(some of the) new features in modern C++

rvalue references and move semantics

- most important language change
  - much better efficiency in certain situations
  - not just a library addition or syntactic sugar

a 'reference' is now known as an 'lvalue reference'

it binds to a value with a known location

```
SomeClass obj;              //  this is on the stack
SomeClass& obj_ref = obj;   //  an lvalue reference
```

some values don't have known locations

```
SomeClass func(SomeClass x, SomeClass y) {
    ...         //  do clever stuff with x and y
    return SomeClass(i, j, k, l);
}
SomeClass x, y;
5;                                  //  result is temporary
x + y;                              //  same story here
func(x, y);                         //  and here
std::string("this is a string");    //  and here
```

```
int& ref = 5;                           //  nonsense!
SomeClass& ref = x + y;                 //  nonsense!
SomeClass& ref = func(x, y);            //  nonsense!
std::string& ref = std::string("eh");   //  nonsense!
```

what if we want a reference to a temporary?

(don't worry about why yet…)

use an rvalue reference

```
int&& r_ref = 5;                //  fine!
SomeObject&& r_ref = x + y;     //  fine!
SomeObject&& r_ref = func();    //  && means 'rvalue reference'
std::string&& r_ref = std::string("eh");
```

how is this useful?

sometimes we want to move values in or out of certain scopes

especially temporary values

```
class VectorWrapper {
    std::vector<int> vec;
public:
    VectorWrapper(std::vector<int> vec) //  hmmm
    : vec(vec) {}
    //  some methods that modify vec
};

int main() {
    std::vector<int> vec(1 << 30);  //  probably 4MB or so
    ...                             //  initialise vec values
    VectorWrapper vw(vec);          //  will copy vec (expensive)
    ...                             //  call methods on vw
    ...                             //  we don't care about vec here
```

```
    return 0;
}
```

what happens if we store an lvalue reference instead?

```
class VectorWrapper {
    std::vector<int> & vec;
public:
    VectorWrapper(std::vector<int> & vec)   //  no copies, good
    : vec(vec) {}
    //  some methods that modify vec
};

VectorWrapper get_wrapper() {
    std::vector<int> vec(1 << 30);
    ...                                     //  setup
    return VectorWrapper(vec);
}

int main() {
    VectorWrapper vw = get_wrapper();       //  unspeakable evil
    return 0;
}
```

vec gets destructed when it goes out of scope

VectorWrapper's reference points at invalid memory!

what if we don't need to set up the vector?

```
VectorWrapper get_wrapper() {
    //  error here - can't bind (mutable) reference to temporary
    return VectorWrapper(std::vector<int>(1 << 30));
}
```

when copies are expensive, we want to be able to move the object without doing a full copy and without danger of invalid references or pointers

```
class VectorWrapper {
    std::vector<int> vec;                   //  no reference here
public:
    VectorWrapper(std::vector<int> && vec)  //  rvalue ref
    : vec(std::move(vec)) {}                //  call move constructor
    //  some methods that modify vec
};

//  vector constructor returns a temporary like any other function
//  we pass a *reference to the temporary* to VectorWrapper()
VectorWrapper get_wrapper() {
    return VectorWrapper(std::vector<int>(1 << 30));
}
```

```
class VectorWrapper {
    std::vector<int> vec;                   //  no reference here
public:
    VectorWrapper(std::vector<int> && vec)  //  rvalue ref
    : vec(std::move(vec)) {}                //  call move constructor
    //  some methods that modify vec
};

//  here, vec is an lvalue (it has a name) but we want to transfer
//  ownership to VectorWrapper()
```

```cpp
VectorWrapper get_wrapper() {
    std::vector<int> vec(1 << 30);
    ...     // set up vec
    return VectorWrapper(std::move(vec));
}
```

what's this `std::move()`?

just takes an lvalue or rvalue and returns an rvalue without triggering copy construction

it's the same as `static_cast<T&&>(t)`

it doesn't actually do any moving! this is handled by overloading functions on rvalue types

the magic happens in the 'move constructor'

all STL data structures have move semantics since C++11

- the move constructor
  - takes a reference to a temporary (or something that has been cast to a temporary)
  - takes ownership of the temporary's internals, and nulls-out the temporary

for vector, the move constructor will take ownership of the temporary's data pointer and will set that of the temporary to `nullptr`

copying a single pointer is much cheaper than allocating memory and copying the entire vector

how does it work? what does this look like?

```cpp
template<typename T> struct MyVec {
    int n; T* ptr;
    MyVec(int size = 0): n(size), ptr(n ? new T[n] : nullptr) {}
    virtual ~MyVec() noexcept { delete[] ptr; }
    // copy semantics
    friend void swap(MyVec& a, MyVec& b) {
        std::swap(a.n, b.n); std::swap(a.ptr, b.ptr);
    }
    MyVec(const MyVec& m): n(m.n), ptr(n ? new T[n] : nullptr) {
        std::copy(m.ptr, m.ptr + m.n, ptr);
    }
    MyVec& operator=(MyVec m) { swap(*this, m); return *this; }
    // move semantics
    MyVec(MyVec&& m): MyVec() { swap(*this, m); }
    MyVec& operator=(MyVec&& m) { swap(*this, m); return *this; }
};
```

to move into a scope, we use `std::move` and function overloading on rvalues

```cpp
template<typename T>
void sink_parameter(T&& t) {
    // do something with t
}

int main() {
    ExpensiveToCopyObj obj;
    ...         // set up obj
    sink_parameter(std::move(obj));
    ...         // we don't need obj here
}
```

what about moving out of scopes?

if an object has a move constructor it will be moved out of the function

else if it has a copy constructor it will be copied else it is an error!

```
ExpensiveToCopyObj get_expensive_obj() {
    ExpensiveToCopyObj obj;
    return obj;
}

int main() {
    ExpensiveToCopyObj obj = get_expensive_obj();   //  uses move constructor
}
```

although! most modern compilers are smart

'copy elision' is used to construct the object in the returning scope

```
ExpensiveToCopyObj get_expensive_obj() {
    ExpensiveToCopyObj obj;
    return obj;
}

int main() {
    //  obj is actually directly constructed here - no move/copy!
    ExpensiveToCopyObj obj = get_expensive_obj();
}
```

that's the fundamentals - the important points:

- rvalue references can be used to reference temporary values
- this lets us overload on temporary values, to treat them differently
  - we can 'steal' their data members, bypassing costly copies
- to treat an lvalue as an rvalue (to move it into another scope), use `std::move`
- *std::move on its own doesn't do anything*, it enables the compiler to choose a different function overload

auto

```
int x;
//  some code that uses x
```

x might be uninitialised

```
std::vector<float> vec{1.0f, 2.0f, 3.0f, 4.0f};
for (std::vector<float>::const_iterator i = vec.begin(); i != vec.end(); ++i) {
    //  do something with i
}
```

that iterator type is cumbersome

it gets worse if you're passing iterators to functions...

```
template<typename It>
void iterate(It b, It e) {
    for (; b != e; ++b) {
        typename std::iterator_traits<It>::value_type deref = *b;
        //  do something with deref
    }
}
```

```
std::vector<float> vec{1.0f, 2.0f, 3.0f, 4.0f};
unsigned s = vec.size();
```

actual size should be `std::vector<float>::size_type`

on windows x64, `sizeof(unsigned) == 4`, `sizeof(std::vector<float>::size_type) == 8`

```
??? = [] (auto i) {i.do_one_thing(); i.do_another_thing();};
```

the types of closures are compiler-internal, can't be written

`auto` variables have their type deduced from their initialiser

```
auto x;                                          //  won't compile
auto a = 5;                                      //  fine
auto b = {"some", "list", "of", "items"};        //  fine
auto c = return_value_of_function();             //  fine
```

why use auto?

must be initialised

immune to type mismatches (especially when working with templates)

faster to write (can become muscle memory)

language features for OOP

override

```
struct Base {
    virtual void do_thing();        //  virtual functions
    virtual void get_thing() const; //
};

struct Derived : public Base {
    void do_thing();                //  overrides Base::do_thing
                                    //  we don't know this unless
                                    //  we look at the interface
                                    //  for Base

    void get_thing();               //  forgot to mark as const
                                    //  won't override!
};
```

what if we misspell the name of the overriding function?

we'll call the base class function, but we won't know until runtime!

the solution: mark overriding functions `override`

```
struct Derived : public Base {
    void do_thing() override;       //  now we know this is
                                    //  overriding something
                                    //  in Base

    void get_thing() override;      //  this will give us a
                                    //  compiler warning
};
```

`override` makes your intent clearer, and helps the compiler flag up mistakes

special member functions

default constructor

copy and move constructors

copy and move assignment operators

destructors

what does the compiler normally give us (if we don't write any of these ourselves)?

```
struct Base {
    Base();                              // default constructor
    ~Base() noexcept;                    // destructor
    Base(Base&&);                        // move constructor
    Base& operator=(Base&&);             // move assignment
    Base(const Base&);                   // copy constructor
    Base& operator=(const Base&);        // copy assignment
};
```

avoiding pitfalls with compiler-generated members

most of the time, you want to use the compiler-generated methods

but, the compiler is picky about when it will generate them for you!

default constructor - generated if we don't write *any* other constructors

destructor - *not virtual by default!*

move operations - generated if we don't define copy, move, or destruct operations

copy constructor - generated if we don't define it *or* any move operation

copy assignment - same as copy constructor

```
struct Base {
    virtual float get_number() const = 0; // warning:
                                          // no virtual destructor!
};
class Derived : public Base {
    float data_member;
public:
    Derived(float a): data_member{a} {}
    float get_number() const override {return data_member;}
};
Base * ptr = new Derived(10.0f); // Base doesn't have a virtual
delete ptr;                      // destructor, so the destructor
                                 // for Derived is never called
```

in that example, the compiler-generated destructor would be fine, if it was virtual

```
struct Base {
    virtual ~Base() noexcept = default; // use default destructor,
                                        // but virtual
    // because we've defined our own destructor, the compiler won't
    // generate copy or move operations now!
    // we tell the compiler to use the copy and move methods it
    // would have generated anyway
    Base(Base&&) = default;
    Base& operator=(Base&&) = default;
    Base(const Base&) = default;
    Base& operator=(const Base&) = default;
};
```

most base classes should look like this (generated destructors in derived classes will be virtual)

if you want to explicitly deny copy or move construction, you can use `= delete`

```
struct Base {
    // you can't move or copy instances of this class
    Base(Base&&) = delete;
    Base& operator=(Base&&) = delete;
    Base(const Base&) = delete;
    Base& operator=(const Base&) = delete;
};
```

in the past you'd achieve this by making the definitions private and not providing implementations, but `= delete` is a clearer approach

if you know you'll never need to derive from a class, mark it `final`

```
struct DontDerive final {
    float a_data_member;
    std::string another_data_member;

    // this class will not have a generated virtual destructor
    // but that's fine because the compiler won't let us derive
    // from this class now - we won't get 'slicing' leaks
};
```

smart pointers

```
SomeObject * ptr = some_function_returning_a_pointer();
```

is ptr a single object, or an array?

do we own what `ptr` points to?

if we do, how do we dispose of it? `delete`? `obliterate(SomeObject*)`?

are there multiple paths that might all try to delete `ptr`?

post C++11, assume a raw pointer is non-owning

owned memory on the heap should be managed by a smart pointer

`std::unique_ptr`

has exclusive ownership of the memory it points to

can't be copied, only moved

(copying an 'owning' pointer is hard - when do you call the destructor?)

```
std::unique_ptr<SomeObject> ptr(new SomeObject);    // C++11
auto ptr = std::make_unique<SomeObject>();          // C++14

{
    auto cpy = ptr;             // compiler error, can't be copied
    auto mvd = std::move(ptr);  // fine, ptr now holds nullptr

    // mvd will be destroyed at end of scope like a 'normal' object
}
```

if you need a custom destructor, that's fine

for example: FFTW library has custom alloc/free functions for arrays with necessary alignment

```
struct fftwf_ptr_destructor {
    template <typename T>
    void operator()(T t) const noexcept { fftwf_free(t); }
```

```
};

using fftwf_r = std::unique_ptr<float, fftwf_ptr_destructor>;
using fftwf_c = std::unique_ptr<fftwf_complex, fftwf_ptr_destructor>;

{
    //  use the FFTW malloc functions to do magic allocation
    fftwf_r real(fftwf_alloc_real(...));
    fftwf_c cplx(fftwf_alloc_complex(...));
    //  we know the ownership of the memory here
    //  we don't have to remember to free it!
}
```

`std::shared_ptr`

a bit like garbage collection - keeps an internal count of the objects that are pointing to it, and is destroyed when the count reaches zero

use when several objects rely on a memory area, but there's not one clear owner

otherwise behaves like `unique_ptr` (but copy assignment is fine, memory will be destroyed when both owners have been destroyed)

```
std::shared_ptr<SomeObject> ptr(new SomeObject);    //  C++11
auto ptr = std::make_shared<SomeObject>();          //  C++14
//  reference count: 1

{
    auto another_owner = ptr;                       //  fine
    //  reference count: 2
}
//  reference count: 1 (another_owner destroyed at end of scope)
```

lambda expressions

'anonymous function' or 'function literal'

```
auto between = [](int val) { return 0 < val && val < 10; };
//             ^--- this bit is the lambda expression  --^
//                      could also be written as
auto between(int val) { return 0 < val && val < 10; }
```

several parts of a lambda expression, some are optional

```
auto x = 10;

auto func_0 = [x] (int i) -> int { return x * i; };
auto func_1 = [&x] (int i) -> int { return x * i; };

func_0(10); //  returns 100
func_1(10); //  returns 100

x = 1;
func_0(10); //  returns 100
func_1(10); //  returns 10
```

the opening square brackets are the capture list - allows you to specify whether variables from outside the lambda scope are captured by value or by reference leave them empty if you don't need to capture variables

the round brackets are the function parameters

leave them out completely if the expression doesn't need parameters

```
auto func = [] {call_one_function(); call_another_function();};
```

you can specify the return type of the lambda after an arrow `-> SomeType`

if you don't, the compiler will work out the return type for you

```
//  this function will truncate its result to an int
auto func_0 = [] (auto i) -> int { return 10.0f * i; };

//  this function will return a double
auto func_1 = [] (auto i) { return 10.0 * i; };
```

finally, the function body goes in curly brackets `{}`

lambdas are helpful for (amongst other things):

- using the standard library (`std::accumulate`, `std::find_if` etc. all take function arguments)

```
template<typename T> struct Between final { //  lives in global namespace
    Between(T a, T b): a{a}, b{b} {}
    bool operator()(T i) const {return a < i && i < b;}
private:
    T a, b;
};
{
    auto minimum = 0.0f;
    auto maximum = 10.0f;
    std::vector<int> coll{-10, -5, 0, 5, 10, 15, 20};
    auto c = std::count_if(coll.begin(),
                           coll.end(),
                           Between(minimum, maximum));
}
```

```
{
    auto minimum = 0.0f;
    auto maximum = 10.0f;
    std::vector<int> coll{-10, -5, 0, 5, 10, 15, 20};
    auto c = std::count_if(coll.begin(),
                           coll.end(),
                           [minimum, maximum](auto i){
                               return minimum < i && i < maximum;
                           });
}
```

- specifying custom deleters for smart pointers
- launching threads - run a lambda expression on another thread!
- any time you need a one-off throw-away function

other stuff

threads

`std::thread` encapsulates another thread of execution

use it when you need to run several operations concurrently

`std::async` is a way of running a task in the background, and fetching the result of that task at a later point use it when you will need a value at some point in the future, but want to carry on with something else in the meantime

```
std::atomic_bool continue = true;
std::thread t([&continue] {
    while (continue) {
        //  do something repetetive and thread-safe
    }
});

//  process user input?

//  end of program
continue = false;
t.join();
```

```
auto fut = std::async([] {return do_some_work() + do_more_work();});
//  interact with the user
//  reticulate splines

//  now we're ready to get the result of the async task
auto work = fut.wait();
```

there's a lot more to thread support in C++, but I haven't used it a great deal check en.cppreference.com/w/cpp/thread for more info

regex

yep, there's regular expressions in the standard library

nullptr

`nullptr` rather than `0` or `NULL`

`nullptr` is a pointer type, not an int (more explicit about programmer intent)

type aliases

`using` rather than `typedef`

```
typedef SomeLongComplicatedType ShortType;      //  not great...
using ShortType = SomeLongComplicatedType;      //  better

template<typename T>
using NestedVec = std::vector<std::vector<T>>;  //  can't do this with typedef
```

range-based for

```
std::vector<int> vec{1, 2, 3, 4};
for (const auto & i : vec)              //  collection has member
    std::cout << i << std::endl;        //  begin and end

int arr[] = {5, 6, 7, 8};
for (const auto & i : arr)              //  falls back to
    std::cout << i << std::endl;        //  std::begin and std::end

for (const auto & i : {9, 10, 11, 12})  //  std::begin and std::end
```

```
    std::cout << i << std::endl;        //  have overloads for
                                        //  std::initializer_list
```

what now?

read * 'Effective Modern C++' by Scott Meyers (O'Reilly Media, Inc. Canada) * `en.cppreference.com`

enable C++14 * in xcode, there's a build-settings option * in cmake, do `set(CMAKE_CXX_FLAGS ${CMAKE_CXX_FLAGS} " -std=c++14 ")`

install `clang-tidy` and use its modernize feature on your existing code

write C++14!

thanks for listening