C++ references and pointers

Lucas Frérot

Table of contents

References and pointers can cause confusion for beginners: this note aims to clarify what references and pointers are good at, what are their differences and how they can be used in C++.

Passing function arguments

Programming language often distinguish themselves in how function arguments are passed to functions. In C++, function arguments can typically be passed in three ways:

- Pass-by-value
- Pass-by-reference
- Pass-by-address

Function arguments that are passed by value are essentially copied when passed to the function that is called. In a language like Python for example, scalar variables are passed by value. Function arguments that are passed by reference are not copied: the variable available in the function *refers* to the variable that was passed to the function.

Let us look at a C++ function that takes its arguments *by-value* (interactive link).

```cpp
#include <iostream>

void increment(int x) {
    x += 1;
    std::cout << "value of x in function: " << x << "\n";
}

int main() {
    int y = 2;
    increment(y);
```

```
    std::cout << "value of y outside function: " << y << "\n";
    return 0;
}
```

The output of this program is:

```
value of x in function: 3
value of y outside function: 2
```

This means that the value of `y` was copied into the function argument `x`. Now here is the same code, but the argument of `increment()` passed by reference (interactive link).

```
#include <iostream>

void increment(int& x) {
    x += 1;
    std::cout << "value of x in function: " << x << "\n";
}

int main() {
    int y = 2;
    increment(y);
    std::cout << "value of y outside function: " << y << "\n";
    return 0;
}
```

The output of this program is:

```
value of x in function: 3
value of y outside function: 3
```

The variable `x` inside the function `increment` *refers* to the variable `y` outside the function. This has two common use cases:

- Modification of the variable within the function: functions can return values, but in imperative languages it is common to want to modify function arguments in addition to returning a value. This is known as a *side-effect* (i.e. when function arguments are modified). A relevant example would be to update particle velocities.
- Avoiding copies of large objects: scalar variables and short strings should be passed by value (this can save some bugs, and compilers can sometimes avoid the copy overhead), but large objects, such as arrays describing atoms positions, velocities, etc., should not be copied when they are passed on to functions. Passing by reference avoids this.

When one is in the second use case (copy needs to be avoided) but not in the first case (arguments do not need, or should not, be modified), one can use a *constant reference* argument (interactive link).

```cpp
#include <iostream>
#include <vector>

auto size(const std::vector<int>& x) {
    return x.size();
}

int main() {
    std::vector<int> v = {1, 2, 3, 4};
    std::cout << "size of vector: " << size(v) << "\n";
    return 0;
}
```

Pass-by-address

In C++, pointers are variables whose value is an address in memory. That address *should* be the address of another existing variable, or the `nullptr` value, which is an invalid address. Passing a variable to a function *by-address* means that the address of that variable is copied into a pointer in the function.

Passing a variable by-address is conceptually the same as passing by-reference (with a few edge case exceptions). However, in C++ practice, pointers are not used the same as reference. Since pointers are variables that contain an address, an extra step is needed to access the value of the addressed variable. This step is called **dereferencing**. Syntactically, this is done with the `*` operator: if `x` is a pointer (i.e. a variable containing an address), then `*x` is the *dereferenced* value, that is the value contained at the memory address `x`. Here is a code sample that does the exact same thing as the `increment` function above with a reference argument (interactive link:

```cpp
#include <iostream>

void increment(int* x) {
    *x += 1;
    std::cout << "value of x in function: " << x << "\n";
    std::cout << "value of *x in function: " << *x << "\n";
}

int main() {
    int y = 2;
    increment(&y);
    std::cout << "value of y outside function: " << y << "\n";
    return 0;
}
```

Here is a sample output:

```
value of x in function: 0x7ffcc6017a84
value of *x in function: 3
value of y outside function: 3
```

Notice two things that changed with regard to the by-reference code above:

- In `main()`, the `increment()` function is called not with `y`, but with `&y`. The ampersand `&` prefix to the variable indicates that we *pass the address* of the variable `y`, not its value. Try to pass `y` instead of `&y` and the compiler will complain with a conversion error, because `int` and `int*` are different variable types and one cannot convert into the other.
- To access the actual value we are interested in , we need to do `*x`. The value of `x` is an address, as can be seen in the output.

The special case of member variables

One can access member variables of an object in C++ with the `.` operator, e.g. `atoms.positions` is the variable `positions` inside the object `atoms`. This assumes that the object `atoms` is of type `Atoms` or `Atoms&`, i.e. it is the original object or a reference to that object. If `atoms` is of type `Atoms*`, we need to dereference it *before* accessing the member variable. One issue is that the dereferencing operator `*` has priority lower than the `.` operator. This means that `*atoms.positions` is understood by the compiler as `*(atoms.positions)` which is not what we want. A way to fix this is to use parentheses to specify the operator priority: `(*atoms).positions` is what we want, because we dereference `atoms` before accessing the `positions` variable. Since this parentheses syntax is heavy, and accessing members of pointer variables is so common, C++ has an equivalent syntax: `atoms->positions`. This does the dereferencing before the member access and is exactly equivalent to `(*atoms).positions`. Within class functions, it is common to use the `this` pointer to refer to member variables with the syntax `this->positions`. The `this` pointer points to the current object, and is the C++ equivalent of the `self` object in Python.

In-scope references

The above examples should cover 80% of use cases of references. Sometimes, however, references need to be used outside of function arguments. For example, this is the case for member variables that refer to variables declared outside of an object's scope. A typical application of this could be a class `AtomsIO` (that handles io, e.g. writing atoms to a file) that defines a member variable `Atoms& atoms`. Such use cases are typically motivated by either the need to avoid copy overhead or to have a single `atoms` object that is shared between other objects.

References are declared just like any other variable, with a notable difference: **references must always be initialized on declaration**. Here's an example (interactive link):

```
#include <iostream>

int main() {
    int x = 2;
    int& y = x;

    y += 1;

    std::cout << "value of x: " << x << "\n";
    return 0;
}
```

Once a reference is declared (here `int& y = x`) it cannot be changed. In the snippet above, `x` and `y` are effectively the same variable. Since all references *must* be initialized on declaration, the code `int& y;` is invalid and will cause a compiler error.

Similarly, if member variables are references, they **must** be initialized in the constructor initializer list, see below (interactive link):

```cpp
#include <iostream>

struct Foo {
    int& x;

    Foo(int& reference): x(reference) {
        x += 1;
        std::cout << "value of x in constructor: " << x << "\n";
    }
};

int main() {
    int y = 2;
    Foo foo(y);
    std::cout << "value of y outside constructor: " << y << "\n";
    return 0;
}
```

This use case is however prone to the dangling reference problem, so pointer-based solutions with the smart pointers defined in the standard library are often preferred.

In-scope pointers

The use cases described above for references are also valid use cases for pointers, with one major caveat: pointers do not have to be initialized with a valid address. This can cause all sorts of issues, because **dereferencing an uninitialized or invalid pointer is undefined behavior**. If you are lucky, your program will just crash. If you are unlucky, you will read/write data from/to a place in memory that you should not have access to, and debugging that will be a bad time. This is why C++ is infamous for being a "shoot yourself in the foot" language.

Most of these problems come from the use of *raw pointers*, i.e. the kind of pointer we have introduced above. These pointers do not have a concept of *memory ownership*, i.e. you have no guarantee that the address stored in the pointer is valid. Ideally, they should only be used in the context shown above, where we pass-by-address and limit operations to the *dereferenced value*, not the pointer itself (we do not change the address, or free the memory). Any other use of *raw pointers* should be avoided as much as possible. Instead, the C++ standard library provides two **smart pointer** classes: `std::unique_ptr` and `std::shared_ptr`. These classes correctly manage ownership and should be preferred over references for member variables that refer to objects outside the object scope (like the `Foo` example above; interactive link).

```cpp
#include <memory>
#include <iostream>

struct Foo {
    std::shared_ptr<int> x;

    Foo(std::shared_ptr<int> ptr): x(ptr) {
        *x += 1;
```

```
        std::cout << "value of *x in constructor: " << *x << "\n";
    }
};

int main() {
    std::shared_ptr<int> y = std::make_shared<int>(2);
    Foo foo(y);
    std::cout << "value of *y outside object: " << *y << "\n";
    return 0;
}
```

Here we have used the function `std::make_shared` to allocate an `int` with the value `2` and create a `std::shared_ptr` that points to the address of this `int`.

Unfortunately, the smart pointer types do not exist in C, so one should always be wary of pointers when using a C library (e.g. MPI, BLAS, LAPACK, PetSc, or FFTW which are reference libraries for scientific computing).

Further reading

References in C++ are broken down into two catergories: *lvalue* references and *rvalue* references ("l" and "r" stand for "left" and "right"). This distinction allows fine-grained object manipulation but may be too advanced for beginners. The cppreference page details with examples the distinction between these references. It also defines *forwarding references*.

Resource management is a sensitive endeavour in C++ and very error-prone. One should familiarize themselves with the C++ Core Guidelines on the subject, which provide a set of good practices.