References

Rob Hackman

Winter 2025

University of Alberta

Table of Contents

A new type

C++ has another type: *references*

```
int main() {
  int x = 3;
  int &y = x;
  y = 10;
  cout << x << endl; // 10
  cout << &x << &y << endl; // Same address
}
```

Here `y` is an *l-value*[1] *reference* to `x`. A reference is a form of aliasing. In effect `y` is just another name for `x`, any operation applied to `y` behaves exactly as it would if applied to `x`.

[1] More on what *l-value* means later

Syntax Note — the ampersand (&)

New C++ programmers often get confused now between the usage of the symbol & in a type as a opposed to its usage as an operator. They are two completely different things.

Used in a type the & represents an l-value reference: `int &y = x;` declares `y` as a reference to an l-value integer.

Used as an operator the & is still the address-of operator, which applied to a piece of data returns its address. So `cout << &x << &y << endl;` asks to print out the address of `x` and the address of `y`

When we ask for the address of `y` we actually get the address of `x`. This is because `y` is a reference to `x`, and thus behaves like `x` in all cases!

Reference Rules

References behave exactly like the data to which they refer. In fact, in simple cases like our one above the compiler can simply treat `y` as if it were `x` — because of this the C++ standard allows that references *may or may not take up any memory at all.*

The standard also specifies some rules about what you cannot do with references, which mostly follow from the afformentioned properties.

Can't leave references uninitialized

You cannot "reseat a reference", which is to say you cannot make a reference refer to a new piece of data — there simply is no syntax to do so.

```
1    int x = 5;
2    int y = 7;
3    int &z = x;
4    z = 3; // x is now 3;
5    z = y; // x is now 7. z still refers to x.
```

Because a reference *always* behaves like the data it refers to, so assigning `z` is not changing the reference `z`, it's assigning the value of the data it refers to.

Because references cannot be reseated, they must be initialized. So like `const` variables, you cannot leave a reference uninitialized.

Can't leave references uninitialized

```
1    int x = 12;
2    int &z; // Illegal!
3    z = x;
4    // What would this even mean?
5    // z doesn't refer to anything!
```

Must initialize lvalue references to data with an address

Not all data necessarily has a memory address, temporary values created as the result of evaluating an expression may not have an address.

```
1    int x = 12;
2    int y = 5;
3    int &z = 3; // Illegal!
4    int &z = y+x; // Illegal!
5    int &z = x; // Okay
```

No pointers to references

You cannot have a "pointer to a reference" data type. Again, references are not guaranteed to have any memory, so how could you point at a reference itself?

You can have a reference to a pointer. What would the declarations of a reference to a pointer and a pointer to a reference look like? Remember spiral rule.

No pointers to references

You cannot have a "pointer to a reference" data type. Again, references are not guaranteed to have any memory, so how could you point at a reference itself?

You can have a reference to a pointer. What would the declarations of a reference to a pointer and a pointer to a reference look like? Remember spiral rule.

```
1    int x = 12;
2    int *p = &x;
3    int &l = x;
4    int &*q = &l;  // Nope
5    int *&r = p; // No problem
```

Cannot have references to references

You cannot have a reference to a reference e.g. `int &&r = ...;`

Cannot have references to references

You cannot have a reference to a reference e.g. `int &&r = ...;`

This is actually a valid C++ type, but not a reference to a reference. More on this later.

Cannot create an array of references

You cannot store references in an array, e.g.

```
int x = 0, y = 1, z = 2;
int &arr[3] = {x, y, z}; // Nope
```

Arrays are blocks of contiguous memory — how could we create an array of references if they're not guaranteed to take up any memory?

Table of Contents

References as Parameters

We *can* use references as parameters. Doing so allows us to write functions that can mutate data given to us by the caller without the need for pointers.

```
void times2(int &n) {
  n = n*2;
}

int main() {
  int x = 10;
  times2(x);
  cout << x << endl; // 20!
}
```

Pass by Reference

In C++ when writing a function that needs to mutate the arguments passed in, it is common for the parameters to be references instead of pointers.

We call this *pass by reference* as it is different than copying a value into the functions stack frame. This is also beneficial when passing data that is large and may be costly to copy, to avoid the cost of copying.

Suggestion For any data type larger than a pointer pass it by reference. If you don't intend to mutate the data declare your reference parameter as `const`

lvalue Reference Passing and Literal Arguments

When passing by lvalue reference we cannot pass a literal value.

- A literal doesn't necessarily have a memory address
- What if the function mutates the value, what does that even mean?

```
void printNTimes(int &x, int n){
  while (n > 0) {
    cout << x;
    --n;
  }
}

int main() {
  int x = 10;
  printNTimes(x, 5); // Okay
  printNTimes(3, 5); // Nope!
}
```

lvalue Reference Passing and Literal Arguments

When passing by lvalue reference we cannot pass a literal value.

- A literal doesn't necessarily have a memory address
  - Compiler *could* give it a temporary address
- What if the function mutates the value, what does that even mean?

```
void printNTimes(int &x, int n){
  while (n > 0) {
    cout << x;
    --n;
  }
}

int main() {
  int x = 10;
  printNTimes(x, 5); // Okay
  printNTimes(3, 5); // Nope!
}
```

lvalue Reference Passing and Literal Arguments

When passing by lvalue reference we cannot pass a literal value.

- A literal doesn't necessarily have a memory address
  - Compiler *could* give it a temporary address
- What if the function mutates the value, what does that even mean?
  - What if we promise we won't mutate it?

```
void printNTimes(int &x, int n){
  while (n > 0) {
    cout << x;
    --n;
  }
}

int main() {
  int x = 10;
  printNTimes(x, 5); // Okay
  printNTimes(3, 5); // Nope!
}
```

lvalue Reference Passing and Literal Arguments

If we promise the compiler we won't mutate a lvalue reference parameter (by marking it const) then it will allow us to pass literals as arguments for that parameter.

```
void printNTimes(const int &x, int n){
  while (n > 0) {
    cout << x;
    --n;
  }
}

int main() {
  int x = 10;
  printNTimes(x, 5); // Okay
  printNTimes(3, 5); // Okay now!
}
```

Table of Contents

Returning a reference

Functions can return references, as well. However, the same rules that apply to returning a pointer apply to returning a reference — namely you had better be certain that when your function returns the data its returning a reference to continues to exist.

So with references, as with pointers, you should never return a reference to a variable on your functions local stack frame.

Returning reference example

Here `larger` returns a reference — since `a` and `b` are references to data given to it by the caller it is returning a reference back to the callers own data.

```
int &larger(int &a, int &b) {
  return a > b ? a : b;
}

int main() {
  int x = 10;
  int y = 5;
  while (x > 0 || y > 0) {
    int l = larger(x,y);
    cout << l << endl;
    --l;
  }
}
```

Returning reference example

Here `larger` returns a reference — since `a` and `b` are references to data given to it by the caller it is returning a reference back to the callers own data.

However, we try this program and it prints 10 forever — why?

```
int &larger(int &a, int &b) {
  return a > b ? a : b;
}

int main() {
  int x = 10;
  int y = 5;
  while (x > 0 || y > 0) {
    int l = larger(x,y);
    cout << l << endl;
    --l;
  }
}
```

Returning reference example

Here `larger` returns a reference — since `a` and `b` are references to data given to it by the caller it is returning a reference back to the callers own data.

However, we try this program and it prints 10 forever — why?

The variable we assign the return value of `larger` to is just an `int` — it is its own `int` which we copy a value into. So decrementing `l` doesn't change `x` or `y`.

```
int &larger(int &a, int &b) {
  return a > b ? a : b;
}

int main() {
  int x = 10;
  int y = 5;
  while (x > 0 || y > 0) {
    int l = larger(x,y);
    cout << l << endl;
    --l;
  }
}
```

Fixed `larger`

In this version `l` is declared as a reference to an `int`, which is also what is returned by `larger`.

Ultimately, each iteration `l` is a reference bound to either `x` or `y`, so when we decrement `l` we're actually decrementing whatever `int` it refers to.

```
int &larger(int &a, int &b) {
  return a > b ? a : b;
}

int main() {
  int x = 10;
  int y = 5;
  while (x > 0 || y > 0) {
    int &l = larger(x,y);
    cout << l << endl;
    --l;
  }
}
```

Alternative Fixed `larger`

In the second version we assigned a reference to the reference returned by `larger`, so that we could use it to refer back to the original data.

Since `larger` returns a reference though we could just operate directly on its return value!

```
int &larger(int &a, int &b) {
  return a > b ? a : b;
}

int main() {
  int x = 10;
  int y = 5;
  while (x > 0 || y > 0) {
    cout << larger(x,y);
    --larger(x,y);
  }
}
```

References and the input operator

Now, why is it that when we write

```
cin >> n;
```

This operation can mutate `n` without requiring its address?

References and the input operator

Now, why is it that when we write

```
cin >> n;
```

This operation can mutate `n` without requiring its address?

Because `n` is passed by reference!

Overloading the I/O operators

So, now, we can discuss how to overload the input and output operators.

The type signature of our overloaded input operator for `Vec3D` should be

```
istream &operator>>(istream &in, Vec3D &v);
```

and our output operator should be

```
ostream &operator<<(ostream &out, const Vec3D &v);
```

`Vec3D` input operator

We can give the operator any behaviour we want for our type, in this case we just read in three `ints` and use them for our `x`, `y`, and `z` values.

Note that the `return` statement is the same as just saying `return in;` at the end, because the expression evaluates to `in`.

```
istream &operator>>(istream &in, Vec3D &v) {
  return in >> v.x >> v.y >> v.z;
}
```

`Vec3D` output operator

Note that the `Vec3D` reference is `const` — printing something probably shouldn't change its state, so we mark it as const to make sure we don't make a mistake.

We've also chosen not to print a line end, usually the output operator should not print a line ending, as the client programmer should decide if they want that or would rather continue printing on this line.

```
ostream &operator<<(ostream &out, const Vec3D &v) {
  out << "[" << v.x << ", " << v.y;
  return out << << ", " << v.z << "]";
}
```

Using our overloaded operators

Once overloaded we can use operators with our types as we would anything else.

Try out this example `main` that uses our `Vec3D` type.

```
int main() {
  Vec3D arr[3];
  for (int i = 0; i < 3; ++i) cin >> arr[i];
  for (int i = 0; i < 3; ++i) {
    cout << arr[i] << endl;
  }
}
```