Nothing is Better than Copy or Move

Roger Orr

OR/2 Limited

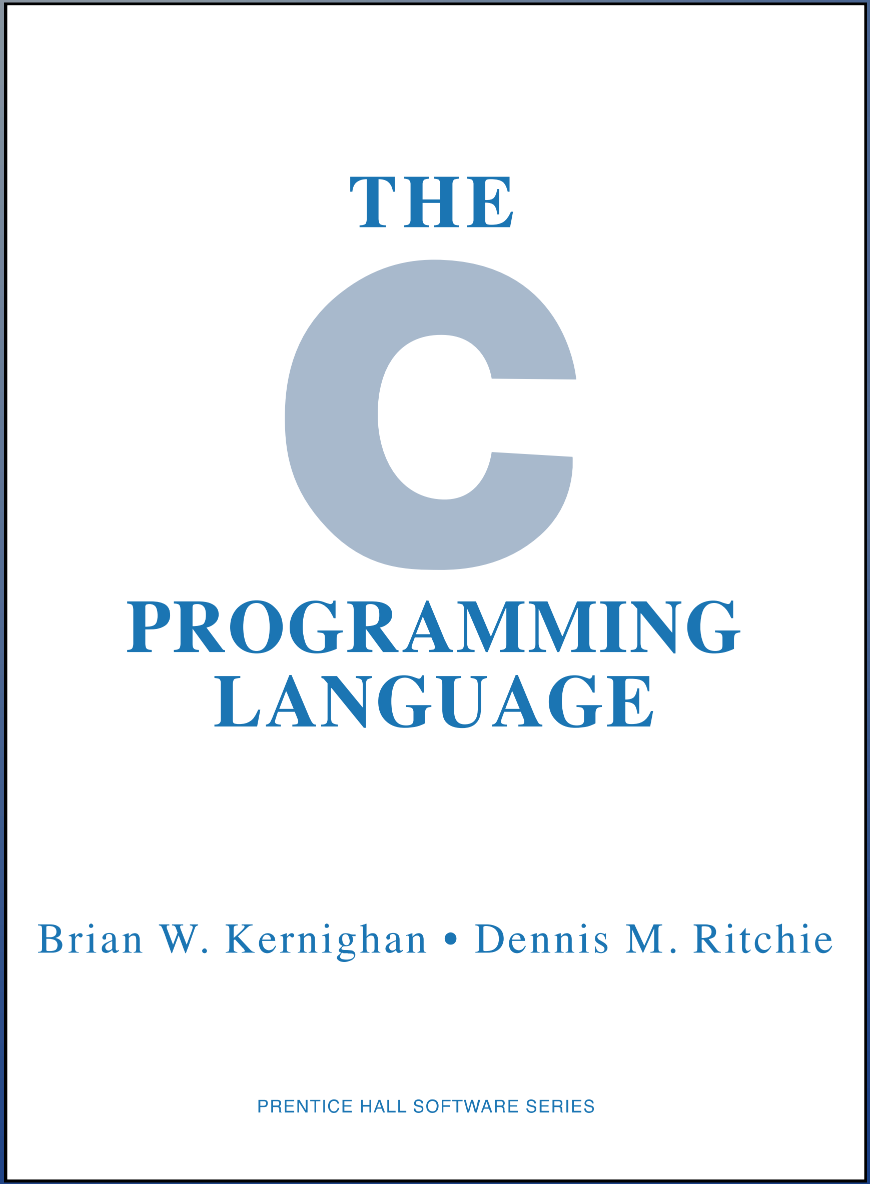
Handling the cost of copies of objects

– ACCU 2018 –

How did we get here?

- At the Jun 2016 meeting of WG21 in Oulu we approved P0135R1 ("Wording for guaranteed copy elision through simplified value categories") for C++17
- This paper removed the oddity that, for a copy to be elided, the elided copy or move constructor had to be valid
    - It may also have been a good excuse for adding a new term to the standard: “temporary materialization”
- But to explain it we need some context...

How did we get here?



THE

C

PROGRAMMING LANGUAGE

Brian W. Kernighan • Dennis M. Ritchie

PRENTICE HALL SOFTWARE SERIES

C++ is of course based on C.

This first edition (1978) says:

6.2 Structures and Functions

"There are a number of restrictions on C structures. The essential rules are that the only operations that you can perform on a structure are take its address with &, and access one of its members.

This implies that structures may not be assigned to or copied as a unit, and that they can not be passed to or returned from functions."

How did we get here?

- So in K&R C the only option for argument passing and return was to use **pointers**.
- This is mapping into a high level language the assembly language model of passing and returning addresses of data structures in registers
- Some of the C runtime functions still follow this model
    - `struct tm *localtime(const time_t *timer);`
- And of course this is still valid in C++

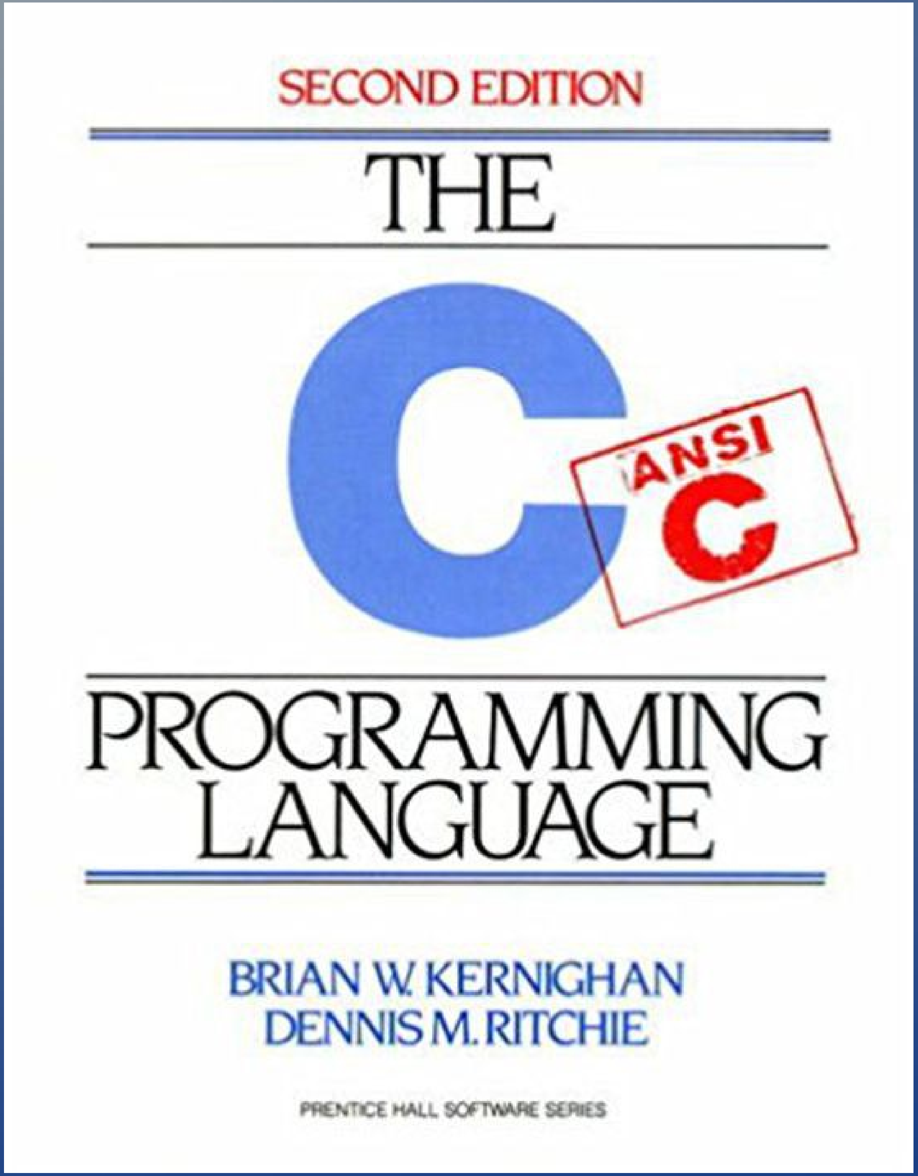
How did we get here?

- Some **benefits** using pointers-to-objects
  - Efficiency, since pointer values can be passed in registers
  - Only one object, so transfer cost is independent of size
  - Can use opaque (incomplete) types, such as `FILE*`, where the client need not know the full type
  - Simple call/return interface, typically one machine word per argument and a register return value. Note that K&R declarations didn't include arguments

How did we get here?

- Some **problems** using pointers-to-objects
    - Where does the object live (and how to tell?)
        - *Local* objects should not be returned or you get a so-called dangling pointer
        - *Heap* objects have to be managed to ensure their deletion - who owns the object?
        - *Static* objects cause problems with reuse in multiple calls (can usually be controlled, with care, in a single-threaded program – much harder to manage with threads)
    - Pointers might be invalid
    - Pointers might be `null` – whether or not the semantics allow this

How did we get here?



The first edition (1978) *also* said:

“These restrictions will be removed in forthcoming versions”

So, when ANSI C arrived in 1989:

“The main change made by the ANSI standard is to define structure assignment - structures may be copied and assigned to, passed to functions, and returned by functions.

This has been supported by most compilers for many years, but the properties are now precisely defined. Automatic structures and arrays may now also be initialized.”

(Many C programmers didn't appear to notice this enhancement...)

How does this magic work?

- We're used to passing structures by value but we may not think about how it *works*
- Passing in an argument is the easier one
  - Reserve stack space
  - Copy the argument into the reserved space
- The copy is created *before* calling the function, in stack space to be used by the called function
  - In some calling conventions the called function finds the structure on the stack, others explicitly pass in the address

How does this magic work?

- For example, with a simple stack based calling convention:

```
void foo(int arg1, example arg2, int arg3);

void bar()
{
  example ex = { /* … */ };
  foo(1, ex, 2);
}
```

  - push '2'
  - reserve sizeof(example) bytes on the stack
  - copy in 'ex'
  - push 1
  - call foo

How does this magic work?

- Most 64bit calling conventions use **registers** for the first few arguments

```
foo(1, ex, 2);
```

  - reserve sizeof(example) bytes on the stack
  - copy in 'ex'
  - load 1 into regA
  - load **address** of the reserved bytes into regB
  - Load 2 into regC
  - call foo

How does this magic work?

- You can easily see this in action by printing the relative addresses of the arguments:

```
void foo(int arg1, example arg2, int arg3);
```

- 32-bit:

```
addresses relative to argument 1:
argument 2: 4
argument 3: 28
```

- 64-bit:

```
addresses relative to argument 1:
argument 2: 32
argument 3: 16
```

How does this magic work?

- Returning a structure is more difficult
- You are typically returning a local variable, existing on the stack of the called function
- By the time the function has returned the local variable has gone!
- Typically an implementation:
  - Reserves stack space in the **caller** for the return value
  - Passes the address of this space to the called function as a (hidden) argument

How does this magic work?

- Made-up pseudo-code for:
  - T foo(int arg) { /*...*/ return local_var; }
- Caller:

```
T x = foo(42);

char __return_udt[sizeof(T)];
foo(__return_udt, 42);
T x;
memcpy(&x, __return_udt, sizeof(T));
```

- Callee:

```
void foo(void * __return_udt, int arg)
{
  /* … */
  memcpy(__return_udt, &local_var, sizeof(T));
}
```

Early days of C++

- C++ was originally called “C with Classes”
- Of particular interest here structures and classes could have special member functions:
    - Constructor
    - Copy constructor
    - Assignment operator
    - Destructor
- These user-defined functions replaced the default memcpy-like behaviour of ANSI C

Early days of C++

- While the member functions enable the preservation of invariants, they can make copying and assigning objects considerably more expensive
- Additionally, while the compiler *might* be able to remove the calls to the functions during optimization this is often not possible
- In particular, if the calls have (or might have) side effects the optimiser cannot remove them

Early days of C++

- In the ARM (1990) “Temporary Elimination”:
  - The fundamental rule is that the introduction of a temporary object and the calls of its constructor/destructor pair may be eliminated if the only way the user can detect its elimination or introduction is by observing side effects generated by the calls.

Early days of C++

- In the ARM (1990) “Temporary Elimination”:
  - The fundamental rule is that the introduction of a temporary object and the calls of its constructor/destructor pair **may** be eliminated if the only way the user can detect its elimination or introduction is by observing side effects generated by the calls.

Let's have a C++98 example...

```
--- lifecycle.h ---
#pragma once

// each method simply logs that it was called
struct lifecycle
{
  lifecycle();
  lifecycle(lifecycle const&);
  lifecycle& operator=(lifecycle const&);
  ~lifecycle();
};

--- main program ---
#include "lifecycle.h"

int main()
{
  lifecycle x = lifecycle(); // default ctor and copy ctor
  return 0;
}
```

What do you expect this program to print?

Let's have a C++98 example...

```
--- main program ---
#include "lifecycle.h"

int main()
{
  lifecycle x = lifecycle();
  return 0;
}
```

What do you expect this program to print?

From the wording of the standard you expect to see something like this:

```
0018FF44 default ctor
0018FF43 copy ctor  (where this might be elided)
0018FF44 dtor
0018FF43 dtor  (where this might be elided)
```

Let's have a C++98 example...

```
--- main program ---
#include "lifecycle.h"

int main()
{
  lifecycle x = lifecycle();
  return 0;
}
```

What do you expect this program to print?

Every C++ compiler*, even **without** optimisation, prints something like this:

```
0018FF44 default ctor
0018FF44 dtor
```

(*that I have tested...)

However C++ did not **mandate** this

```
--- main program ---

struct A
{
  A(std::string v) {}
  lifecycle l;
};

int main()
{
  A x = A("s");
}
```

With an earlier version of MSVC I obtained:

```
0095F8BE default ctor
0095F8BF copy ctor
0095F8BE dtor
0095F8BF dtor
```

Another C++98 example...

```
#include "lifecycle.h"

struct A
{
  A(int) {}
  lifecycle logger;
};

A source()
{ return 42; } // create a temporary, return by copy

int main()
{
   A x = source();
   return 0;
}
```

How about this one?
Would it be different with optimisation on?

Another C++98 example...

```
A source()
{ return 42; } // create a temporary, return by copy

int main()
{
   A x = source(); // copy-construct 'x'
   return 0;
}
```

Every compiler I've tried, with or without optimisation, prints:

```
0xffffcbdf default ctor
0xffffcbdf dtor
```

A third C++98 example...

```
// struct A as before

A source()
{  return A(42); } // create temporary, return by copy

void sink(A a)
{ return; }

int main()
{
   sink(source()); // pass copy of returned value to 'sink'
   return 0;
}
```

How about this one?
Would it be different with optimisation on?

A third C++98 example...

```
// struct A as before

A source()
{  return A(42); } // create temporary, return by copy

void sink(A a)
{ return; }

int main()
{
   sink(source()); // pass copy of returned value to 'sink'
   return 0;
}
```

Even un-optimised VC 6 (I still have a copy!) gives:

```
0018FF3C default ctor
0018FF3C dtor
```

Named Return Value

```
// struct A as before

A source()
{
  A result(42); // create object
  return result; // return by copy
}

int main()
{
   A x = source(); // copy-construct x
   return 0;
}
```

How about this one?
Would it be different with optimisation on?

Named Return Value

This one depends …

MSVC 6, even with /Ox optimisation:

```
0018FF30 default ctor
0018FF44 copy ctor
0018FF30 dtor
0018FF44 dtor
```

Gcc 5.4 -std=c++98, even with no optimisation:

```
0xffffcbff default ctor
0xffffcbff dtor
```

Why the difference?

Named Return Value

- The earlier examples all eliminated an **un-named** object. The temporary objects have no identifier and their addresses could not be taken: the only way to detect if a separate object was or was not used is by examining the constructor and destructor side-effects
- This example eliminates a **named** object and we **can** detect this by comparing the addresses of the named objects

Named Return Value Optimisation

- C++ explicitly allows a compiler to use the named return value as the actual return value:

  “if the expression in the return statement is the name of a local object, …, an implementation is **permitted** to omit creating the temporary object to hold the function return value”

- NRVO (as it is known) is optional – and it may not be obvious whether or not it has been invoked

What about now?

- C++11 introduced move semantics into the language – how does this change things?

```
struct lifecycle
{
  lifecycle();
  lifecycle(lifecycle const&);
  lifecycle& operator=(lifecycle const&);
  ~lifecycle();

  lifecycle(lifecycle &&); // move-construction
  lifecycle& operator=(lifecycle &&); // move-assignment
};
```

- Named return values are effectively r-values

Let's revisit an example, for C++11

```
// struct A as before

A source()
{
  A result(42); // create object
  return result; // can return by move
}

int main()
{
   A x = source();
   return 0;
}
```

The named returned value is now an r-value*

*effectively: its copy can become a move operation, so the copy construction in C++03 becomes move construction in C++11 (assuming no NRVO)

Can actually break C++03 code

```
struct foo
{
  foo(std::string& str) : _str(str) {}

  ~foo() { std::cout << "Ending \"" << _str << "\"." << std::endl; }

  std::string& _str;
};

std::string foobar()
{
  std::string s("foobar");
  foo f(s);

  return (s); // We may get NRVO
}
```

The order of events without NRVO is: move from s, destroy f, destroy s. So the moved-from s is printed...

This can be a problem with scope-guard like helpers

Temporary Materialization

- In C++14 and before 'copy elision' was used to describe the permission to directly construct an object in cases where the syntax implied a copy/move construction from a temporary object
- As we've seen, *general* existing practice of most compilers was to perform copy elision *even when not optimising* (this is because a syntactic analysis is needed to determine whether it is valid to perform it)
- However, the wording still made it optional

Temporary Materialization

- C++17 instead talks about prvalues* and describes the rules for when these are used to initialise an actual object
- Converting such a prvalue into an object is called a **temporary materialization conversion**
- The new wording on these conversions makes it clear what behaviour is **required**

*A prvalue (“pure r-value”) is an expression whose evaluation initializes an object

Temporary Materialization

- These rules also cover temporary objects generated during expression evaluation. Common examples of when a prvalue materializes into an object are:
  - 1. Initialisation of a variable
  - 2. Binding to a reference
  - 3. Performing member access
- For example in this contrived example:

```
auto a = string("A") + string("B").c_str();
    1^              2^             3^
```

C++17 does now **mandate** this

```
--- main program ---

struct A
{
  A(std::string v) {}
  lifecycle l;
};

int main()
{
  A x = A("s"); // temporary materialization: no actual copy or move
}
```

A C++17 compiler **must** produce output similar to this:

```
0095F8BE default ctor
0095F8BE dtor
```

A simplification

```
struct noncopyable
{
  noncopyable() = default;
  noncopyable(noncopyable const &) = delete;
 };

int main()
{
  noncopyable anObject = noncopyable();
}
```

In C++14 and before this was an **error** as the copy constructor was marked as deleted – even though the compiler was going to elide the call to it!
The materialization conversion to form anObject does not require an accessible copy constructor – direct construction is required

Let's revisit that example...

```
// struct A as before

A source()
{
  A result(42); // create object
  return result; // can return by move
}

int main()
{
   A x = source(); // temporary materialization conversion into x
   return 0;
}
```

There is a second change in up-to-date C++:
- the materialization conversion to form x does not permit copy/move from a temporary

Don't be too clever!

```
// struct A as before

A source()
{
  A result(42); // create object
  return std::move(result);
}

int main()
{
   A x = source(); // temporary materialization into x
   return 0;
}
```

We've added an explicit move to the named return value. What does this change?

Don't be too clever!

```
// struct A as before

A source()
{
  A result(42); // create object
  return std::move(result); // Bad: NRVO is disabled by use of std::move
}

int main()
{
   A x = source(); // temporary materialization into x
   return 0;
}
```

Nothing is better than move

Without std::move in the worst case we get the move constructor called, in the best case we invoke NRVO and don't need to move – std::move here means we **must** move!

Another example...

```
A source()
{ return 42; } // create an un-materialized temporary

int main()
{
   A x = source(); // materialization conversion into 'x'
   return 0;
}
```

C++17 *guarantees* this prints:

```
0xffffcbdf default ctor
0xffffcbdf dtor
```

Syntactically you no longer need the copy-constructor

In implementation, this was mostly just standardising existing practice by compiler vendors

One bad object poisons the move

```
struct copyable
{
  copyable() = default;
  copyable(copyable const &) {}
};

struct A
{
  A(int) {}
  lifecycle logger;
  copyable c;
};
```

How about this one? The copyable structure is trivial – there's no obvious reason to write move constructor

However, this *disables* the automatically generated move constructor for types, such as A, that hold a member of this type

What are the rules?

- Those who attended ACCU 2014 may recall Howard Hinnant's closing keynote which was:

  Everything You Ever Wanted To Know About Move Semantics (and then some)

- I commend his presentation to you for a summary “from the horse's mouth.”
- He produced one summary slide which I reproduce on the next slide, summarising in a matrix the rules about the effect of declaring special members on the defaults

Special Members

compiler implicitly declares

user declares

	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

What are the rules?

- The **simplest** rule applies when you don't declare *any* special member functions but are able to use the defaults, generated from the special member functions of the members and bases of your class
- If you have some data needing manual management can you use composition with a helper object?
- The next simplest rule is to either `=default` or `=delete` all the special members you don't explicitly declare

Comparison of copy and move

- Many people incorrectly think of move as effectively 'free'
- The performance difference between copy and move varies widely
- For a primitive type, such as int, copying or moving are effectively **identical**

```
int i = 5;
j = i;
k = std::move(i);
```

- If i has any value other than 5 after the last line, move is **costlier** than copy!

Comparison of copy and move

- So for a data structure with primitive fields copy and move are basically the same
  - std::array<int, 1000> s;
- Move is faster than copy when only **part** of the object needs to be transferred to transfer the whole value
- Typical examples of this are fields that are
  - Pointers
  - Handles
- Copying means allocating memory or an OS call

Comparison of copy and move

- However even when using a move constructor or assignment, the pointer value and possibly some other control data, must still be copied which involves memory access
- Hence the desire to improve the wording in C++17 to make it explicit when temporary objects are actually instantiated

How do I recognise copy and move?

- The C++ standard can seem quite complicated, but fortunately in many cases a few simple rules can help with copying and moving
- For example, Abseil “tip of the week” #77:
- Two Names: It’s a Copy
    - a second name for the same data
- One Name (at once!): It’s a Move
    - you can’t refer to a name any more
- Zero Names: It’s a Temporary

Avoiding copy and move

- The older style of passing and returning pointers can be more efficient for objects that don't need modifying
- Passing a larger object by const reference may well be better than copying it
- Smaller, trivially copyable objects may be optimal already – let's look at the special rules to allow this

Small trivial objects

- There are special rules in C++17 for passing arguments that are trivially copyable and deletable allowing the compiler to create copies
- This exception from the rules of temporary materialization (which would otherwise forbid such additional copies) is to allow such objects to passed to, or returned from, functions in registers
- Each ABI will have its own restrictions on when this applies

MSVC x86/x64 convention

- The Microsoft 32-bit calling convention allows suitable structures up to 8 bytes wide to be returned in EAX and EDX (if needed)
- The Microsoft 64-bit calling convention allows suitable structures of 1, 2, 4 or 8 bytes to be both supplied and returned in registers
- (The 64-bit calling convention prefers the use of registers over the stack, for speed)

Linux x86/x64 convention

- The 32-bit calling convention allows suitable structures up to 4 bytes in size to be returned in registers
- The 64-bit calling convention allows suitable structures of 1, 2, 4, 8 or 16 bytes to be both supplied and returned in registers
- (Again, the 64-bit calling convention prefers the use of registers over the stack, for speed)

Should I care?

- For most programmers, most of the time, this is not very relevant
- However, if you are trying to squeeze out the last bit of performance the extra indirection of passing the address of an argument in memory rather than directly in a register may matter
- Check the ABI for your chosen platform(s) carefully

Passing arguments to functions

- When passing input-only arguments to functions in C++ there are two choices:
  - Pass by value
  - Pass by const reference

Passing arguments to functions

- When passing input-only arguments to functions in C++ there are three choices:
    - Pass by value
    - Pass by const reference
    - Pass by pointer

Passing arguments to functions

- When passing input-only arguments to functions in C++ there are four choices:
  - Pass by value
  - Pass by const reference
  - Pass by pointer
  - Pass by accessor, such as a smart pointer
- How do I choose ???

Returning or passing a reference?

- Before C++11 this idiom was common:
```
void foo(std::vector<std::string> &result);
```
- Since C++11 this is more normal:
```
std::vector<std::string> foo();
```
- The first idiom can still be useful if the object to return is expensive to move, although in that case using a unique_ptr<> may be another approach
- A drawback of the first approach is the preconditions on 'result' are unclear

Avoiding copy and move

- When a method takes a **copy** of an argument it may be better to take the argument by **value** and move it into the target

-
  ```
  void addName1(std::string const &name) {
    collection.add(name); // Takes a copy
  }
  ```

-
  ```
  void addName2(std::string name) {
    collection.add(std::move(name));
  }
  ```

Avoiding copy and move

- If the call site needs to construct the argument the second call can avoid a copy:
- a.addName1("Rectilinear")
  - Create an std::string and pass it into addName, which then **copies** it
- a.addName2("Rectilinear")
  - Create an string and pass it into addName2, which then **moves** it
- Note: with the *small string optimisation*, the cost of copy and move may be the same...

Avoiding copy and move

- Alternatively, you can provide **overloads** taking const& and &&

-
  ```
  void addName1(std::string const &name) {
    collection.add(name); // Copies
  }
  ```

-
  ```
  void addName2(std::string &&name) {
    collection.add(std::move(name)); // Moves
  }
  ```
- This is optimal, but for 'n' arguments you'll need 2^n overloads each with very subtly different code...so only do it if the need is really there

Avoiding copy and move

- Passing **shared pointers** as arguments can result in extra copies
- While this doesn't copy the **payload** it does mean extra calls to (**atomically**) increment and decrement the use count
- Core guidelines R.30: “Take smart pointers as parameters only to explicitly express lifetime semantics”
    - For example, the method takes a **copy** of the shared pointer for use later

Avoiding copy and move

-
  ```
  void action1(std::shared_ptr<foo> arg) {
    arg->do_something();
  }
  ```

-
  ```
  void action2(foo &arg) {
    arg.do_something();
  }
  ```
- The first case needs code to create and destroy the supplied arg, and also needs state management to ensure clean stack unwinding in the presence of an exception
- The second case also expresses intent directly (arg should not be null)

Avoiding copy and move

-
  ```
  void test1(std::shared_ptr<foo> &p) {
    action1(p);
  }
  ```

-
  ```
  void test2(std::shared_ptr<foo> &p) {
    action2(*p);
  }
  ```
- With clang 6.0.0 and -O2, test1 produces **94** assembler instructions …

Avoiding copy and move

-
  ```
  void test1(std::shared_ptr<foo> &p) {
    action1(p);
  }
  ```

-
  ```
  void test2(std::shared_ptr<foo> &p) {
    action2(*p);
  }
  ```
- With clang 6.0.0 and -O2, test1 produces **94** assembler instructions
- But test2 produces just **two** instructions

```
    mov     rdi, qword ptr [rdi]
    jmp     action2(foo&)              # TAILCALL
```

More on NRVO

- NRVO is quite fragile – but there are reasons for that which are clearer once you have some understanding of how it works
- Multiple return values are one example:

-
  ```
  std::string foo(int i) {
    std:string result("X");
    if (i < 0) return {};
    // ...
    return result; // No NRVO
  }
  ```
- The first return constructs into the target, so result cannot *also* be constructed there

More on NRVO

- Function arguments are **not** available for NRVO:

-
  ```
  A foo(A input) {
    return input; // No NRVO
  }
  ```
- The reason for this restriction is because of the implementation.
- The *caller* constructs input, but cannot in general know whether or not the target returns this argument, so it must use a distinct location from the return object

More on NRVO

- Function arguments are **not** available for NRVO:

-
  ```
  A foo(A input) {
    return input; // move construction
  }
  ```
- However, C++17 does **move** the argument (if a move constructor is accessible)

Conclusion

- Since C++11 there has, rightly, been much focus on adding support for move to code
- However, C++ was already quite good at eliminating copies in various cases and this has improved further with C++17
- While move is often faster than copy, it is not usually free
- Doing nothing is better than either copy or move
- Check the number of objects being created before simply adding `std::move()`