What are the differences between a pointer variable and a reference variable in C++?

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

But what are the differences?

Summary from answers and links below:

1. A pointer can be re-assigned any number of times while a reference can not be re-seated after binding.
2. Pointers can point nowhere (`NULL`), whereas reference always refer to an object.
3. You can't take the address of a reference like you can with pointers.
4. There's no "reference arithmetics" (but you can take the address of an object pointed by a reference and do pointer arithmetics on it as in `&obj + 5`).

To clarify a misconception:

> *The C++ standard is very careful to avoid dictating how a compiler must implement references, but every C++ compiler implements references as pointers. That is, a declaration such as:*
>
> ```
> int &ri = i;
> ```
>
> ***if it's not optimized away entirely**, allocates the same amount of storage as a pointer, and places the address of i into that storage.*

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

As a general rule,

- Use references in function parameters and return types to define useful and self-documenting interfaces.
- Use pointers to implement algorithms and data structures.

Interesting read:

- My alltime favorite C++ FQA lite.
- References vs. Pointers.
- An Introduction to References.
- References and const.

c++ pointers reference c++-faq

edited Mar 2 at 22:35 Siddhant Saraf 20 6

asked Sep 11 '08 at 20:03 prakash 21.4k 22 81 109

17 I think point 2 should be "A pointer is allowed to be NULL but a reference is not. Only malformed code can create a NULL reference and its behavior is undefined." – Mark Ransom Oct 8 '10 at 17:21

3 Pointers are just another type of object, and like any object in C++, they can be a variable. References on the other hand are never objects, *only* variables. – Kerrek SB Jun 16 '12 at 10:14

2 I don't feel qualified to edit the "clarify a misconception" part of this question, but I think it should be observed that if `int i` lives in a register for its entire life, the reference isn't likely to be a pointer - it can simply alias the same register. – Drew Dormann Apr 24 '13 at 18:53

2 reference is a variable alias – Khaled A Khunaifer Dec 23 '13 at 8:53

3 I like how the very first sentence is a total fallacy. References have their own semantics. – Lightness Races in Orbit Jun 1 '14 at 1:58

26 Answers

1. A pointer can be re-assigned:

```
int x = 5;
```

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

A reference cannot, and must be assigned at initialization:

```
int x = 5;
int y = 6;
int &r = x;
```

2. A pointer has its own memory address and size on the stack (4 bytes on x86), whereas a reference shares the same memory address (with the original variable) but also takes up some space on the stack. Since a reference has the same address as the original variable itself, it is safe to think of a reference as another name for the same variable. Note: What a pointer points to can be on the stack or heap. Ditto a reference. My claim in this statement is not that a pointer must point to the stack. A pointer is just a variable that holds a memory address. This variable is on the stack. Since a reference has its own space on the stack, and since the address is the same as the variable it references. More on stack vs heap. This implies that there is a real address of a reference that the compiler will not tell you.

```
int x = 0;
int &r = x;
int *p = &x;
int *p2 = &r;
assert(p == p2);
```

3. You can have pointers to pointers to pointers offering extra levels of indirection. Whereas references only offer one level of indirection.

```
int x = 0;
int y = 0;
int *p = &x;
int *q = &y;
int **pp = &p;
pp = &q;//*pp = q
**pp = 4;
assert(y == 4);
assert(x == 0);
```

4. Pointer can be assigned NULL directly, whereas reference cannot. If you try hard enough, and you know how, you can make the address of a reference NULL. Likewise, if you try hard enough you can have a reference to a pointer, and then that reference can contain NULL.

```
int *p = NULL;
int &r = NULL; <--- compiling error
```

5. Pointers can iterate over 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. This is no matter what size the object is that the pointer points to.

6. A pointer needs to be dereferenced with `*` to access the memory location it points to, whereas a reference can be used directly. A pointer to a class/struct uses `->` to access it's members whereas a reference uses a `.`.

7. A pointer is a variable that holds a memory address. Regardless of how a reference is implemented, a reference has the same memory address as the item it references.

8. References cannot be stuffed into an array, whereas pointers can be (Mentioned by user @litb)

9. Const references can be bound to temporaries. Pointers cannot (not without some indirection):

```
const int &x = int(12); //legal C++
int *y = &int(12); //illegal to dereference a temporary.
```

This makes `const&` safer for use in argument lists and so forth.

edited May 26 '13 at 14:58

community wiki
18 revs, 7 users 59%
Brian R. Bondy

8 ...but dereferencing NULL is undefined. For example, you can't test if a reference is NULL (e.g., &ref == NULL). – Pat Notz Sep 11 '08 at 22:07

20 Number 2 is *not* true. A references is not simply "another name for the same variable." References may be passed to functions, stored in classes, etc. in a manner very similar to pointers. They exist independently from the variables they point to. – Derek Park Sep 12 '08 at 23:37

13 Brian, the stack is not relevant. References and pointers do not have to take space on the stack. They can both be allocated on the heap. – Derek Park Sep 19 '08 at 4:33

8 Brian, the fact that a variable (in this case a pointer or reference) requires space does *not* mean it requires space on the stack. Pointers and references may not only *point* to the heap, they may actually be *allocated* on the heap. – Derek Park Sep 19 '08 at 15:26

16 another important diff: references cannot be stuffed into an array – ▷ Johannes Schaub - litb ◁ Feb 27 '09 at 21:10

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.

edited Jul 7 at 21:27

Peter Mortensen
8,990 10 61 96

answered Feb 27 '09 at 21:26

Christoph
84.1k 23 107 164

5 @kriss: No, you can also get a dangling reference by returning an automatic variable by reference. – Ben Voigt Nov 2 '10 at 6:14

1 @Ben Voight: OK, but that's really calling for troubles. Some compilers even return warnings if you do that. – kriss Nov 2 '10 at 7:38

5 @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 '10 at 13:15

1 @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 '10 at 15:15

2 References are not a kind of pointer. They are a new name for an existing object. – catphive Jul 20 '11 at 1:28

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.

edited Oct 8 '10 at 16:52 — Peter Mortensen — 8,990 — 10 61 96

answered Sep 11 '08 at 21:43 — Matt Price — 14.7k — 7 26 37

2 but whats the use case for this ? – digitalSurgeon Oct 22 '09 at 14:10

9 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 '09 at 18:14

3 @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 '10 at 17:06

3 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 '13 at 20:14

Can someone give me a concrete example of how a const& can extend the life of the variable it's referencing? Since references must be initialized, they'll always have to be declared after the variable it'll be referencing, and so I can't think of a situation where the reference would be on a different scope, or on a scope that would last longer than the variable's scope. – Bruno Santos Mar 15 '14 at 1:32

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

```
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.

Edit: a few clarifications.

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.

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(a,Long,list,of,parameters);  // crash occurs here - obvious why?
}

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

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

Edit: Some further thoughts.

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 - 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& ref(T* p)
{
    if (p == NULL)
        throw std::invalid_argument(std::string("NULL reference"));
    return *p;
}

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

edited May 22 '11 at 19:23

answered Sep 11 '08 at 21:06

Mark Ransom
161k 15 153 348

2 As a nitpick, I'd say that the reference isn't actually null - it references bad memory. Although it is a valid point that just because it's a reference, doesn't mean that it refers to something that's valid – Nick Sep 12 '08 at 13:42

29 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 '08 at 16:00

4 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 '09 at 21:14

3 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 '11 at 11:41

2 @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 '11 at 2:11

You forgot the most important part:

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 :-)

edited Jul 7 at 21:21

Peter Mortensen
8,990 10 61 96

answered Sep 11 '08 at 22:10

Orion Edwards
64k 33 162 239

1 @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 '08 at 12:57

1 Smart Pointers have both . (methods on smart pointer class) and -> (methods on underlying type). – JBRWilkinson Apr 9 '14 at 9:11

@user6105 Orion Edwards statement is actually 100% true. *"access members of [the] de-referenced pointer"* A pointer does not have any members. The object the pointer refers to has members, and access to those is exactly what `->` provides for references to pointers, just as with the pointer itself. – Max Truxa Apr 15 at 17:37

Actually, a reference is not really like a pointer.

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.

edited Jul 7 at 21:25

Peter Mortensen
8,990 10 61 96

answered Sep 19 '08 at 12:23

Vincent Robert
19.6k 8 53 98

1 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++ – Arkadiy Mar 2 '09 at 16:27

2 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 '09 at 20:36

1 @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 '12 at 23:08

int *p = NULL; int &r=*p; reference pointing to NULL; if(r){} -> boOm ;) – sree Oct 4 at 12:37

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 at 17:35

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

- 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 particularly troublesome
    // for optimizers. It will constantly call maybeModify on array[0] while
    // adding array[1] to array[2]..array[size-1]. There's no real reason to
    // do this, other than to demonstrate the power of references.
    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 accessing array[1]
    // all the time, access it once and store the result in a register,
    // 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 reference to a register
        array[i] += a1;  // Use the saved register value over and over
    }
    a[0] = a0; // Store the modified a[0] back into the array
}
```

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 the
    // 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.

edited Jul 7 at 21:34

Peter Mortensen
8,990 10 61 96

answered Sep 1 '13 at 3:44

Cort Ammon
3,118 7 18

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 '13 at 4:27

I believe the optimizer already does that "a bunch of pointer arithemetic does not occur" check for a bunch of other reasons. – Ben Voigt Sep 11 '13 at 4:32

I find this very impressive example which is also not hard to understand. +1 – Stefan Dec 30 '13 at 9:32

"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 at 17:31

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

edited Oct 15 at 19:28

answered Sep 11 '08 at 20:07

Arkadiy
12.7k 2 33 79

A reference can never be `NULL`.

edited Jun 1 '14 at 22:37

answered Sep 11 '08 at 20:12


Cole Johnson
4,016 10 27 49

RichS

2 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 Jun 13 '14 at 11:36

5 @cmaster: **In a valid program**, a reference cannot be null. But a pointer can. This is not a myth, this is a fact. – Mehrdad Aug 8 '14 at 4:42

1 @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 Dec 29 '14 at 10:43

int *p = NULL; int &r=*p; reference pointing to NULL; if(r){} -> boOm ;) – – sree Oct 4 at 12:43

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.

edited Oct 23 at 23:16

Андрей Беньковский
194 14

answered May 20 '11 at 19:26

Kunal Vyas
486 1 6 21

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 references.
int a = 0;
int &b = a; // b is an alias for a. Not so common to use.
```

answered Jan 1 '13 at 17:45

fatma.ekici
718 1 10 17

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.

answered Oct 29 '14 at 17:17

Lightness Races in Orbit
178k 30 259 491

---

pointers to references! Good juxtaposition – jxramos Dec 31 '14 at 20:31

This is based on the tutorial. What is written makes it more clear:

```
>>> The address that locates a variable within memory is
    what we call a reference to that variable. (5th paragraph at page 63)

>>> The variable that stores the reference to another
    variable is what we call a pointer. (3rd paragraph at page 64)
```

Simply to remember that,

```
>>> reference stands for memory location
>>> pointer is a reference container (Maybe because we will use it for
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`.

edited Jul 7 at 21:35

Peter Mortensen
8,990 10 61 96

answered Jan 13 '14 at 13:14

Life
206 2 7

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 at 17:32

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 at 17:27

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 at 18:09

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

answered Feb 6 '12 at 8:59

Andrzej
1,899 10 17

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 '13 at 17:16

I also used to think like this. But see the linked article describing why it is not so. – Andrzej Nov 12 '13 at 9:08

2 @Andrzj: That's just a very long version of the single sentence in my comment: **The handle is copied.** – Ben Voigt Nov 12 '13 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 Awan Dec 11 '13 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. – Miles Rout Apr 27 '14 at 9:28

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.

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.

edited Jul 3 '12 at 14:41 — Tim Cooper — 71.8k 20 134 162

answered Sep 12 '08 at 23:27 — MSN — 34.2k 4 46 71

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 '11 at 16:27

1 Impact notwithstanding, "A reference on the stack doesn't take up any space at all" is patently false. – Lightness Races in Orbit Apr 25 '11 at 23:09

@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 '11 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 '11 at 22:22

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 variable "x"

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=rlJrrGV0iOg

edited Mar 15 '13 at 3:11

answered Mar 15 '13 at 3:03 — Arlene Batada — 198 1 2 8

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 STRING");
    // Dereference the string on the heap, and assign it to the reference
    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 memory addresses
    std::cout << str_ptr << " : " << &str_ref << std::endl;
    // Exactly the same, now remember to free the memory on the heap
    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 de-reference and assign 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 to the heap allocation,
right? So how do we free the space we just ignorantly created?
    delete &str_ref;
    /*And, it works, because we are taking the memory address that the reference is
    storing, and deleting it, which is all a pointer is doing, just we have to specify
    the address with '&' whereas a pointer does that implicitly, this is sort of like
    calling delete &(*str_ptr); (which also compiles and runs fine).*/
}
```

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

answered Oct 14 '14 at 21:38

Tory
89 7

I use references unless I need either of these:

- 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;`

edited Jul 7 at 21:22

Peter Mortensen
8,990 10 61 96

answered Sep 12 '08 at 13:41

Aardvark
5,303 4 33 55

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.

edited Apr 29 at 15:00

answered Jan 29 '10 at 15:15

kriss
12.2k 8 45 81

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

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

It internally becomes

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

edited Jul 7 at 21:29

Peter Mortensen
8,990 10 61 96

answered Feb 26 '13 at 5:20

tanweer alam
55 3

6 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 '13 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 at 23:00

1 @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 at 23:41

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;
}
```

edited Jul 8 at 9:06

answered Feb 9 at 13:17

Pravasi Meet

3,777 2 12 34

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.

answered Sep 12 '08 at 17:59

Don Wakefield

5,127 1 21 45

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

```
void increment(int *ptrint) { (*ptrint)++; }
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.

answered Oct 15 '09 at 1:57

Adisak

3,937 19 34

A reference points to the place where the object is *now*. In applications that use managed memory this may not be where you saw it last.

So for example using .Net CLR you could create an object (by reference) on the short term heap, and when you next look it's shifted in physical memory to the long-term heap. You won't see this happen as it's done by the garbage collector on a separate thread, and will usually take place between one application clock cycle and the next. You cannot get this address because it is not guarenteed as valid for use at any future time, and as pointed out elsewhere, it may not even exist for an integer or pointer held in a register.

answered Dec 2 '13 at 10:50

Richard Petheram
152 8

3 The question is tagged c++. – Lightness Races in Orbit Oct 29 '14 at 17:17

But this doesn't show any difference between a reference and a pointer. The CLR adjusts tracking pointers when it does gc heap compaction, just like it adjusts tracking references. – Ben Voigt Aug 26 at 22:59

A pointer (&) and a tracking pointer (%) are not the same thing. A tracking pointer is indeed what other CLR users refer to simply as a reference and follows objects about the heap. A pointer is still a fixed value that refers to a memory address in the old-fashioned way. – Richard Petheram Aug 27 at 11:37

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

- 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.

answered Dec 27 '14 at 13:00

George R
1,222 1 16 23

References was included in C++ language generally for: 1. support ability of passing parameters to functions "by reference", e.g.:

```
int swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}
```

2. using references as return value of function:

```
int arraySize = 10;
int array[arraySize];

int get(int index) {
    if (index < 0 || index >= arraySize)
        return -1;
    return array[index];
}

int &set(int index) {
    if (index < 0 || index >= arraySize)
        return -1;
    return array[index];
}

//after you can use it in your code as follow:
int main() {
    int index = 0;
    int value = 0;
    //...
    //changing 'index' and 'value'  variables
    //...
    set(index) = value;
    //...
}
```

Also, you can use "independence reference", defined it like:

```
int variable = 10;
int &ref = variable;
```

But it's not recommended, because frequently you can find the replacement for it, but it inaccurate using can distort your code.

Use of reference variables have some restrictions: 1. you can't create reference to reference 2. you can't create array of references 3. you can't create pointer to reference (e.g. you can't apply '&' to it) 4. you can't use reference for byte fields.

Conclusion: Use references if you want to have function, which will change value of it parameters or if you want use function in the left side of the expression (as in examples above). In all other case use the pointers.

answered Sep 18 at 23:17

V. Panchenko
65 6

Your closing 'recommendation' is at best poorly worded and at worst directly opposite to basically all other sources on good practice in C++ programming. – underscore_d Oct 11 at 17:26

This 'recommendation' you can find in this book: amazon.com/The-Complete-Reference-4th-Edition/dp/0072226803 I think author of this book have more experience that you, Mr. @underscore_d – V. Panchenko Oct 11 at 17:58

I said "at *worst* [i.e. *possibly*] directly opposite" because I'm having trouble understanding your wording. How about instead of saying I need to buy a book, which you presumably already own, you quote the author's words that led to your opinion? I imagine that will be more readable. Also, there's no "Mr." in my screen name, so it'd be nicer if you didn't assume. – underscore_d Oct 11 at 18:05

protected by Mysticial Mar 28 '13 at 4:46

Thank you for your interest in this question. Because it has attracted low-quality answers, posting an answer now requires 10 reputation on this site.

Would you like to answer one of these unanswered questions instead?