References

Part of C++ FQA Lite. To see the original answers, follow the **FAQ** links.

This page is about C++ references - a duplicate language feature introduced in order to support other duplicate features.

[8.1] What is a reference?

FAQ: It's an alias for an object - another name by which it can be called. The implementation is frequently identical to that of pointers. But don't think of references as pointers - a reference *is* the object.

FQA: A C++ reference is like a C++ pointer except for the following differences:

- You use it as if it were a value: `ref.member`, not `ptr->member`, etc. (in this sense `ref` behaves like `(*ptr)`).
- It must be initialized to point to an object - otherwise, the code won't compile.
- After the initialization, you can't make it point to another object.
- You can't take the address of a reference like you can with pointers (forming a pointer to a pointer).
- 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`).

Strange phrases like "a reference IS the object" are used quite frequently in the C++ community. Such claims are only useful to hide the fact that C++ pointers & references are so similar that having both in the language is an unnecessary complication. In other contexts, the claims are simply false. For example, a wide class of bugs comes from accessing *dangling references* - references to objects which were already destroyed. If a reference is the object, or just another name for it, how can that happen? Names of destroyed objects are inaccessible - it takes a previously assigned pointer to access a destroyed object (C++ also breaks *that* rule - you can access a destroyed global object by its name from a destructor of another global object, but that's a different can of worms).

[8.2] What happens if you assign to a reference?

FAQ: A reference *is* the object, so of course you assign to the referent object.

FQA: Which means that you can't understand the effect of a statement as simple as `a=b;` without knowing whether `a` is a reference or not. A nice feature complementary to references (which make you wonder what "a" means) is operator overloading (which makes you wonder what "=" means). Be careful as you work your way through a quagmire of C++ code.

[8.3] What happens if you return a reference?

FAQ: You can assign to the return value of a function. This is useful for operator overloading, as in `array[index] = value;` where array is an object of a class with overloaded `operator[]` returning a reference.

FQA: Exactly - and *that's* why references exist.

C++ references are essential for supporting C++ operator overloading. That's because C has no facility for assigning to the result of a function call (a function can return a pointer and you can assign to the pointed object, but you need to use an asterisk for dereferencing the pointer, which is different from assigning with the built-in `operator[]`). Some might say that references serve a more generic purpose - they make pointers to objects feel like objects, but for most purposes that can be achieved with `typedef TStruct* T;`.

Operator overloading, in turn, is useful (though not essential) for templates - a duplicate facility solving some of the problems of C macros and creating new, frequently more costly and complicated problems. For example, operator overloading is at the heart of STL - user-defined iterators must have the interface of a C pointer for interoperability with STL algorithms, which can only be achieved with operator overloading.

Operator overloading is also useful for providing user-defined containers such as strings and vectors, duplicating the functionality of built-in strings and arrays.

C++ introduces duplicate facilities in order to introduce other duplicate facilities. Then its apologists try to convince everyone that the old facilities are "evil". Then they explain why removing the old facilities is "impractical".

[8.4] What does `object.method1().method2()` mean?

FAQ: This is called method chaining. `method1` returns a reference to an object of a class having `method2`. This is used in iostream - `cout << a << endl` is method chaining (the method is `operator<<`). A "slick" way to use method chaining is to simulate "named parameters": `a.setWidth(x).setHeight(y)`).

FQA: It's just like `object->method1()->method2()`, but with references instead of pointers.

While method chaining is natural in object-oriented languages, it's good to be aware of problems related to it. At a "high" level of discussion concerned with design, method chaining can be a sign of abstraction violation (for example, do we want the user of `object` to be able to do arbitrary operations with the return value of `method1`?). At a "low" level of discussion concerned with coding, `method1` has no way to report an error except for throwing an exception, which is not always desirable. Inferring a rule like "method chaining is evil" from these issues is probably an exaggeration, but an entire *design* relying on method chaining may raise questions.

For example, iostream is a library for I/O and formatting, and both are a frequent source of run-time errors. How should those errors be checked in statements like `cout << a << b << c ...`? The method chaining used in iostream also makes formatting quite hard - consider the "I/O manipulators".

The "implementation" of named arguments using method chaining is a particularly bad joke.

[8.5] How can you reseat a reference to make it refer to a different object?

FAQ: You can't. The reference *is* the object.

FQA: You can't do it in portable C++. While the reference is probably implemented as a pointer by your compiler, there's no C++ operator to get the address where that pointer is stored. In particular,

`&ref` gives the address of the referent object.

One has to work around this extremely rarely, but the need can emerge, especially when the source code of parts of a program is unavailable. If the reference is stored inside an object, you can figure out its offset based on the sizes of other members of the class of the hosting object. You can then modify it as if it were a pointer (as in `*(T**)p = &myobj;`). This is severe abstraction violation, and it isn't portable C++. Make sure you have no other way to achieve your purpose before doing this. In particular, a way involving paying money to the vendor of the code unavailable in source form is frequently a better idea than "clever" hacks like this.

[8.6] When should I use references, and when should I use pointers?

FAQ: Use references unless you can't, especially in interfaces. In particular, references can't point to `NULL`, so you can't have a "sentinel reference". C programmers may dislike the fact that you can't tell whether a modified value is local or a reference to something else. But this is a form of *information hiding*, which is good because you should program in the language of a problem rather than the machine.

FQA: As with most duplicate features, there's no good answer to this question.

C++ programmers use references to denote "a pointer which can't be null and points to a single object rather than an array". Using pointers in these cases confuses people because they assume that pointers are used in the *other* cases. You don't want to confuse people, so you use references to pass arguments to functions. Then it turns out that sometimes you want to pass a null pointer to those functions, and you change references to pointers throughout the code. C++ code ends up containing loads of pointers and references without an apparent reason for the choice made in each particular case.

If you choose to use a reference, make sure you don't need null pointers, pointer arithmetics or reseating. You can't have arrays of references, and you can't store references in container classes, because that would require having uninitialized references and/or pointers to references, and you can't have that. Member references in classes/structs must be initialized in constructors using the ridiculous colon syntax. This makes it harder to reuse initialization code in different constructors, and the problem propagates to the classes using your class as a member since you can't provide a default constructor. There are "smart pointers" (objects with overloaded `->` and `*` operators), but there are no "smart references" (you can't overload the dot), so you won't be able to easily switch to something "smart" later - but that's probably rarely bad since smart pointers are rarely good.

If you choose to use a pointer, make sure you don't confuse other C++ programmers. Pointers to objects of classes with overloaded operators lead to code like `(*parr)[i]`, which gets annoying. In many cases the compiler won't warn you when a pointer is left uninitialized, which may lead to errors harder to debug than simple null pointer dereferencing (not to mention security holes - but there's enough other possibilities for these in a C++ program to make this a separate issue).

These problems mean that in many situations, you must choose between two almost equally bad alternatives. While many people successfully use C pointers *in C*, it doesn't mean that always choosing pointers over references *in C++* produces no new problems - C++ has features, and C++ programmers have habits interacting badly with pointers.

But this, in turn, *doesn't* invalidate the argument "pointers are better than references because it's easy to see whether a side-effect is local or not". *Of course* the behavior of references is "information hiding" - but is it the kind of information you would like to be hidden? Isn't information hiding about making it *easy* to figure out what a program does? How does hiding side effects, which are a very basic cross-cutting semantical aspect of any imperative language, make the code closer to "the language of the problem"?

[8.7] What is a handle to an object? Is it a pointer? Is it a reference? Is it a pointer-to-a-pointer? What is it?

FAQ: A "handle" is something identifying and giving access to an object. The term is meant to be vague, omitting implementation details (a handle can be a pointer or an index into an array or a database key, etc.). Handles are often encapsulated in smart pointer classes.

FQA: One very common way to implement handles in C relies on incomplete types and typedefs:

```
typedef struct FooState* Foo;
```

The definition of `struct FooState` is included in the files implementing `Foo`, but not in the files using it. This is probably the closest thing to "encapsulation" you can get in an unmanaged environment. In particular, it has important advantages compared to C++ classes:

- When the definition of `FooState` is changed, calling code doesn't have to be recompiled (with C++ classes, changing a private member triggers recompilation of user code). This shortens the rebuild cycles and allows to provide stable binary interfaces, simplifying upgrades in many scenarios.
- The module defining `Foo` controls the allocation and deallocation of the objects (with C++ classes, the user is responsible for allocation and must choose between the stack, the global data, the free store and aggregation inside another object). In particular, `FooState` may contain a pointer to the "real" state structure, allowing transparent reallocation of these structures.

The C handle technique is also way better than the "smart pointer" tricks in C++. In addition to readability problems posed by operator overloading (is it a bare pointer or a smart pointer?), and the tedious coding involved, smart pointers provide little encapsulation. That's because overloaded `operator->` must return either a bare pointer or a smart pointer, which means that the *last* smart pointer *must* return a bare pointer, or the compilation will never end (the latter is easy to achieve with templates - like many other not so useful things). So you end up returning a bare pointer to an object of a C++ class, but in a way more convoluted than average C++ code.

When implementing "heavy" classes (unlike, for example, simple objects representing values like points in a 2-dimensional space), using C-style handles is typically much better than pointers or references to objects of C++ classes. Which is quite surprising considering the fact that supporting OO at the language level is one of the main motivations behind C++.

---

revised 17 October 2009