomes the lifetime of the reference.

```
std::string s1 = "123";
std::string s2 = "456";

std::string s3_copy = s1 + s2;
const std::string& s3_reference = s1 + s2;
```

In this example `s3_copy` copies the temporary object that is a result of the concatenation. Whereas `s3_reference` in essence becomes the temporary object. It's really a

reference to a temporary object that now has the same lifetime as the reference.

If you try this without the `const` it should fail to compile. You cannot bind a non-const reference to a temporary object, nor can you take its address for that matter.

Share Improve this answer

edited Oct 8, 2010 at 16:52

Follow



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Sep 11, 2008 at 21:43



Matt Price

45.3k ● 9 ● 39 ● 44

8 but whats the use case for this ? – Ahmad Mushtaq Oct 22, 2009 at 14:10

26 Well, `s3_copy` will create a temporary and then copy construct it into `s3_copy` whereas `s3_reference` directly uses the temporary. Then to be really pedantic you need to look at the Return Value Optimization whereby the compiler is allowed to elide the copy construction in the first case.
– Matt Price Oct 22, 2009 at 18:14

7 @digitalSurgeon: The magic there is quite powerful. The object lifetime is extended by the fact of the `const &` binding, and only when the reference goes out of scope the destructor of the *actual* referenced type (as compared to the reference type, that could be a base) is called. Since it is a reference, no slicing will take place in between.
– David Rodríguez - dribeas Jan 14, 2010 at 17:06

11 Update for C++11: last sentence should read "You cannot bind a non-const lvalue reference to a temporary" because you *can* bind a non-const *rvalue* reference to a temporary,

and it has the same lifetime-extending behaviour. – [Oktalist](#)

Nov 10, 2013 at 20:14

-
- 9 @AhmadMushtaq: The key use of this is **derived classes**. If there is no inheritance involved, you might as well use value semantics, which will be cheap or free due to RVO/move construction. But if you have `Animal x = fast ? getHare() : getTortoise()` then `x` will face the classic slicing problem, while `Animal& x = ...` will work correctly. – [Arthur Tacca](#) Nov 2, 2017 at 11:04
-



177



Apart from syntactic sugar, a reference is a `const` pointer (not pointer to a `const`). You must establish what it refers to when you declare the reference variable, and you cannot change it later.

Update: now that I think about it some more, there is an important difference.

A const pointer's target can be replaced by taking its address and using a const cast.

A reference's target cannot be replaced in any way short of UB.

This should permit the compiler to do more optimization on a reference.

Share Improve this answer

[edited Apr 12, 2018 at 15:59](#)

Follow

answered Sep 11, 2008 at 20:07



-
- 18 I think this is the best answer by far. Others talk about references and pointers like they are different beasts and then lay out how they differ in behavior. It doesn't make things any easier imho. I've always understood references as being a `T* const` with different syntactic sugar (that happens to eliminate a lot of `*` and `&` from your code).
– [Carlo Wood](#) Jan 10, 2017 at 1:34
-
- 10 "A const pointer's target can be replaced by taking its address and using a const cast." Doing so is undefined behavior. See stackoverflow.com/questions/25209838/... for details. – [dgnuff](#) Jun 22, 2018 at 4:51
-
- 2 Trying to change either the referent of a reference or the value of a const pointer (or any const scalar) is equality illegal. What you can do: remove a const qualification that was added by implicit conversion: `int i; int const *pci = &i; /* implicit conv to const int* */ int *pi = const_cast<int*>(pci);` is OK. – [curiousguy](#) Jun 29, 2018 at 16:00
-
- 3 The difference here is UB versus literally impossible. There is no syntax in C++ that would let you change what reference points at. – user3458 Jun 29, 2018 at 16:03
-
- 2 Not impossible, harder, you can just access the memory area of the pointer that is modeling that reference and change its content. That can certainly be done.
– [Nicolas Bousquet](#) Nov 17, 2018 at 21:39
-



Contrary to popular opinion, it is possible to have a reference that is NULL.

139



```
int * p = NULL;  
int & r = *p;  
r = 1; // crash! (if you're lucky)
```



Granted, it is much harder to do with a reference - but if you manage it, you'll tear your hair out trying to find it. References are *not* inherently safe in C++!

Technically this is an **invalid reference**, not a null reference. C++ doesn't support null references as a concept as you might find in other languages. There are other kinds of invalid references as well. *Any* invalid reference raises the spectre of **undefined behavior**, just as using an invalid pointer would.

The actual error is in the dereferencing of the NULL pointer, prior to the assignment to a reference. But I'm not aware of any compilers that will generate any errors on that condition - the error propagates to a point further along in the code. That's what makes this problem so insidious. Most of the time, if you dereference a NULL pointer, you crash right at that spot and it doesn't take much debugging to figure it out.

My example above is short and contrived. Here's a more real-world example.

```

class MyClass
{
    ...
    virtual void DoSomething(int,int,int,int,int);
};

void Foo(const MyClass & bar)
{
    ...
    bar.DoSomething(i1,i2,i3,i4,i5); // crash occurs
    violation - obvious why?
}

MyClass * GetInstance()
{
    if (somecondition)
        return NULL;
    ...
}

MyClass * p = GetInstance();
Foo(*p);

```

I want to reiterate that the only way to get a null reference is through malformed code, and once you have it you're getting undefined behavior. It **never** makes sense to check for a null reference; for example you can try `if(&bar==NULL)...` but the compiler might optimize the statement out of existence! A valid reference can never be NULL so from the compiler's view the comparison is always false, and it is free to eliminate the `if` clause as dead code - this is the essence of undefined behavior.

The proper way to stay out of trouble is to avoid dereferencing a NULL pointer to create a reference. Here's an automated way to accomplish this.

```

template<typename T>
T& deref(T* p)
{
    if (p == NULL)
        throw std::invalid_argument(std::string("NULL"));
    return *p;
}

MyClass * p = GetInstance();
Foo(deref(p));

```

For an older look at this problem from someone with better writing skills, see [Null References](#) from Jim Hyslop and Herb Sutter.

For another example of the dangers of dereferencing a null pointer see [Exposing undefined behavior when trying to port code to another platform](#) by Raymond Chen.

Share Improve this answer

edited Jan 8, 2018 at 22:57

Follow

answered Sep 11, 2008 at 21:06



Mark Ransom

308k ● 44 ● 416 ● 647

71 The code in question contains undefined behavior. Technically, you cannot do anything with a null pointer except set it, and compare it. Once your program invokes undefined behavior, it can do anything, including appearing to work correctly until you are giving a demo to the big boss.
– [KeithB](#) Sep 12, 2008 at 16:00

12 mark has a valid argument. the argument that a pointer could be NULL and you therefor have to check is not real either: if

you say a function requires non-NULL, then the caller has to do that. so if the caller doesn't he is invoking undefined behavior. just like mark did with the bad reference

– [Johannes Schaub - litb](#) Feb 27, 2009 at 21:14

18 The description is erroneous. This code might or might not create a reference that is NULL. Its behavior is undefined. It might create a perfectly valid reference. It might fail to create any reference at all. – [David Schwartz](#) Aug 20, 2011 at 11:41

14 @David Schwartz, if I were talking about the way things had to work according to the standard, you'd be correct. But that's *not* what I'm talking about - I'm talking about actual observed behavior with a very popular compiler, and extrapolating based on my knowledge of typical compilers and CPU architectures to what will *probably* happen. If you believe references to be superior to pointers because they're safer and don't consider that references can be bad, you'll be stumped by a simple problem someday just as I was.
– [Mark Ransom](#) Aug 22, 2011 at 2:11

8 Dereferencing a null pointer is wrong. Any program that does that, even to initialize a reference is wrong. If you are initializing a reference from a pointer you should always check that the pointer is valid. Even if this succeeds the underlying object may be deleted at any time leaving the reference to refer to non-existing object, right? What you are saying is good stuff. I think the real issue here is that reference does NOT need to be checked for "nullness" when you see one and pointer should be, at minimum, asserted.
– [t0rakka](#) Mar 22, 2017 at 14:07



You forgot the most important part:

130

member-access with pointers uses `->`

member-access with references uses `.`



`foo.bar` is *clearly* superior to `foo->bar` in the same way that [vi](#) is *clearly* superior to [Emacs](#) :-)



Share Improve this answer

edited Jul 7, 2015 at 21:21

Follow



[Peter Mortensen](#)

31.6k ● 22 ● 109 ● 133

answered Sep 11, 2008 at 22:10



[Orion Edwards](#)

123k ● 66 ● 245 ● 339

-
- 6 @Orion Edwards >member-access with pointers uses `->` >member-access with references uses `.` This is not 100% true. You can have a reference to a pointer. In this case you would access members of de-referenced pointer using `->`
- ```
struct Node { Node *next; }; Node *first; // p is a reference to a pointer
void foo(Node*&p) { p->next = first; } Node *bar = new Node; foo(bar);
```
- OP: Are you familiar with the concepts of rvalues and lvalues? – [user6105](#) Sep 12, 2008 at 12:57
- 
- 4 Smart Pointers have both `.` (methods on smart pointer class) and `->` (methods on underlying type). – [JBRWilkinson](#) Apr 9, 2014 at 9:11
- 
- 3 why is that `.` and `->` has something to do with vi and emacs :) – [tdao](#) Aug 21, 2016 at 0:45
- 
- 16 @artM - it was a joke, and probably doesn't make sense to non-native english speakers. My apologies. To explain, whether vi is better than emacs is entirely subjective. Some people think vi is far superior, and others think the exact opposite. Similarly, I think using `.` is better than using `->`, but just like vi vs emacs, it's entirely subjective and you can't prove anything – [Orion Edwards](#) Aug 25, 2016 at 21:41 ✎
- 
- 6 @user3840170 it's a joke :-) This may be hard to believe, but back in the early days of stack overflow (note the answer is



dated september 2008) people used to have a sense of humour and it was a nice site to hang out on. To be honest it's hard to believe that some pedant hasn't marked this answer as deleted by now – [Orion Edwards](#) Jan 4, 2021 at 23:12

---



References are very similar to pointers, but they are specifically crafted to be helpful to optimizing compilers.

89



- References are designed such that it is substantially easier for the compiler to trace which reference aliases which variables. Two major features are very important: no "reference arithmetic" and no reassigning of references. These allow the compiler to figure out which references alias which variables at compile time.
- References are allowed to refer to variables which do not have memory addresses, such as those the compiler chooses to put into registers. If you take the address of a local variable, it is very hard for the compiler to put it in a register.

As an example:

```
void maybeModify(int& x); // may modify x in some way

void hurtTheCompilersOptimizer(short size, int array[])
{
 // This function is designed to do something parti
 // for optimizers. It will constantly call maybeMo
 // adding array[1] to array[2]..array[size-1]. The
 // do this, other than to demonstrate the power of
 for (int i = 2; i < (int)size; i++) {
```

```

 maybeModify(array[0]);
 array[i] += array[1];
 }
}

```

An optimizing compiler may realize that we are accessing `a[0]` and `a[1]` quite a bunch. It would love to optimize the algorithm to:

```

void hurtTheCompilersOptimizer(short size, int array[])
{
 // Do the same thing as above, but instead of acce
 // all the time, access it once and store the resu
 // which is much faster to do arithmetic with.
 register int a0 = a[0];
 register int a1 = a[1]; // access a[1] once
 for (int i = 2; i < (int)size; i++) {
 maybeModify(a0); // Give maybeModify a referen
 array[i] += a1; // Use the saved register val
 }
 a[0] = a0; // Store the modified a[0] back into th
}

```

To make such an optimization, it needs to prove that nothing can change `array[1]` during the call. This is rather easy to do. `i` is never less than 2, so `array[i]` can never refer to `array[1]`. `maybeModify()` is given `a0` as a reference (aliasing `array[0]`). Because there is no "reference" arithmetic, the compiler just has to prove that `maybeModify` never gets the address of `x`, and it has proven that nothing changes `array[1]`.

It also has to prove that there are no ways a future call could read/write `a[0]` while we have a temporary register copy of it in `a0`. This is often trivial to prove, because in

many cases it is obvious that the reference is never stored in a permanent structure like a class instance.

Now do the same thing with pointers

```
void maybeModify(int* x); // May modify x in some way

void hurtTheCompilersOptimizer(short size, int array[])
{
 // Same operation, only now with pointers, making
 // optimization trickier.
 for (int i = 2; i < (int)size; i++) {
 maybeModify(&array[0]);
 array[i] += array[1];
 }
}
```

The behavior is the same; only now it is much harder to prove that maybeModify does not ever modify array[1], because we already gave it a pointer; the cat is out of the bag. Now it has to do the much more difficult proof: a static analysis of maybeModify to prove it never writes to  $\&x + 1$ . It also has to prove that it never saves off a pointer that can refer to array[0], which is just as tricky.

Modern compilers are getting better and better at static analysis, but it is always nice to help them out and use references.

Of course, barring such clever optimizations, compilers will indeed turn references into pointers when needed.

EDIT: Five years after posting this answer, I found an actual technical difference where references are different

than just a different way of looking at the same addressing concept. References can modify the lifespan of temporary objects in a way that pointers cannot.

```
F createF(int argument);

void extending()
{
 const F& ref = createF(5);
 std::cout << ref.getArgument() << std::endl;
};
```

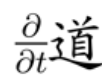
Normally temporary objects such as the one created by the call to `createF(5)` are destroyed at the end of the expression. However, by binding that object to a reference, `ref`, C++ will extend the lifespan of that temporary object until `ref` goes out of scope.

Share Improve this answer

edited Oct 8, 2018 at 21:51

Follow

answered Sep 1, 2013 at 3:44



[Cort Ammon](#)

10.8k ● 33 ● 47


---

True, the body does have to be visible. However, determining that `maybeModify` does not take the address of anything related to `x` is substantially easier than proving that a bunch of pointer arithmetic does not occur. – [Cort Ammon](#) Sep 11, 2013 at 4:27

---

I believe the optimizer already does that "a bunch of pointer arithmetic does not occur" check for a bunch of other reasons. – [Ben Voigt](#) Sep 11, 2013 at 4:32

---

"References are very similar to pointers" - semantically, in appropriate contexts - but in terms of generated code, only in some implementations and not through any definition/requirement. I know you've pointed this out, and I don't disagree with any of your post in practical terms, but we have too many problems already with people reading too much into shorthand descriptions like 'references are like/usually implemented as pointers'. – [underscore\\_d](#) Oct 11, 2015 at 17:31 

- 
- 1 I have a feeling that someone wrongly flagged as obsolete a comment along the lines of `void maybeModify(int& x) { 1[&x]++; }`, which the other comments above are discussing – [Ben Voigt](#) Dec 3, 2015 at 23:28
- 



Actually, a reference is not really like a pointer.

**83**



A compiler keeps "references" to variables, associating a name with a memory address; that's its job to translate any variable name to a memory address when compiling.



When you create a reference, you only tell the compiler that you assign another name to the pointer variable; that's why references cannot "point to null", because a variable cannot be, and not be.

Pointers are variables; they contain the address of some other variable, or can be null. The important thing is that a pointer has a value, while a reference only has a variable that it is referencing.

Now some explanation of real code:

```
int a = 0;
int& b = a;
```

Here you are not creating another variable that points to `a`; you are just adding another name to the memory content holding the value of `a`. This memory now has two names, `a` and `b`, and it can be addressed using either name.

```
void increment(int& n)
{
 n = n + 1;
}

int a;
increment(a);
```

When calling a function, the compiler usually generates memory spaces for the arguments to be copied to. The function signature defines the spaces that should be created and gives the name that should be used for these spaces. Declaring a parameter as a reference just tells the compiler to use the input variable memory space instead of allocating a new memory space during the method call. It may seem strange to say that your function will be directly manipulating a variable declared in the calling scope, but remember that when executing compiled code, there is no more scope; there is just plain flat memory, and your function code could manipulate any variables.

Now there may be some cases where your compiler may not be able to know the reference when compiling, like when using an extern variable. So a reference may or may not be implemented as a pointer in the underlying code. But in the examples I gave you, it will most likely not be implemented with a pointer.

Share Improve this answer

edited Jul 7, 2015 at 21:25

Follow



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Sep 19, 2008 at 12:23



Vincent Robert

36.1k ● 15 ● 83 ● 122

---

2 A reference is a reference to l-value, not necessarily to a variable. Because of that, it's much closer to a pointer than to a real alias (a compile-time construct). Examples of expressions that can be referenced are `*p` or even `*p++` – user3458 Mar 2, 2009 at 16:27

---

6 Right, I was just pointing the fact that a reference may not always push a new variable on the stack the way a new pointer will. – Vincent Robert Mar 3, 2009 at 20:36

---

4 @VincentRobert: It will act the same as a pointer... if the function is inlined, both reference and pointer will be optimized away. If there's a function call, the address of the object will need to be passed to the function. – Ben Voigt Feb 13, 2012 at 23:08

---

1 `int *p = NULL; int &r=*p; reference pointing to NULL; if(r){} -> boom ;)` – uss Oct 4, 2015 at 12:37

---

2 This focus on the compile stage seems nice, until you remember that references can be passed around at runtime,

at which point static aliasing goes out of the window. (And then, references are *usually* implemented as pointers, but the standard doesn't require this method.) – [underscore\\_d](#) Oct 11, 2015 at 17:35

---



A reference can never be `NULL` .

48

Share Improve this answer

edited Jun 1, 2014 at 22:37

Follow



Cole Tobin

9,424 ● 15 ● 51 ● 77

answered Sep 11, 2008 at 20:12



RichS

13 See Mark Ransom's answer for a counter-example. This is the most often asserted myth about references, but it is a myth. The only guarantee that you have by the standard is, that you immediately have UB when you have a NULL reference. But that is akin to saying "This car is safe, it can never get off the road. (We don't take any responsibility for what may happen if you steer it off the road anyway. It might just explode.)" – [cmaster - reinstate monica](#) Jun 13, 2014 at 11:36

20 @cmaster: **In a valid program**, a reference cannot be null. But a pointer can. This is not a myth, this is a fact. – [user541686](#) Aug 8, 2014 at 4:42

11 @Mehrdad Yes, valid programs stay on the road. But there is no traffic barrier to enforce that your program actually does. Large parts of the road are actually missing markings. So it's extremely easy to get off the road at night. And it is crucial for debugging such bugs that you *know* this can happen: the null reference can propagate before it crashes your program,



just like a null pointer can. And when it does you have code like `void Foo::bar() { virtual_baz(); }` that segfaults. If you are not aware that references may be null, you can't trace the null back to its origin.

– [cmaster - reinstate monica](#) Dec 29, 2014 at 10:43 

---

4 `int *p = NULL; int &r=*p; reference pointing to NULL; if(r){ -> boOm ;}` – [uss](#) Oct 4, 2015 at 12:43

---

13 @sree `int &r=*p;` is undefined behavior. At that point, you don't have a "reference pointing to NULL," you have a program that *can no longer be reasoned about at all*.  
– [cdhowie](#) Mar 28, 2017 at 18:46

---



38

There is a semantic difference that may appear esoteric if you are not familiar with studying computer languages in an abstract or even academic fashion.



At the highest-level, the idea of references is that they are transparent "aliases". Your computer may use an address to make them work, but you're not supposed to worry about that: you're supposed to think of them as "just another name" for an existing object and the syntax reflects that. They are stricter than pointers so your compiler can more reliably warn you when you about to create a dangling reference, than when you are about to create a dangling pointer.



Beyond that, there are of course some practical differences between pointers and references. The syntax to use them is obviously different, and you cannot "re-seat" references, have references to nothingness, or have pointers to references.

Share Improve this answer

answered Oct 29, 2014 at 17:17

Follow



Lightness Races in Orbit

385k ● 77 ● 663 ● 1.1k



36



While both references and pointers are used to indirectly access another value, there are two important differences between references and pointers. The first is that a reference always refers to an object: It is an error to define a reference without initializing it. The behavior of assignment is the second important difference: Assigning to a reference changes the object to which the reference is bound; it does not rebind the reference to another object. Once initialized, a reference always refers to the same underlying object.

Consider these two program fragments. In the first, we assign one pointer to another:

```
int ival = 1024, ival2 = 2048;
int *pi = &ival, *pi2 = &ival2;
pi = pi2; // pi now points to ival2
```

After the assignment, `ival`, the object addressed by `pi` remains unchanged. The assignment changes the value of `pi`, making it point to a different object. Now consider a similar program that assigns two references:

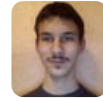
```
int &ri = ival, &ri2 = ival2;
ri = ri2; // assigns ival2 to ival
```

This assignment changes ival, the value referenced by ri, and not the reference itself. After the assignment, the two references still refer to their original objects, and the value of those objects is now the same as well.

Share Improve this answer

Follow

edited Oct 23, 2015 at 23:16



Андрей Беньковский

3,892 ● 7 ● 24 ● 35

answered May 20, 2011 at 19:26



Kunal Vyas

1,579 ● 1 ● 23 ● 40

---

"a reference always refers to an object" is just completely false – [Ben Voigt](#) Jul 21, 2017 at 18:29

---



30



A reference is an alias for another variable whereas a pointer holds the memory address of a variable.

References are generally used as function parameters so that the passed object is not the copy but the object itself.

```
void fun(int &a, int &b); // A common usage of ref
int a = 0;
int &b = a; // b is an alias for a. Not so common
```

Share Improve this answer

Follow

answered Jan 1, 2013 at 17:45



fatma.ekici

2,827 ● 5 ● 31 ● 31

---

## The direct answer



What is a reference in C++? Some specific instance of type that **is not an object type**.

27



What is a pointer in C++? Some specific instance of type that **is an object type**.



From [the ISO C++ definition of object type](#):



An *object* type is a (possibly cv-qualified) type that is not a function type, not a reference type, and not cv void.

It may be important to know, object type is a top-level category of the type universe in C++. Reference is also a top-level category. **But pointer is not.**

Pointers and references are mentioned together [in the context of compound type](#). This is basically due to the nature of the declarator syntax inherited from (and extended) C, which has no references. (Besides, there are more than one kind of declarator of references since C++ 11, while pointers are still "untyped": `& + &&` vs. `*`.) So drafting a language specific by "extension" with similar style of C in this context is somewhat reasonable. (I will still argue that the syntax of declarators wastes the syntactic expressiveness *a lot*, makes both human users and implementations frustrating. Thus, all of them are not qualified to be *built-in* in a new language design. This is a totally different topic about PL design, though.)

Otherwise, it is insignificant that pointers can be qualified as a specific sorts of types with references together. They simply share too few common properties besides the syntax similarity, so there is no need to put them together in most cases.

Note the statements above only mentions "pointers" and "references" as types. There are some interested questions about their instances (like variables). There also come too many misconceptions.

The differences of the top-level categories can already reveal many concrete differences not tied to pointers directly:

- Object types can have top-level `cv` qualifiers. References cannot.
- Variable of object types do occupy storage as per [the abstract machine](#) semantics. Reference do not necessary occupy storage (see the section about misconceptions below for details).
- ...

A few more special rules on references:

- [Compound declarators are more restrictive on references.](#)
- References can [collapse](#).
  - Special rules on `&&` parameters (as the "forwarding references") based on reference

collapsing during template parameter deduction  
allow ["perfect forwarding"](#) of parameters.

- References have special rules in initialization. The lifetime of variable declared as a reference type can be different to ordinary objects via extension.
  - BTW, a few other contexts like initialization involving `std::initializer_list` follows some similar rules of reference lifetime extension. It is another can of worms.
- ...

## The misconceptions

### [Syntactic sugar](#)

I know references are syntactic sugar, so code is easier to read and write.

Technically, this is plain wrong. References are not syntactic sugar of any other features in C++, because they cannot be exactly replaced by other features without any semantic differences.

(Similarly, *lambda-expressions* are *not* syntactic sugar of any other features in C++ because it cannot be precisely simulated with "unspecified" properties like [the declaration order of the captured variables](#), which may be important because the initialization order of such variables can be significant.)

C++ only has a few kinds of syntactic sugars in this strict sense. One instance is (inherited from C) the built-in (non-overloaded) operator `[]`, which [is defined exactly having same semantic properties of specific forms of combination over built-in operator unary `\*` and binary `+`](#).

## Storage

*So, a pointer and a reference both use the same amount of memory.*

The statement above is simply wrong. To avoid such misconceptions, look at the ISO C++ rules instead:

From [\[intro.object\]/1](#):

... An object occupies a region of storage in its period of construction, throughout its lifetime, and in its period of destruction. ...

From [\[dcl.ref\]/4](#):

It is unspecified whether or not a reference requires storage.

Note these are *semantic* properties.

## Pragmatics

Even that pointers are not qualified enough to be put together with references in the sense of the language design, there are still some arguments making it debatable to make choice between them in some other contexts, for example, when making choices on parameter types.

But this is not the whole story. I mean, there are more things than pointers vs references you have to consider.

If you don't have to stick on such over-specific choices, in most cases the answer is short: **you do not have the necessity to use pointers, so you don't**. Pointers are usually bad enough because they imply too many things you don't expect and they will rely on too many implicit assumptions undermining the maintainability and (even) portability of the code. **Unnecessarily relying on pointers is definitely a bad style and it should be avoided in the sense of modern C++**. Reconsider your purpose and you will finally find that **pointer is the feature of last sorts** in most cases.

- Sometimes the language rules explicitly require specific types to be used. If you want to use these features, obey the rules.
  - Copy constructors require specific types of cv- & reference type as the 1st parameter type. (And usually it should be `const` qualified.)
  - Move constructors require specific types of cv- && reference type as the 1st parameter type. (And usually there should be no qualifiers.)



- Specific overloads of operators require reference or non reference types. For example:
  - Overloaded `operator=` as special member functions requires reference types similar to 1st parameter of copy/move constructors.
  - Postfix `++` requires dummy `int`.
  - ...
- If you know pass-by-value (i.e. using non-reference types) is sufficient, use it directly, particularly when using an implementation supporting C++17 mandated copy elision. (**Warning:** However, to **exhaustively** reason about the necessity can be [very complicated](#).)
- If you want to operate some handles with ownership, use smart pointers like `unique_ptr` and `shared_ptr` (or even with homebrew ones by yourself if you require them to be *opaque*), rather than raw pointers.
- If you are doing some iterations over a range, use iterators (or some ranges which are not provided by the standard library yet), rather than raw pointers unless you are convinced raw pointers will do better (e.g. for less header dependencies) in very specific cases.
- If you know pass-by-value is sufficient and you want some explicit nullable semantics, use wrapper like `std::optional`, rather than raw pointers.
- If you know pass-by-value is not ideal for the reasons above, and you don't want nullable semantics, use

{lvalue, rvalue, forwarding}-references.

- Even when you do want semantics like traditional pointer, there are often something more appropriate, like `observer_ptr` in Library Fundamental TS.

The only exceptions cannot be worked around in the current language:

- When you are implementing smart pointers above, you may have to deal with raw pointers.
- Specific language-interoperation routines require pointers, like `operator new`. (However, `cv-void*` is still quite different and safer compared to the ordinary object pointers because it rules out unexpected pointer arithmetics unless you are relying on some non conforming extension on `void*` like GNU's.)
- Function pointers can be converted from lambda expressions without captures, while function references cannot. You have to use function pointers in non-generic code for such cases, even you deliberately do not want nullable values.

So, in practice, the answer is so obvious: **when in doubt, avoid pointers**. You have to use pointers only when there are very explicit reasons that nothing else is more appropriate. Except a few exceptional cases mentioned above, such choices are almost always not purely C++-specific (but likely to be language-implementation-specific). Such instances can be:

- You have to serve to old-style (C) APIs.

- You have to meet the ABI requirements of specific C++ implementations.
- You have to interoperate at runtime with different language implementations (including various assemblies, language runtime and FFI of some high-level client languages) based on assumptions of specific implementations.
- You have to improve efficiency of the translation (compilation & linking) in some extreme cases.
- You have to avoid symbol bloat in some extreme cases.

## Language neutrality caveats

If you come to see the question via [some Google search result \(not specific to C++\)](#), this is very likely to be the wrong place.

References in C++ is quite "odd", as it is essentially not first-class: [they will be treated as the objects or the functions being referred to](#) so they have no chance to support some first-class operations like being the left operand of [the member access operator](#) independently to the type of the referred object. Other languages may or may not have similar restrictions on their references.

References in C++ will likely not preserve the meaning across different languages. For example, references in general do not imply nonnull properties on values like they in C++, so such assumptions may not work in some

other languages (and you will find counterexamples quite easily, e.g. Java, C#, ...).

There can still be some common properties among references in different programming languages in general, but let's leave it for some other questions in SO.

(A side note: the question may be significant earlier than any "C-like" languages are involved, like [ALGOL 68 vs. PL/I.](#))

Share Improve this answer

edited Mar 1, 2019 at 12:43

Follow

answered Feb 17, 2019 at 7:37



FrankHB

2,495 ● 28 ● 21

---

Nitpick: "use iterators rather than pointers" pointers model the iterator concept. E.g. a conforming implementation can choose `std::vector<T>::iterator` to be an alias for `T*` – [Caleth](#) Sep 30, 2022 at 8:35

---

The subscript operator is not syntactical sugar for `*(a + b)`. There are subtle differences in regards to temporary materialization, value categories, etc. I'm honestly not sure if C++ has anything that is truly syntax sugar with no asterisk attached. – [Jan Schultke](#) Sep 28, 2023 at 0:16

---



It doesn't matter how much space it takes up since you can't actually see any side effect (without executing code) of whatever space it would take up.

23



On the other hand, one major difference between references and pointers is that temporaries assigned to const references live until the const reference goes out of scope.

For example:

```
class scope_test
{
public:
 ~scope_test() { printf("scope_test done!\n"); }
};

...

{
 const scope_test &test= scope_test();
 printf("in scope\n");
}
```

will print:

```
in scope
scope_test done!
```

This is the language mechanism that allows ScopeGuard to work.

Share Improve this answer

edited Jul 3, 2012 at 14:41

Follow



user142162

answered Sep 12, 2008 at 23:27



MSN

54.5k ● 7 ● 78 ● 107

- 
- 1 You can't take the address of a reference, but that doesn't mean that they don't physically take up space. Barring optimisations, they most certainly can.  
– [Lightness Races in Orbit](#) Apr 24, 2011 at 16:27
- 
- 3 Impact notwithstanding, "A reference on the stack doesn't take up any space at all" is patently false.  
– [Lightness Races in Orbit](#) Apr 25, 2011 at 23:09
- 
- 1 @Tomalak, well, that depends also on the compiler. But yes, saying that is a bit confusing. I suppose it would be less confusing to just remove that. – [MSN](#) Apr 26, 2011 at 21:52
- 
- 1 In any given specific case it may or it may not. So "it does not" as a categorical assertion is wrong. That's what I'm saying. :) [I can't remember what the standard says on the issue; the rules of reference members may impart a general rule of "references may take up space", but I don't have my copy of the standard with me here on the beach :D]  
– [Lightness Races in Orbit](#) Apr 26, 2011 at 22:22
- 



This is based on the [tutorial](#). What is written makes it more clear:

23



```
>>> The address that locates a variable within memory
 what we call a reference to that variable. (5th pa
```



```
>>> The variable that stores the reference to another variable is what we call a pointer. (3rd paragraph)
```

Simply to remember that,

```
>>> reference stands for memory location
>>> pointer is a reference container (Maybe because we several times, it is better to remember that reference
```

What's more, as we can refer to almost any pointer tutorial, a pointer is an object that is supported by pointer arithmetic which makes pointer similar to an array.

Look at the following statement,

```
int Tom(0);
int & alias_Tom = Tom;
```

`alias_Tom` can be understood as an `alias` of a `variable` (different with `typedef`, which is `alias` of a `type`) `Tom`. It is also OK to forget the terminology of such statement is to create a reference of `Tom`.

Share Improve this answer

Follow

edited Jul 7, 2015 at 21:35



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Jan 13, 2014 at 13:14



|||||||

449 ● 3 ● 9

- 1 And if a class has a reference variable, It should be initialized with either a `nullptr` or a valid object in the initialization list.

– [Misgevolution](#) Jun 15, 2015 at 17:32

- 1 The wording in this answer is too confusing for it to be of much real use. Also, @Misgevolution, are you seriously recommending to readers to initialise a reference with a `nullptr` ? Have you actually read any other part of this thread, or...? – [underscore\\_d](#) Oct 11, 2015 at 17:27 ✎
- 1 My bad, sorry for that stupid thing I said. I must have been sleep deprived by that time. 'initialize with nullptr' is totally wrong. – [Misgevolution](#) Oct 11, 2015 at 18:09 ✎



21



A reference is not another name given to some memory. It's a immutable pointer that is automatically de-referenced on usage. Basically it boils down to:

```
int& j = i;
```



It internally becomes



```
int* const j = &i;
```

Share Improve this answer

Follow

edited Jan 10, 2017 at 1:46



[Ryan](#)

4,986 ● 1 ● 34 ● 37

answered Feb 26, 2013 at 5:20



[tanweer alam](#)

437 ● 4 ● 6

- 15 This isn't what the C++ Standard says, and it is not required for the compiler to implement references in the way



described by your answer. – [jogojapan](#) Feb 26, 2013 at 5:22

@jogojapan: Any way that is valid for a C++ compiler to implement a reference is also a valid way for it to implement a `const` pointer. That flexibility doesn't prove that there is a difference between a reference and a pointer. – [Ben Voigt](#) Aug 26, 2015 at 23:00 ✎

- 2 @BenVoigt It may be true that any valid implementation of one is also a valid implementation of the other, but that doesn't follow in an obvious way from the definitions of these two concepts. A good answer would have started from the definitions, and demonstrated why the claim about the two being ultimately the same is true. This answer seems to be some sort of comment on some of the other answers. – [jogojapan](#) Aug 26, 2015 at 23:41

A reference **is** another name given to an object. The compiler is allowed to have any kind of implementation, as long as you can't tell the difference, this is known as the "as-if" rule. The important part here is that you can't tell the difference. If you can discover that a pointer does not have storage, the compiler is in error. If you can discover that a reference does not have storage, the compiler is still conformant. – [sp2danny](#) Sep 6, 2017 at 14:52



20



A reference to a pointer is possible in C++, but the reverse is not possible means a pointer to a reference isn't possible. A reference to a pointer provides a cleaner syntax to modify the pointer. Look at this example:

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

void swap(char * &str1, char * &str2)
{
```

```

 char *temp = str1;
 str1 = str2;
 str2 = temp;
}

int main()
{
 char *str1 = "Hi";
 char *str2 = "Hello";
 swap(str1, str2);
 cout<<"str1 is "<<str1<<endl;
 cout<<"str2 is "<<str2<<endl;
 return 0;
}

```

And consider the C version of the above program. In C you have to use pointer to pointer (multiple indirection), and it leads to confusion and the program may look complicated.

```

#include<stdio.h>
/* Swaps strings by swapping pointers */
void swap1(char **str1_ptr, char **str2_ptr)
{
 char *temp = *str1_ptr;
 *str1_ptr = *str2_ptr;
 *str2_ptr = temp;
}

int main()
{
 char *str1 = "Hi";
 char *str2 = "Hello";
 swap1(&str1, &str2);
 printf("str1 is %s, str2 is %s", str1, str2);
 return 0;
}

```

Visit the following for more information about reference to pointer:

- [C++: Reference to Pointer](#)
- [Pointer-to-Pointer and Reference-to-Pointer](#)

As I said, a pointer to a reference isn't possible. Try the following program:

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

int main()
{
 int x = 10;
 int *ptr = &x;
 int &*ptr1 = ptr;
}
```

Share Improve this answer

edited Jul 8, 2015 at 9:06

Follow

answered Feb 9, 2015 at 13:17



**Destructor**

14.4k ● 11 ● 68 ● 130



18



There is one fundamental difference between pointers and references that I didn't see anyone had mentioned: references enable pass-by-reference semantics in function arguments. Pointers, although it is not visible at first do not: they only provide pass-by-value semantics. This has been very nicely described in [this article](#).



Regards, &rzej



Share Improve this answer

answered Feb 6, 2012 at 8:59

Follow



[Andrzej](#)

5,117 ● 1 ● 29 ● 37

- 
- 1 References and pointers are both handles. They both give you the semantic where your *object* is passed by reference, but the *handle* is copied. No difference. (There are other ways to have handles too, such as a key for lookup in a dictionary) – [Ben Voigt](#) Nov 7, 2013 at 17:16

---

I also used to think like this. But see the linked article describing why it is not so. – [Andrzej](#) Nov 12, 2013 at 9:08

- 
- 2 @Andrzej: That's just a very long version of the single sentence in my comment: **The handle is copied.** – [Ben Voigt](#) Nov 12, 2013 at 15:10

---

I need more explanation on this "The handle is copied". I understand some basic idea but I think physically the reference and pointer both point the memory location of variable. Is it like alias stores the value variable and updates it as value of variable is change or something else? I'm novice, and please don't flag it as a stupid question. – [Asim](#) Dec 11, 2013 at 23:13

- 
- 1 @Andrzej False. In both cases, pass-by-value is occurring. The reference is passed by value and the pointer is passed by value. Saying otherwise confuses newbies. – [mrr](#) Apr 27, 2014 at 9:28
- 



I use references unless I need either of these:

18



- Null pointers can be used as a sentinel value, often a cheap way to avoid function overloading or use of a bool.
- You can do arithmetic on a pointer. For example, `p += offset;`

Share Improve this answer

Follow

edited Jul 7, 2015 at 21:22



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Sep 12, 2008 at 13:41



Aardvark

8,561 ● 7 ● 47 ● 64

5 You can write `&r + offset` where `r` was declared as reference – [M.M](#) Feb 23, 2017 at 5:13



17



At the risk of adding to confusion, I want to throw in some input, I'm sure it mostly depends on how the compiler implements references, but in the case of gcc the idea that a reference can only point to a variable on the stack is not actually correct, take this for example:

```
#include <iostream>
int main(int argc, char** argv) {
 // Create a string on the heap
 std::string *str_ptr = new std::string("THIS IS A
 // Dereference the string on the heap, and assign
 std::string &str_ref = *str_ptr;
 // Not even a compiler warning! At least with gcc
 // Now lets try to print it's value!
 std::cout << str_ref << std::endl;
 // It works! Now lets print and compare actual mem
```

```

std::cout << str_ptr << " : " << &str_ref << std::endl;
// Exactly the same, now remember to free the memory
delete str_ptr;
}

```

Which outputs this:

```

THIS IS A STRING
0xbb2070 : 0xbb2070

```

If you notice even the memory addresses are exactly the same, meaning the reference is successfully pointing to a variable on the heap! Now if you really want to get freaky, this also works:

```

int main(int argc, char** argv) {
 // In the actual new declaration let immediately delete it
 // to the reference
 std::string &str_ref = *(new std::string("THIS IS A STRING"));
 // Once again, it works! (at least in gcc)
 std::cout << str_ref;
 // Once again it prints fine, however we have no pointer
 // allocation, right? So how do we free the space we just
 // allocated?
 delete &str_ref;
 /*And, it works, because we are taking the memory address
 // of the reference is
 // storing, and deleting it, which is all a pointer is
 // specifying
 // the address with '&' whereas a pointer does that in a
 // different way
 // of like
 // calling delete &(*str_ptr); (which also compiles and
 // works)
}

```

Which outputs this:

```

THIS IS A STRING

```

Therefore a reference IS a pointer under the hood, they both are just storing a memory address, where the address is pointing to is irrelevant, what do you think would happen if I called `std::cout << str_ref;` AFTER calling `delete &str_ref`? Well, obviously it compiles fine, but causes a segmentation fault at runtime because it's no longer pointing at a valid variable, we essentially have a broken reference that still exists (until it falls out of scope), but is useless.

In other words, a reference is nothing but a pointer that has the pointer mechanics abstracted away, making it safer and easier to use (no accidental pointer math, no mixing up '.' and '->', etc.), assuming you don't try any nonsense like my examples above ;)

Now **regardless** of how a compiler handles references, it will **always** have some kind of pointer under the hood, because a reference **must** refer to a specific variable at a specific memory address for it to work as expected, there is no getting around this (hence the term 'reference').

The only major rule that's important to remember with references is that they must be defined at the time of declaration (with the exception of a reference in a header, in that case it must be defined in the constructor, after the object it's contained in is constructed it's too late to define it).

**Remember, my examples above are just that, examples demonstrating what a reference is, you would never want to use a reference in those ways!**

**For proper usage of a reference there are plenty of answers on here already that hit the nail on the head**

Share Improve this answer

answered Oct 14, 2014 at 21:38

Follow



Tory

477 ● 5 ● 10



17



Another difference is that you can have pointers to a void type (and it means pointer to anything) but references to void are forbidden.

```
int a;
void * p = &a; // ok
void & p = a; // forbidden
```

I can't say I'm really happy with this particular difference. I would much prefer it would be allowed with the meaning reference to anything with an address and otherwise the same behavior for references. It would allow to define some equivalents of C library functions like memcpy using references.

Share Improve this answer

edited Apr 29, 2015 at 15:00

Follow

answered Jan 29, 2010 at 15:15



kriss

24.1k ● 17 ● 101 ● 119





Also, a reference that is a parameter to a function that is inlined may be handled differently than a pointer.

15



```
void increment(int *pint) { (*pint)++; }
void increment(int &refint) { refint++; }
void incptrtest()
{
 int testptr=0;
 increment(&testptr);
}
void increftest()
{
 int testref=0;
 increment(testref);
}
```

Many compilers when inlining the pointer version one will actually force a write to memory (we are taking the address explicitly). However, they will leave the reference in a register which is more optimal.

Of course, for functions that are not inlined the pointer and reference generate the same code and it's always better to pass intrinsics by value than by reference if they are not modified and returned by the function.

Share Improve this answer

answered Oct 15, 2009 at 1:57

Follow



Adisak

6,916 ● 1 ● 41 ● 48



Another interesting use of references is to supply a default argument of a user-defined type:



```
class UDT
{
public:
 UDT() : val_d(33) {};
 UDT(int val) : val_d(val) {};
 virtual ~UDT() {};
private:
 int val_d;
};

class UDT_Derived : public UDT
{
public:
 UDT_Derived() : UDT() {};
 virtual ~UDT_Derived() {};
};

class Behavior
{
public:
 Behavior(
 const UDT &udt = UDT()
) {};
};

int main()
{
 Behavior b; // take default

 UDT u(88);
 Behavior c(u);

 UDT_Derived ud;
 Behavior d(ud);

 return 1;
}
```

The default flavor uses the 'bind const reference to a temporary' aspect of references.



14

This program might help in comprehending the answer of the question. This is a simple program of a reference "j" and a pointer "ptr" pointing to variable "x".



```
#include<iostream>

using namespace std;

int main()
{
 int *ptr=0, x=9; // pointer and variable declaration
 ptr=&x; // pointer to variable "x"
 int & j=x; // reference declaration; reference to vari

 cout << "x=" << x << endl;

 cout << "&x=" << &x << endl;

 cout << "j=" << j << endl;

 cout << "&j=" << &j << endl;

 cout << "*ptr=" << *ptr << endl;

 cout << "ptr=" << ptr << endl;

 cout << "&ptr=" << &ptr << endl;
 getch();
}
```

Run the program and have a look at the output and you'll understand.

Also, spare 10 minutes and watch this video:

<https://www.youtube.com/watch?v=rIJrrGV0iOg>

Share Improve this answer

edited Mar 15, 2013 at 3:11

Follow

answered Mar 15, 2013 at 3:03



Arlene Batada

1,635 ● 2 ● 12 ● 11



I feel like there is yet another point that hasn't been covered here.

13



Unlike the pointers, references are **syntactically equivalent** to the object they refer to, i.e. any operation that can be applied to an object works for a reference, and with the exact same syntax (the exception is of course the initialization).



While this may appear superficial, I believe this property is crucial for a number of C++ features, for example:

- *Templates*. Since template parameters are duck-typed, syntactic properties of a type is all that matters, so often the same template can be used with both `T` and `T&`.  
(or `std::reference_wrapper<T>` which still relies on an implicit cast to `T&`)  
Templates that cover both `T&` and `T&&` are even more common.

- *Lvalues*. Consider the statement `str[0] = 'X';`  
Without references it would only work for c-strings (`char* str`). Returning the character by reference allows user-defined classes to have the same notation.
- *Copy constructors*. Syntactically it makes sense to pass objects to copy constructors, and not pointers to objects. But there is just no way for a copy constructor to take an object by value - it would result in a recursive call to the same copy constructor. This leaves references as the only option here.
- *Operator overloads*. With references it is possible to introduce indirection to an operator call - say, `operator+(const T& a, const T& b)` while retaining the same infix notation. This also works for regular overloaded functions.

These points empower a considerable part of C++ and the standard library so this is quite a major property of references.

Share Improve this answer

answered Jul 6, 2017 at 19:48

Follow



Ap31

3,304 ● 1 ● 19 ● 25

---

"implicit cast" a cast is a syntax construct, it exist in the grammar; a cast is always explicit – [curiousguy](#) Oct 22, 2017 at 23:58

---



12



There is a very important non-technical difference between pointers and references: An argument passed to a function by pointer is much more visible than an argument passed to a function by non-const reference.

For example:

```
void fn1(std::string s);
void fn2(const std::string& s);
void fn3(std::string& s);
void fn4(std::string* s);

void bar() {
 std::string x;
 fn1(x); // Cannot modify x
 fn2(x); // Cannot modify x (without const_cast)
 fn3(x); // CAN modify x!
 fn4(&x); // Can modify x (but is obvious about it)
}
```

Back in C, a call that looks like `fn(x)` can only be passed by value, so it definitely cannot modify `x`; to modify an argument you would need to pass a pointer `fn(&x)`. So if an argument wasn't preceded by an `&` you knew it would not be modified. (The converse, `&` means modified, was not true because you would sometimes have to pass large read-only structures by `const` pointer.)

Some argue that this is such a useful feature when reading code, that pointer parameters should always be used for modifiable parameters rather than non-`const` references, even if the function never expects a `nullptr`. That is, those people argue that function signatures like

`fn3()` above should not be allowed. [Google's C++ style guidelines](#) are an example of this.

Share Improve this answer

edited Jun 25, 2018 at 18:58

Follow



Donald Duck

8,831 ● 23 ● 79 ● 101

answered Nov 2, 2017 at 11:16



Arthur Tacca

9,868 ● 3 ● 37 ● 53



12



A reference is a const pointer. `int * const a = &b` is the same as `int& a = b`. This is why there's no such thing as a const reference, because it is already const, whereas a reference to const is `const int * const a`. When you compile using `-O0`, the compiler will place the address of `b` on the stack in both situations, and as a member of a class, it will also be present in the object on the stack/heap identically to if you had declared a const pointer. With `-Ofast`, it is free to optimise this out. A const pointer and reference are both optimised away.

Unlike a const pointer, there is no way to take the address of the reference itself, as it will be interpreted as the address of the variable it references. Because of this, on `-Ofast`, the const pointer representing the reference (the address of the variable being referenced) will always be optimised off the stack, but if the program absolutely needs the address of an actual const pointer (the address of the pointer itself, not the address it points to) i.e. you print the address of the const pointer, then the const

pointer will be placed on the stack so that it has an address.

Otherwise it is identical i.e. when you print the that address it points to:

```
#include <iostream>
```

```
int main() {
 int a =1;
 int* b = &a;
 std::cout << b ;
}
```

```
int main() {
 int a =1;
 int& b = a;
 std::cout << &b ;
}
```

they both have the same assembly output

-Ofast:

main:

```
 sub rsp, 24
 mov edi, OFFSET FLAT:_ZSt4cout
 lea rsi, [rsp+12]
 mov DWORD PTR [rsp+12], 1
 call std::basic_ostream<char, std::char_traits<char> >::_M_
std::basic_ostream<char, std::char_traits<char> >::_M_
const*)
 xor eax, eax
 add rsp, 24
 ret
```

-O0:

main:

```
 push rbp
 mov rbp, rsp
 sub rsp, 16
 mov DWORD PTR [rbp-12], 1
```



```

 lea rax, [rbp-12]
 mov QWORD PTR [rbp-8], rax
 mov rax, QWORD PTR [rbp-8]
 mov rsi, rax
 mov edi, OFFSET FLAT:_ZSt4cout
 call std::basic_ostream<char, std::char_traits<::operator<<(void const*)
 mov eax, 0
 leave
 ret

```

The pointer has been optimised off the stack, and the pointer isn't even dereferenced on -Ofast in both cases, instead it uses a compile time value.

As members of an object they are identical on -O0 through -Ofast.

```

#include <iostream>
int b=1;
struct A {int* i=&b; int& j=b;};
A a;
int main() {
 std::cout << &a.j << &a.i;
}

```

The address of b is stored twice in the object.

```

a:
 .quad b
 .quad b

```

```

 mov rax, QWORD PTR a[rip+8] //&a.j
 mov esi, OFFSET FLAT:a //&a.i

```

When you pass by reference, on -O0, you pass the address of the variable referenced, so it is identical to

passing by pointer i.e. the address the const pointer contains. On -Ofast this is optimised out by the compiler in an inline call if the function can be inlined, as the dynamic scope is known, but in the function definition, the parameter is always dereferenced as a pointer (expecting the address of the variable being referenced by the reference) where it may be used by another translation unit and the dynamic scope is unknown to the compiler, unless of course the function is declared as a static function, then it can't be used outside of the translation unit and then it passes by value so long as it isn't modified in the function by reference, then it will pass the address of the variable being referenced by the reference that you're passing, and on -Ofast this will be passed in a register and kept off of the stack if there are enough volatile registers in the calling convention.

Share Improve this answer

edited Sep 15, 2021 at 12:30

Follow

answered Jun 4, 2020 at 15:59



Lewis Kelsey

4,667 ● 1 ● 39 ● 54



Maybe some metaphors will help; In the context of your desktop screenspace -

11



- A reference requires you to specify an actual window.
- A pointer requires the location of a piece of space on screen that you assure it will contain zero or more



instances of that window type.



Share Improve this answer

answered Dec 27, 2014 at 13:00

Follow



George R

3,850 ● 4 ● 35 ● 39



8



## Difference between pointer and reference

A pointer can be initialized to 0 and a reference not. In fact, a reference must also refer to an object, but a pointer can be the null pointer:

```
int* p = 0;
```



But we can't have `int& p = 0;` and also `int& p=5 ;`.

In fact to do it properly, we must have declared and defined an object at the first then we can make a reference to that object, so the correct implementation of the previous code will be:

```
Int x = 0;
Int y = 5;
Int& p = x;
Int& p1 = y;
```

Another important point is that is we can make the declaration of the pointer without initialization however no such thing can be done in case of reference which must make a reference always to variable or object. However such use of a pointer is risky so generally we check if the

pointer is actually is pointing to something or not. In case of a reference no such check is necessary, because we know already that referencing to an object during declaration is mandatory.

Another difference is that pointer can point to another object however reference is always referencing to the same object, let's take this example:

```
Int a = 6, b = 5;
Int& rf = a;

Cout << rf << endl; // The result we will get is 6, be
to the value of a.

rf = b;
cout << a << endl; // The result will be 5 because the
stored into the address of a so the former value of a
```

Another point: When we have a template like an STL template such kind of a class template will always return a reference, not a pointer, to make easy reading or assigning new value using operator []:

```
Std ::vector<int>v(10); // Initialize a vector with 10
V[5] = 5; // Writing the value 5 into the 6 element of
returned type of operator [] was a pointer and not a r
this *v[5]=5, by making a reference we overwrite the e
assignment "="
```

Share Improve this answer

edited Jun 20, 2020 at 9:12

Follow



Community Bot

1 • 1

answered Jan 6, 2017 at 14:01



dhokar.w

470 ● 4 ● 8

- 
- 1 We still can have `const int& i = 0`. – [Revolver\\_Ocelot](#) Jan 6, 2017 at 14:24
- 
- 1 In this case the reference will be used only in read we can't modify this const reference even using "const\_cast" because "const\_cast" accept only pointer not reference. – [dhokar.w](#) Jan 6, 2017 at 14:37
- 
- 1 const\_cast works with references pretty well: [coliru.stacked-crooked.com/a/eebb454ab2cfd570](http://coliru.stacked-crooked.com/a/eebb454ab2cfd570) – [Revolver\\_Ocelot](#) Jan 6, 2017 at 16:52
- 
- 1 you are making a cast to reference not casting a reference try this; `const int& i=; const_cast<int>(i); i` try to throw away the constness of the reference to make possible write and assignement of new value to the reference but this is not possible . please focus !! – [dhokar.w](#) Jan 6, 2017 at 19:38
- 



8



## Some key pertinent details about references and pointers

### Pointers



- Pointer variables are declared using the *unary suffix declarator operator* \*
- Pointer objects are assigned an address value, for example, by assignment to an array object, the address of an object using the *& unary prefix*

*operator*, or assignment to the value of another pointer object

- A pointer can be reassigned any number of times, pointing to different objects
- A pointer is a variable that holds the assigned address. It takes up storage in memory equal to the size of the address for the target machine architecture
- A pointer can be mathematically manipulated, for instance, by the increment or addition operators. Hence, one can iterate with a pointer, etc.
- To get or set the contents of the object referred to by a pointer, one must use the *unary prefix operator* \* to *dereference* it

## References

- References must be initialized when they are declared.
- References are declared using the *unary suffix declarator operator* &.
- When initializing a reference, one uses the name of the object to which they will refer directly, without the need for the *unary prefix operator* &
- Once initialized, references cannot be pointed to something else by assignment or arithmetical manipulation

- There is no need to dereference the reference to get or set the contents of the object it refers to
- Assignment operations on the reference manipulate the contents of the object it points to (after initialization), not the reference itself (does not change where it points to)
- Arithmetic operations on the reference manipulate the contents of the object it points to, not the reference itself (does not change where it points to)
- In pretty much all implementations, the reference is actually stored as an address in memory of the referred to object. Hence, it takes up storage in memory equal to the size of the address for the target machine architecture just like a pointer object

Even though pointers and references are implemented in much the same way "under-the-hood," the compiler treats them differently, resulting in all the differences described above.

## Article

A recent article I wrote that goes into much greater detail than I can show here and should be very helpful for this question, especially about how things happen in memory:

[Arrays, Pointers and References Under the Hood In-Depth Article](#)

Follow



Community Bot

1 • 1

answered Aug 1, 2019 at 8:02



Xitalogy

1,612 • 2 • 15 • 17

- 
- 1 I suggest adding the main points from the article to the answer itself. Link-only answers are usually discouraged, see [stackoverflow.com/help/deleted-answers](https://stackoverflow.com/help/deleted-answers) – HolyBlackCat Aug 1, 2019 at 8:05
- 

@HolyBlackCat I was wondering about that. The article is long and in-depth, and develops from first principles up to in-depth treatments with lots of code examples and memory dumps then finishes with exercises that further develop the in-depth code examples and explanations. It also has a lot of diagrams. I will try to figure out how to put some of the key points in here directly, but not sure right now how to do that in the best way. Thank you very much for your input. I will do my best before my answer gets deleted. – Xitalogy Aug 1, 2019 at 8:18

---

1

2

Next



**Highly active question.** Earn 10 reputation (not counting the [association bonus](#)) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.