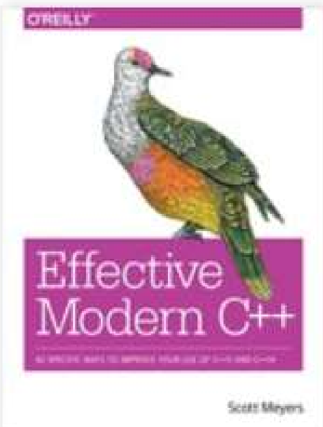
Modern C++ Explained

Xuelan Mei

Sep. 2020



Effective Modern C++

38 REVIEWS

by Scott Meyers

Publisher: O'Reilly Media, Inc.
Release Date: November 2014
ISBN: 9781491903995
Topic: C++

- Lambda Expressions in C++
- Smart Pointer
- R-value Reference
- Deducing types
- Moving to modern C++

λ

Functional Programming

Lambda Expression

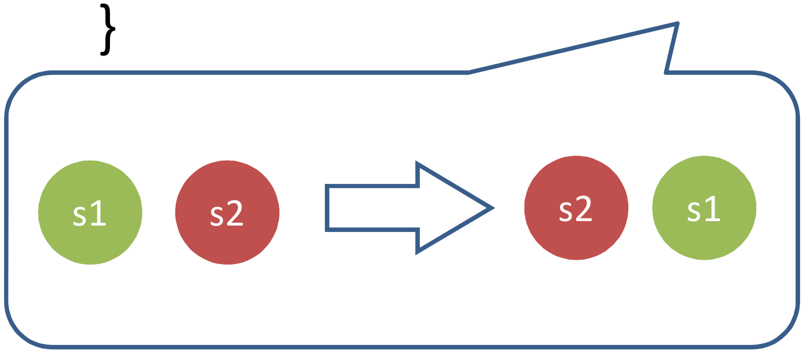
Lambda

is an anonymous function passed to another function

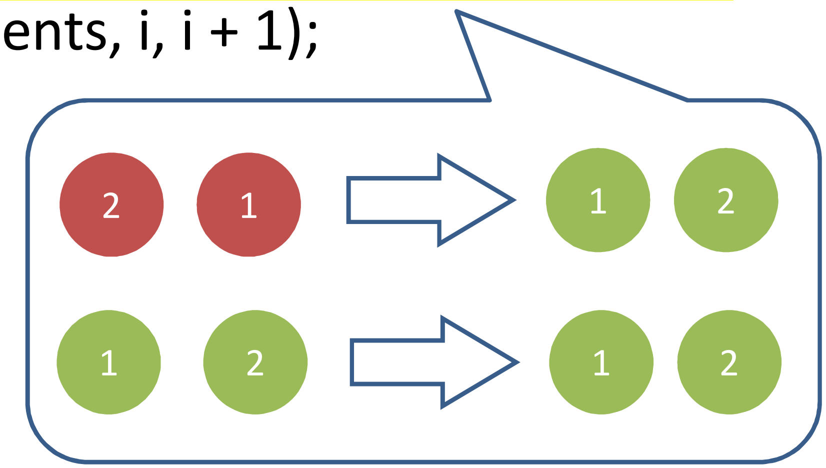
Example Bubble Sort

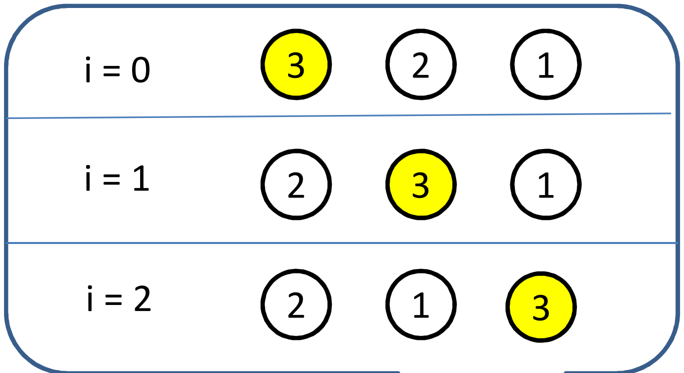
```
void bubbleSort(vector<Student> & students) {
    for (int j = students.size(); j > 0; j--)
        for (int i = 0; i < j - 1; i++)
            if (students[i].score > students[i+1].score)
                SWAP(students, i, i + 1);
}
```

```
void bubbleSort(vector<Student> & students) {
    for (int j = students.size(); j > 0; j--)
        for (int i = 0; i < j - 1; i++)
            if (students[i].score > students[i+1].score)
                SWAP(students, i, i + 1);
}
```

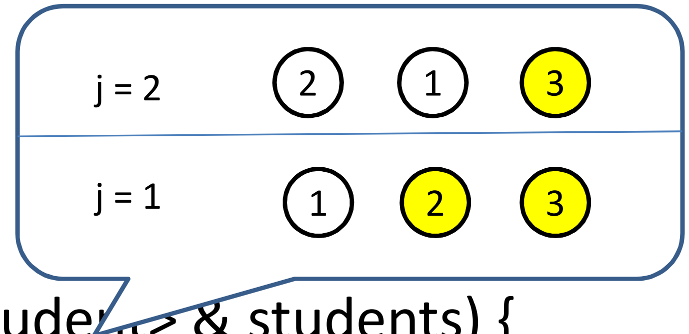


```
void bubbleSort(vector<Student> & students) {
    for (int j = students.size(); j > 0; j--)
        for (int i = 0; i < j - 1; i++)
            if (students[i].score > students[i+1].score)
                SWAP(students, i, i + 1);
}
```






```
void bubbleSo ector<Student> & students) {
  for (int j = students.size(); j > 0; j--)
    for (int i = 0; i < j - 1; i++)
      if (students[i].score > students[i+1].score)
        SWAP(students, i, i + 1);
}
```




```
void bubbleSort(vector<Student> & students) {
   for (int j = students.size(); j > 0; j--)
      for (int i = 0; i < j - 1; i++)
         if (students[i].score > students[i+1].score)
            SWAP(students, i, i + 1);
}
```

The purpose of design

PRACTICE ONE: CHANGE IS COMING

```
void bubbleSort(vector<Student> & students) {
    for (int j = headCount; j > 0; j--)
        for (int i = 0; i < j - 1; i++)
            if (students[i].score > students[i+1].score)
                SWAP(students, i, i + 1);
}
```

How can we alternate this piece of logic?

To alternate a piece of logic

- C
  - Function pointer
  - Demo
- C++ 98
  - Function object
  - Demo
- C++ 11+
  - Lambda
  - Demo

Lambda

is convenient and define right at the location

In C++11 and later, a lambda expression—often called a *lambda*—is a convenient way of defining an anonymous function object (a *closure*) right at the location where it is invoked or passed as an argument to a function.

```
[ captures ] ( params ) -> ret { body }
```

```
[ captures ] ( params ) { body }
```

1. An function passed as an argument to another function
2. Anonymous, Convenient way
3. Define right at the location

Lambda Expressions

```
[ captures ] ( params ) -> ret { body }
[ captures ] ( params ) { body }
```

The list of parameters, as in named functions. C++14, If auto is used as a type of a parameter, the lambda is a generic lambda.

```
std::find_if(container.begin(), container.end(),
             [](int val) { return 0 < val && val < 10; });
```

Lambda Expressions

```
[ captures ] ( params ) -> ret { body }
[ captures ] ( params ) { body }
```

Return type. If not present it's implied by the function return statements (or void if it doesn't return any value)

```
std::find_if(container.begin(), container.end(),
             [](int val) { return 0 < val && val < 10; });
```

Return type is bool as it can be deducted by return statement

Lambda Expressions

```
[ captures ] ( params ) -> ret { body }
[ captures ] ( params ) { body }
```

Function body

```
std::find_if(container.begin(), container.end(),
             [](int val) { return 0 < val && val < 10; });
```

Lambda Expressions

```
[ captures ] ( params ) -> ret { body }
[ captures ] ( params ) { body }
```

a comma-separated list of zero or more captures, optionally beginning with a capture-default.

```
std::find_if(container.begin(), container.end(),
             [](int val) { return 0 < val && val < 10; });
```

Capture nothing

Understand capture list

Practice Two: Lady first

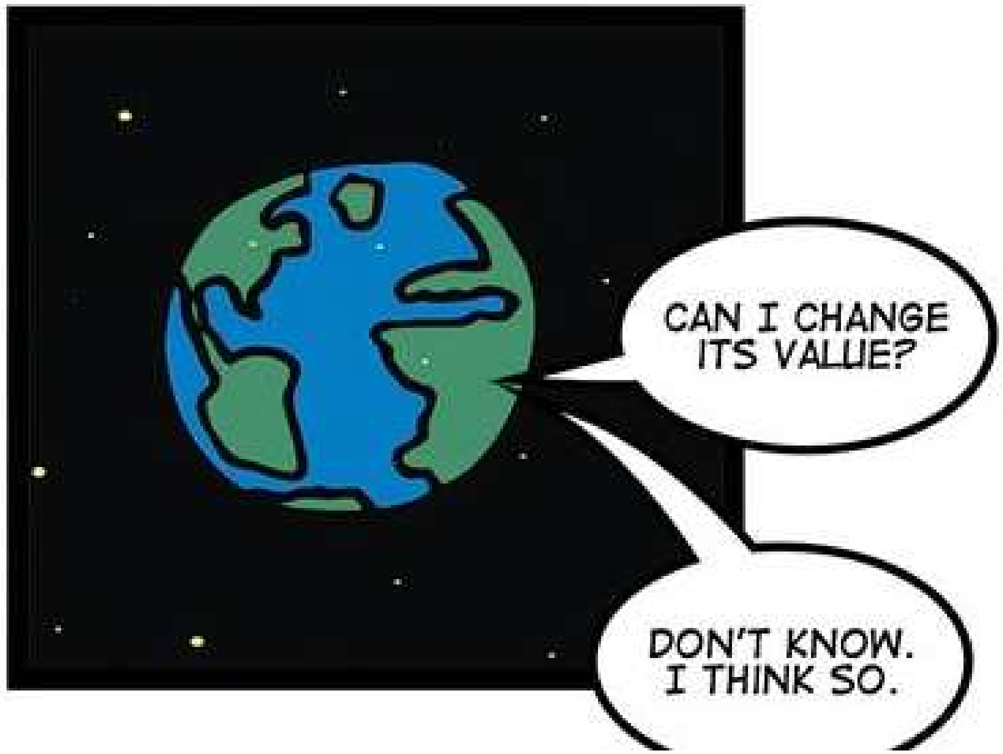
```
void bubbleSort(vector<Student> & students) {
    for (int j = headCount; j > 0; j--)
        for (int i = 0; i < j - 1; i++)
            if (students[i].gender == Gender.male)
                SWAP(students, i, i + 1);
}
```

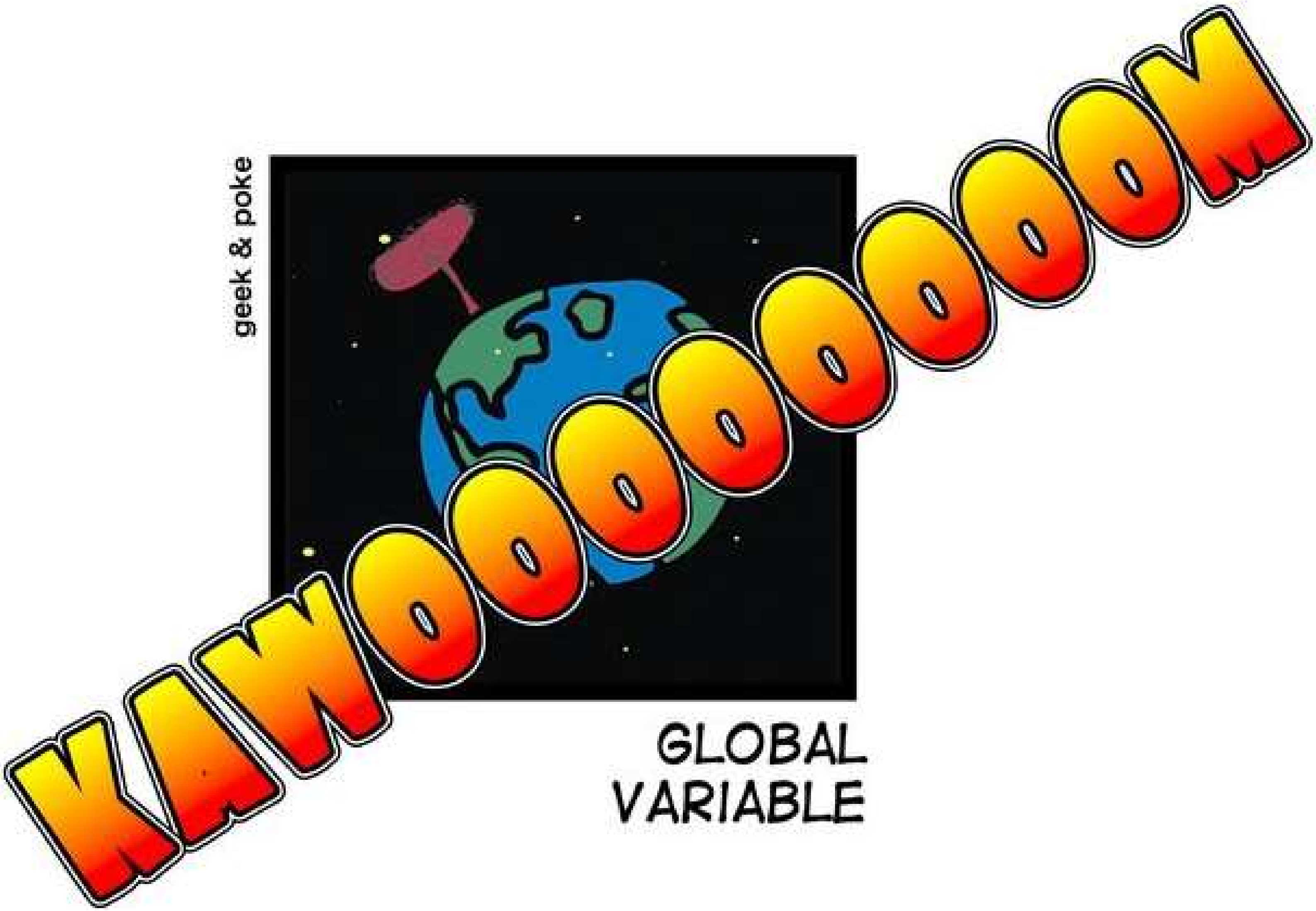
How can we alternate this piece of logic?

```
void bubbleSort(vector<Student> & students) {
    for (int j = headCount; j > 0; j--)
        for (int i = 0; i < j - 1; i++)
            if (students[i].gender == Gender.male)
                SWAP(students, i, i + 1);
}
```

A constant in the logic

How can we alternate this constant in the logic?


CAN I CHANGE ITS VALUE?
DON'T KNOW. I THINK SO.



GLOBAL
VARIABLE

To alternate a constant in a piece of logic

- C
  - Function pointer
  - Demo
- C++ 98
  - Function object
  - Demo
- C++ 11+
  - Lambda
  - Demo

Global variable is in need

Significant design improvement

Private member variable ☺

Syntax sugar

Capture

Capture List

- **[a, &b]** where a is captured by value and b is captured by reference.
- **[this]** captures the this pointer by value
- **[&]** captures all automatic variables mentioned in the body of the lambda by reference
- **[=]** captures all automatic variables mentioned in the body of the lambda by value
- **[]** captures nothing

Practice

Print all even numbers from 0 to 9

Practice

Print all numbers which are times of 3

Avoid default capture modes.

Effective Modern C++ Item 31

Example: schedule a timer

```
void startATimer(Time time){
    Song song = getSongFromSetting();
    timerService.scheduleTimer(time, [&]() { musicPlayer.play(song); });
}
```

Problem:

The lambda will be called long after this startATimer ended.
And song is a local variable in startATimer
What is the value of song when Timer expired and lambda is called?

Example: schedule a timer

```
void startATimer(Time time){
    Song song = getSongFromSetting();
    timerService.scheduleTimer(time, [&song]() { musicPlayer.play(song); });
}
```

Problem:

The lambda will be called long after this startATimer ended.
And song is a local variable in startATimer
What is the value of song when Timer expired and lambda is called?

with an explicit capture, it's easier to see that the viability of the lambda is dependent on divisor's lifetime

Example: schedule a timer

```
void startATimer(Time time){
    Song song = getSongFromSetting();
    timerService.scheduleTimer(time, [=]() { musicPlayer.play(song); });
}
```

Problem:

Problem solved in this case.
Yet, still you don't know what you are capturing

Example: schedule a timer

```
...
public:
    void startATimer(Time time){
        timerService.scheduleTimer(time, [=]() { musicPlayer.play(song); });
    }
private:
    Song song;
...
```

Problem:
Same line of code, different definition of variable.
What this line is capturing?

```
void startATimer(Time time){
    timerService.scheduleTimer(time, [song]() { musicPlayer.play(song); });
}
```

//capture by value, but compile will fail
//capture only apply to non-static local variable

```
void startATimer(Time time){
    timerService.scheduleTimer(time, [=]() { musicPlayer.play(song); });
}
```

Problem:
What this line is capturing?

```
void startATimer(Time time){
    timerService.scheduleTimer(time, [=]() { musicPlayer.play(song); });
}
```

```
void startATimer(Time time){
    timerService.scheduleTimer(time, [this]() { musicPlayer.play(this -> song); });
}
```

Problem:

What this line is capturing?

Item 31: Avoid default capture modes.

- It is to avoid capturing something you don’t know
- It is to encourage to write code you understand

Use init capture to move objects into closures.

Effective Modern C++ Item 32

```
void startATimer(Time time){
    Song songCopy = song;
    timerService.scheduleTimer(time, [songCopy]() { musicPlayer.play(songCopy); });
}
```

Problem:

Problem solved, yet multiple copy construction is called.

Capture initialization

- Available in C++ 14

```
void startATimer(Time time){
    timerService.scheduleTimer(time, [song = song]() { musicPlayer.play(song); });
}
```

Capture initialization

Higher order function

How to make use of lambda

Higher order function

- A function take other functions as argument
  - Bubble Sort

```
void aFunction( ___?___ aLambda){
    ...
    ______________?______________; //calling a lambda
}
```

- What is the type of a lambda?
- How to call a lambda?

Type of lambda

```
#include <functional>
#include <iostream>

int main()
{

    using namespace std;

    // Assign the same lambda expression to a function object.
    function<int(int, int)> f2 = [](int x, int y) { return x + y; };

    cout << f2(3, 4) << endl;
}
```

Calling a lambda

```cpp
#include <iostream>
#include <functional>

int main()
{
    using namespace std;

    // The following code declares a lambda expression that returns
    auto addtwointegers = [](int x) -> function<int(int)> {
        return [=](int y) { return x + y; };
    };

    // The lambda expression applies the argument z to the function f
    // and multiplies by 2.
    auto higherorder = [](const function<int(int)>& f, int z) {
        return f(z) * 2;
    };

    // Call the lambda expression that is bound to higherorder.
    auto answer = higherorder(addtwointegers(7), 8);

    // Print the result, which is (7+8)*2.
    cout << answer << endl;
}
```

Higher order function

- A function take other functions as argument
  - Bubble Sort

```
void aFunction( __?__ aLambda){
    ...
    ________________?________________; //calling a lambda
}
```

function<int(int, int)>

aLambda(1, 2);

- What is the type of a lambda?
- How to call a lambda?

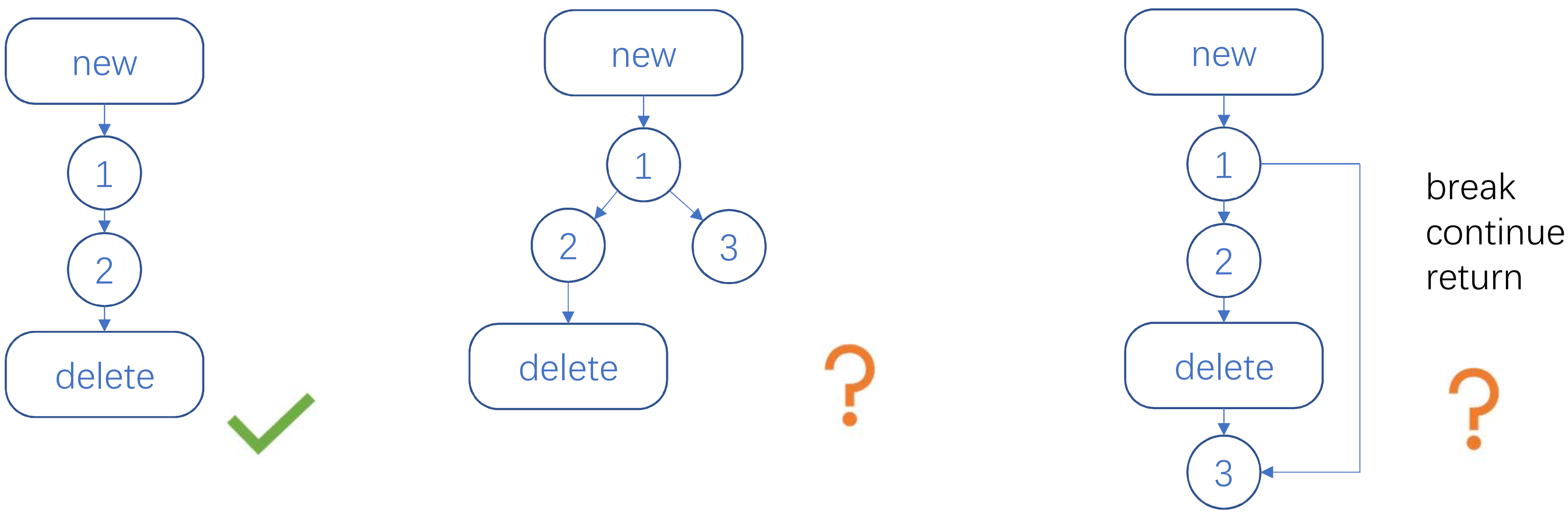
Smart Pointers

Problem

A resource is something that, once you're done using it, you need to return to the system. If you don't, bad things happen.

- allocated heap memory
- thread of execution
- open socket
- open file
- locked mutex
- disk space
- database connection

Problem



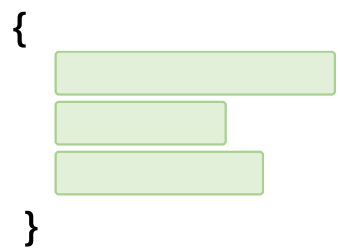
Things get mess when more than one programmer works on the same piece of code, no one knows all the branches, and no one knows all the resources

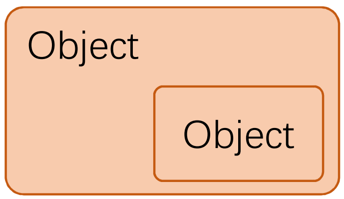
RAII

Resource Acquisition Is Initialization or RAII, is a C++ programming technique which binds the **life cycle** of a resource that must be acquired before use to the lifetime of an object

Object lifetime

- local objects are destructed at the closing brace in reverse order
- sub-objects of an object are destructed when container object is destructed in reverse order





Object pointer

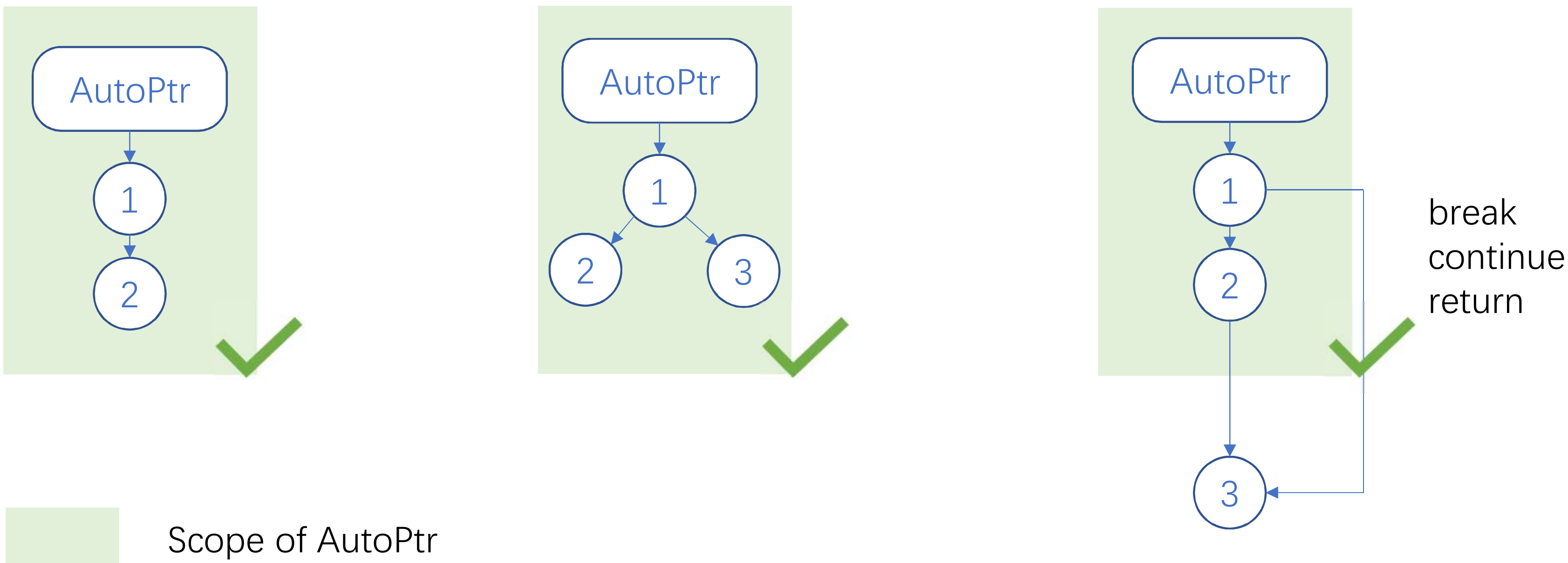
- When it is delete

```
void creatAndDelete{
    AClass * p = createAClass();
    ...
    delete p;
}
```

```
void creatAndDelete{
    AutoPtr p(createAClass());
    ...
}
```

```
class AutoPtr{
public:
    AutoPtr(AClass * ptr) {ptr = ptr;}
    ~AutoPtr() {delete ptr;}
private:
    AClass ptr;
}
```

Problem revisit with AutoPtr



All problems solved as AutoPtr will sure have its scope and will be destructed out of its scope
RAII is also applicable to other resources like mutext…

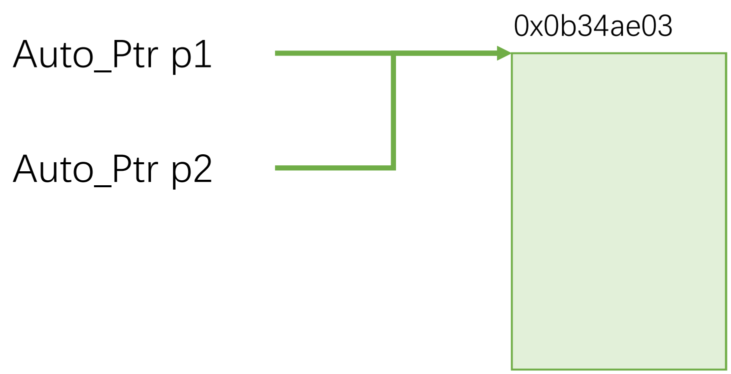
RAII summary

- Resources are acquired and immediately turned over to **resource-managing objects**
- Resource-managing objects use their destructors to ensure that resources are released

Are we there yet?

What should happen when an RAII object is copied?

Copy problem of RAII



0x0b34ae03 will be deleted twice, it will be deleted when p1 destructed, and again when p2 is destructed

Two Possibilities

To get rid of the copy problem of RAII

- **Prohibit copying**
- **Reference-count the underlying resource**

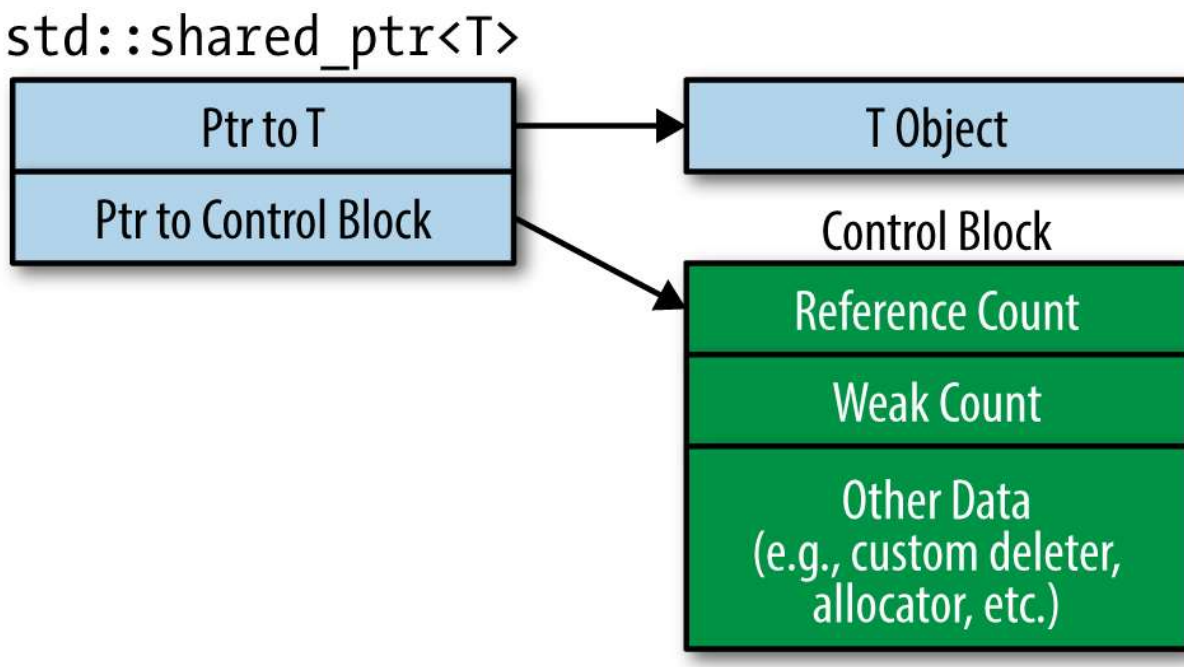
C++ 11 unique_ptr

- Copying a std::unique_ptr **isn't** allowed
- std::unique_ptr is thus a move-only type

embodies exclusive **ownership** semantics. A non-null std::unique_ptr always owns what it points to. Moving a std::unique_ptr transfers ownership from the source pointer to the destination pointer.

Create	std::unique_ptr<Task> taskPtr(new Task(23));
	makeUnique
Move	std::unique_ptr<Task> taskPtr4 = std::move(taskPtr2);

C++ 11 shared_ptr (1/3)



C++ 11 shared_ptr (2/3)

Pros:

std::shared_ptrs offer convenience approaching that of garbage collection for the shared lifetime management of arbitrary resources.

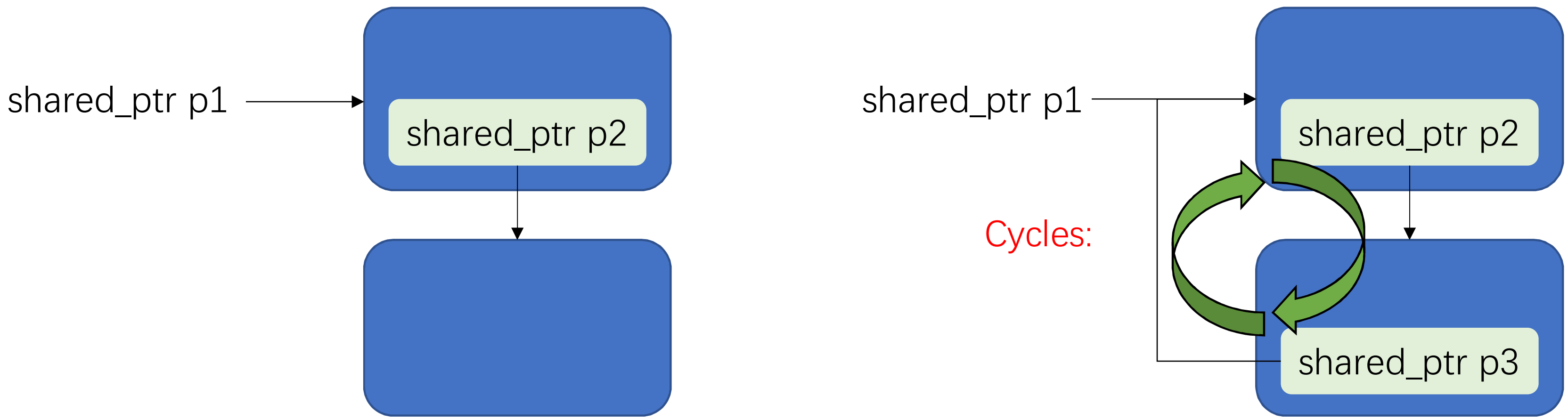
Cons:

Compared to std::unique_ptr, std::shared_ptr objects are typically twice as big, incur overhead for control blocks, and require atomic reference count manipulations.

Avoid creating std::shared_ptrs from variables of raw pointer type.

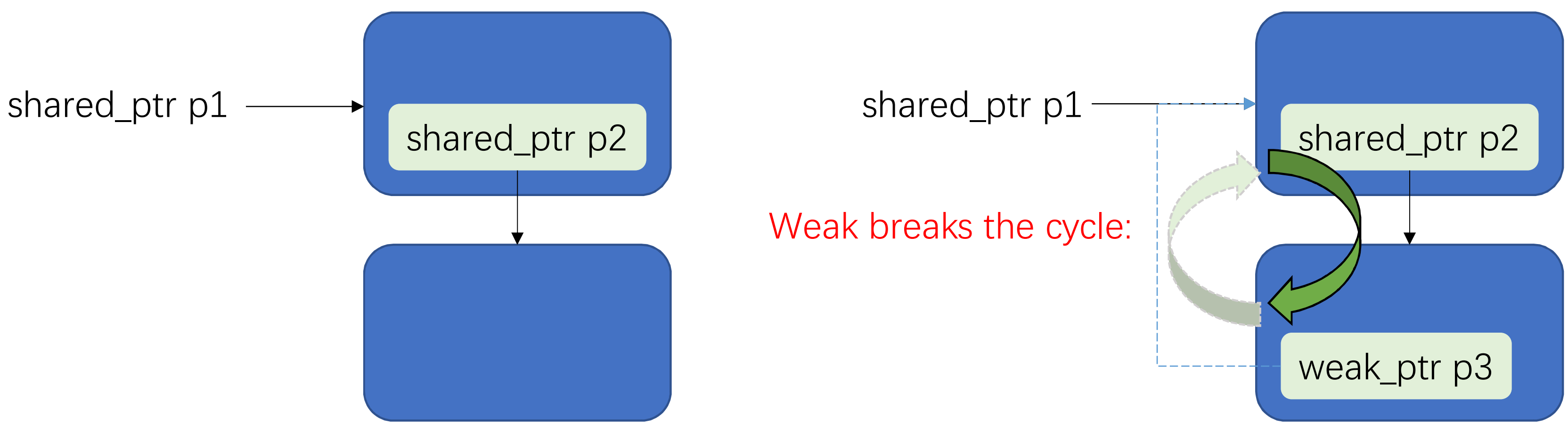
C++ 11 shared_ptr (3/3)

What's more:



C++ 11 weak_ptr

What's more:



C++ smart pointers

- C++11:
  - std::auto_ptr (legacy of C++98)
  - std::unique_ptr
  - std::shared_ptr
  - and std::weak_ptr

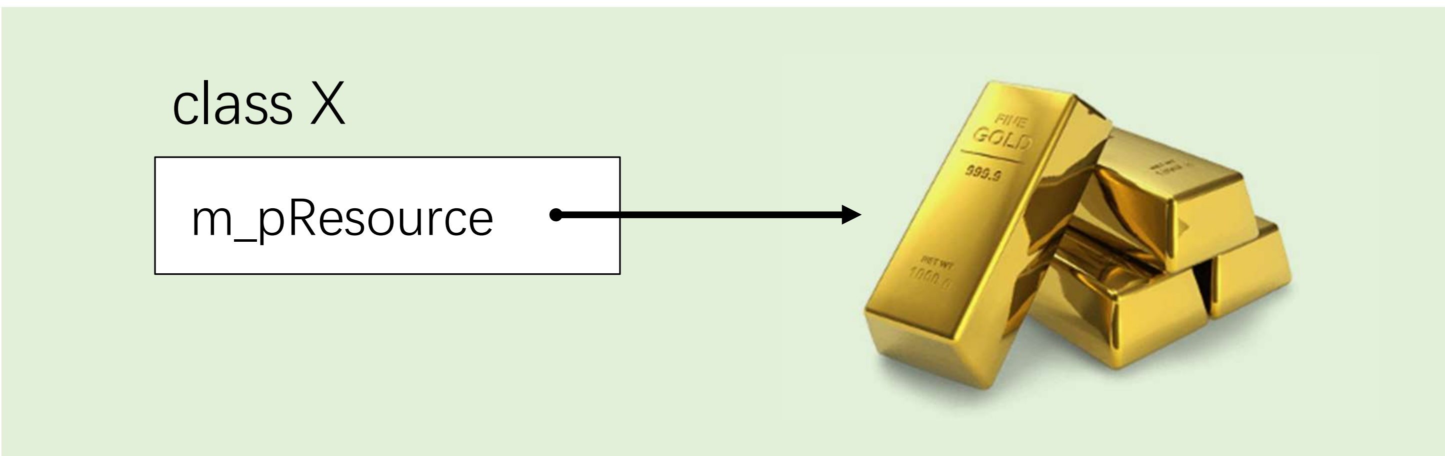
Effective modern C++ items

- **Item 18:** Use std::unique_ptr for exclusive-ownership resource management.
- **Item 19:** Use std::shared_ptr for shared-ownership resource management.
- **Item 20:** Use std::weak_ptr for std::shared_ptr-like pointers that can dangle.

Understand Rvalue Reference

Problem: efficiency of copy

- How to copy an object with expensive resource?



Problem: Efficiency of Copy

- How to copy an object with expensive resource?

```
X& X::operator=(X const & rhs)
{
  // [...]
  // Make a clone of what rhs.m_pResource refers to.
  // Destruct the resource that m_pResource refers to.
  // Attach the clone to m_pResource.
  // [...]
}
```

Problem: Efficiency of Copy

```
X a;
X b;
b = a;
```

Make a clone of resource that a holds

Destruct the resource that b holds

Let b to holds the resource just cloned from a

Leads to 2 copies of resource
Which is necessary, as a and b may have chance to be accessed separately

```
X foo();
X b;
b = foo();
```

Make a clone of resource that temp object holds

Destruct the resource that b holds

Let b to holds the resource just cloned from temp object

Leads to 2 copies of resource
Which makes no sense, as there is no chance to access to the temp object after the assign operation

Problem: Efficiency of Copy

```
X foo();
X b;
b = foo();
```

An alternative way of copy

Move Semantics

Make a clone of resource that temp object holds

Destruct the resource that b holds

Let b to holds the resource just cloned from temp object

Swap the resource holds by b and temp object

Leads to 2 copies of resource

Which makes no sense, as there is no chance to access to the temp object after the assign operation

One copy of resource

Resource holds by b is automatically destructed when the temp object is to destruct

Problem: Efficiency of Copy

How can we tell the differences of these 2 case?

```
X a;
X b;
b = a;
```

Make a clone of resource that a holds

Destruct the resource that b holds

Let b to holds the resource just cloned from a

Leads to 2 copies of resource
Which is necessary, as a and b may have chance to be accessed separately

```
X foo();
X b;
b = foo();
```

Swap the resource holds by b and temp object

One copy of resource
Resource holds by b is automatically destructed when the temp object is to destruct

R-value and L-value in C

An *lvalue* is an expression e that may appear on the left or on the right hand side of an assignment

An *rvalue* is an expression that can only appear on the right hand side of an assignment

```
int a = 42;
int b = 43;

// a and b are both l-values;
a = b; // ok
b = a; // ok
a = a * b; // ok
```

```
// a * b is an r-value
int c = a * b; // ok, rvalue on right hand side of assignment
a * b = 42; // error, rvalue on left hand side of assignment
```

R-value and L-value in C++

An *lvalue* is an expression that refers to a memory location and allows us to take the address of that memory location via the & operator

Rvalue is an expression that is not an lvalue

```
// l-values;

int I = 42;
i = 43; // ok, I is a l-value
int* p = &i; //ok, I is a l-value

int& foo();
foo() = 42; //ok, foo() is an l-value
int* p1 = &foo(); // ok, foo() is an l-value
```

```
// r-values;

int foobar();
int j = 0;
j = foobar(); // ok, foobar() is an rvalue

int* p2 = &foobar(); // error, cannot take the
                     //addresss of a r-value
j = 42; // ok, 42 is an rvalue
```

Overload assignment operation for r-value

How can we tell the differences of these 2 case?

```
X a;
X b;
b = a;
```

```
X foo();
X b;
b = foo();
```

```
X& X::operator=(X & a){
    Make a clone of resource that a holds
    Destruct the resource that b holds
    Let b to holds the resource just cloned from a
}
```

```
X& X::operator=(????? a){
  Swap the resource holds by b and temp object
}
```

Overload assignment operation for r-value

R-value reference: &&

```
X a;
X b;
b = a;
```

```
X foo();
X b;
b = foo();
```

```
X& X::operator=(X & a){
    Make a clone of resource that a holds
    Destruct the resource that b holds
    Let b to holds the resource just cloned from a
}
```

```
X& X::operator=(X && a){
  Swap the resource holds by b and temp object
}
```

Overload function with R-value reference

- Review overload function with R-value

```
void foo(X& x);
void foo(X&& x);


X x;
foo(x); // x is lvalue, calls foo(X& x)


X foobar();
foo(foobar()); // x is rvalue, calls foo(X&& x)
```

It has a name

```
X& X::operator=(X const & rhs);
X& X::operator=(X&& rhs);

void foo(X&& x){
    X another = x;
};
```

Which version of assignment operation gets called

It has a name

```
X& X::operator=(X const & rhs);
X& X::operator=(X&& rhs);

void foo(X&& x){
    X another = x;
};
```

It is Lvalue version of assign operator gets called

Things that are declared as rvalue reference can be lvalues or rvalues. The distinguishing criterion is: *if it has a name*, then it is an lvalue. Otherwise, it is an rvalue.

std::move(x)

std::move turn l-value to r-value

Practice: turning to the right

Turning to the right

```
Base(Base const & rhs);
Base(Base&& rhs);

Derived(Derived const & rhs):Base(rhs){
    // Derived-specific stuff
}

Derived(Derived const & rhs):Base(rhs){
    // Derived-specific stuff
}
```

How is this implementation?
Is it correct?
If it is not, what is the problem?
How to fix it

Turning to the right

```
Base(Base const & rhs);
Base(Base&& rhs);

Derived(Derived const & rhs):Base(rhs){
    // Derived-specific stuff
}

Derived(Derived const & rhs):Base(std::move(rhs)){
    // Derived-specific stuff
}
```

Force move semantics

Implement

```
void foo(X&);
```

Not Implement

```
void foo(X&&);
```

foo only accept L-value but not R-value

Force move semantics

Implement

```
void foo(X const &);
```

Not Implement

```
void foo(X&&);
```

Foo accept both L-value and R-value
But there is no distinguish between L-value and R-value

Force move semantics

Implement

```
void foo(X&&);
```

Not Implement

```
void foo(X&);
void foo(X const &);
```

Foo only accept R-value, which forces move semantics

Argument forward: The problem

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg arg)
{
    return shared_ptr<T>(new T(arg));
}
```

A wrapper to pass args to T’s member function, in our case, T’s constructor

Extra copy

it introduces an extra call by value, which is particularly bad if the constructor takes its argument by reference

Argument forward: The problem

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg& arg)
{
    return shared_ptr<T>(new T(arg));
}
```

Pass by reference
More efficient than previous version

Won't take r-value

```
factory<X> (hoo()); // error if hoo returns by value
factory<X>(41); // error
```

Argument forward: The problem

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg const & arg)
{
    return shared_ptr<T>(new T(arg));
}
```

Takes both r-value and l-value for factory

Restrict T’s function input

```
X::X (Arg const & arg);
factory<X> (arg); // OK, X's constructor takes const args

Y::Y (Arg & arg);
factory<Y> (arg); // error, Y's constructor may change args
```

Argument forward: The problem

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg const & arg)
{
    return shared_ptr<T>(new T(arg));
}

template<typename T, typename Arg>
shared_ptr<T> factory(Arg & arg)
{
    return shared_ptr<T>(new T(arg));
}
```

Overload, take both const and non const argument

Duplication

Even worse, when T has more than one args, it introduce exponential complexity

Argument forward: Solution

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg&& arg)
{
    return shared_ptr<T>(new T(std::forward<Arg>(arg)));
}
```

Universal references

```
TR    R
T&    &     ->   T&     // lvalue reference to cv TR -> lvalue reference to T
T&    &&    ->   T&     // rvalue reference to cv TR -> TR (lvalue reference to T)
T&&   &     ->   T&     // lvalue reference to cv TR -> lvalue reference to T
T&&   &&    ->   T&&    // rvalue reference to cv TR -> TR (rvalue reference to T)
```

If a function template parameter has type T&& for a deduced type T, or if an object is declared using auto&&, the parameter or object is a universal reference.

Universal references correspond to rvalue references if they're initialized with rvalues. They correspond to lvalue references if they're initialized with lvalues

Argument forward: Solution

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg&& arg)
{
    return shared_ptr<T>(new T(std::forward<Arg>(arg)));
}
```

std::forward

std::forward allows rvalue arguments to be passed on as rvalues, and lvalues to be passed on as lvalues, a scheme called “perfect forwarding.”

Item 23: Understand std::move and std::forward.

- std::move performs an unconditional cast to an rvalue. In and of itself, it doesn't move anything.
- std::forward casts its argument to an rvalue only if that argument is bound to an rvalue.
- Neither std::move nor std::forward do anything at runtime.

Item 24: Distinguish universal references from rvalue references.

- If a function template parameter has type T&& for a deduced type T, or if an object is declared using auto&&, the parameter or object is a universal reference.
- If the form of the type declaration isn't precisely type&&, or if type deduction does not occur, type&& denotes an rvalue reference.
- Universal references correspond to rvalue references if they're initialized with rvalues. They correspond to lvalue references if they're initialized with lvalues.

Item 28: Understand reference collapsing.

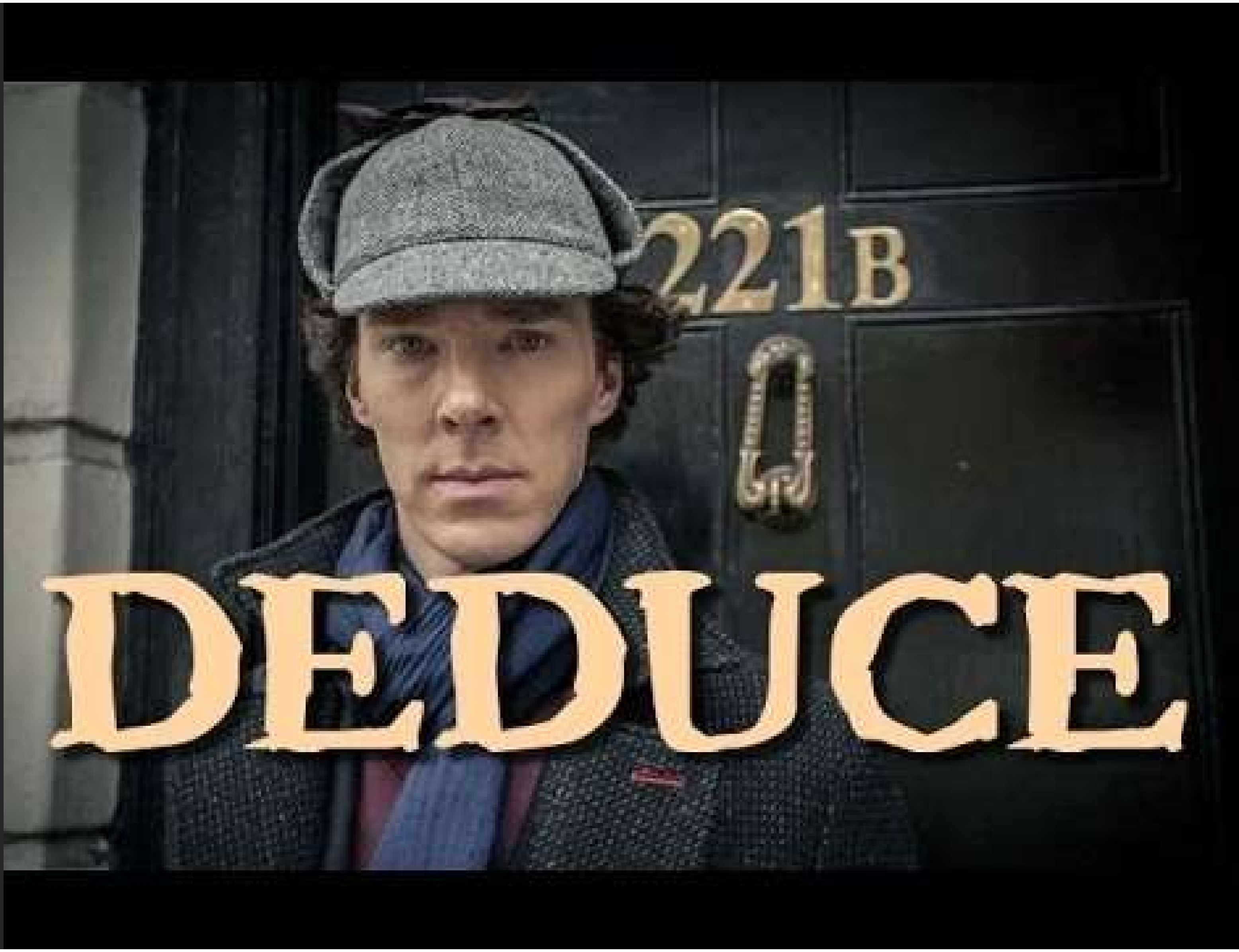
```
TR    R
T&    &    ->  T&    // lvalue reference to cv TR -> lvalue reference to T
T&    &&   ->  T&    // rvalue reference to cv TR -> TR (lvalue reference to T)
T&&   &    ->  T&    // lvalue reference to cv TR -> lvalue reference to T
T&&   &&   ->  T&&   // rvalue reference to cv TR -> TR (rvalue reference to T)
```

Item 29: Assume that move operations are not present, not cheap, and not used.

In generic programming

- Assume that move operations are not present, not cheap, and not used.
- In code with known types or support for move semantics, there is no need for assumptions.

Deducing types



auto: It’s very convenient

```
int str = “Hello world”;
error: invalid conversion from ‘const char*’ to ‘int’
auto str = “Hello world”;
```

```
vector<MySpecificStruct> v;
vector<MySpecificStruct>::iterator it = v.begin();
auto it = v.begin();
```

auto is to deduce a type from a expression

auto is more than convenient

```
template<typename T, typename S>
void foo(T lhs, S rhs){
    _?_ prod = lhs * rhs;
    ...
}
```

What is the type of prod?
It depends on T and S
As a programmer we don't know
But Compiler knows
So we put auto instead a specific type name

auto target = b;

- Reference part and const are ignored

```
int a = 1;
const int & b = a
auto target = b; // target is an int not const int &
```

being a reference is not so much a type characteristic as it is a behavioral characteristic of a variable. The fact that the expression from which I initialize a new variable behaves like a reference does not imply that I want my new variable to behave like a reference as well.

const auto& target = b;

```
int a = 1;
int & b = a
auto& target = b; // target is int &
const auto & target_2 = b; // target_2 is const int &
```

auto can also be decorated by & and const

auto&& target = b;

```
int a = 1;
auto&& target = a; // target is int &
auto&& target_2 = 42; // target_2 is int &&
```

universal reference

forwarding reference

auto	T&	->	T	
auto&	T&	->	T&	Deducing lvalue reference
auto&	T&&	->	T&	Deducing rvalue
auto&&	T&	->	T&	Deducing lvalue
auto&&	T&&	->	T&&	

decltype

```
template<typename T, typename S>
void foo(T lhs, S rhs){
    auto prod = lhs * rhs;
    ...
}
```

```
template<typename T, typename S>
void foo(T lhs, S rhs){
    decltyp( lhs * rhs ) prod = lhs * rhs;
    ...
}
```

decltype is to deduce a type from a expression in a different syntax
And more than that

decltype vs. typeof

Before C++ 11

Not standard

```
template<typename T, typename S>
void foo(T lhs, S rhs){
    typedef typeof( lhs * rhs ) produdct_type;
    ...
}
```

C++ 11

standard

```
template<typename T, typename S>
void foo(T lhs, S rhs){
    typedef decltyp( lhs * rhs ) produdct_type;
    ...
}
```

decltype: deducing the function return type

```
template<typename T, typename S>        //C++ 14
auto multiply(T lhs, S rhs){
    return lhs * rhs;
}
```

```
template<typename T, typename S>        //doesn't compile lhs and rhs are not in scope when
decltype(lhs * rhs) multiply(T lhs, S rhs){        //preceding the function name
    return lhs * rhs;
}
```

```
template<typename T, typename S>        //OK
auto multiply -> decltype(lhs * rhs) (T lhs, S rhs){
    return lhs * rhs;
}
```

New in C++ 14

- Function return type deduction

```
auto add(int a, int b){
 return a + b;
}
```

- Type deduction for lambda arguments

```
[](auto x, auto y) { return x + y;}
```

Item 2: Understand **auto** type deduction

- arguments that are references are treated as non-references
- When deducing types for universal reference parameters, lvalue arguments get special treatment.
- When deducing types for by-value parameters, const and/or volatile arguments are treated as non-const and non-volatile.

Item 5: Prefer **auto** to explicit type declarations

auto variables must be initialized, are generally immune to type mismatches that can lead to portability or efficiency problems, can ease the process of refactoring, and typically require less typing than variables with explicitly specified types.

to modern C++

Item 8: Prefer **nullptr** to 0 and NULL

- Prefer nullptr to 0 and NULL.
- Avoid overloading on integral and pointer types.

```
void f(int);
void f(void*);

f(0);
f(NULL); // which function does it call
f(nullptr);
```

Keyword: delete

```
class basic_ios {
public:
    basic_ios() = delete;
    basic_ios(const basic_ios&) = delete;
    basic_ios(& operator = (const basic_ios&) = delete;
}
```

Example one

- Refactoring, dependence injection

```
Class A{
Public:
    A(){
        m_device = new CertainDevice();
    }
Private:
    CommonDevice * m_device;
}
```

```
Class A{
Public:
    A(CommonDevice * device) : m_device(device){}
Private:
    CommonDevice * m_device;
}
```

Is there any problem of this refactoring?

Example Two

- Enqueue

```
Class Msg{
...
Private:
    Payload * payload;
}

messageQueue.push_back(incomingMsg)
```

Is there any problem of this piece of code?

C++ 98

```
class basic_ios {
private:
    basic_ios();
    basic_ios(const basic_ios&);
    basic_ios(& operator = (const basic_ios&);
}
```

And pay attention, it will need to be deliberately failing to define them, to avoid those member functions or friends to get access to them

delete non-member function

```
bool isGoodNumber(int num);

isGoodNumber('a');  //OK
isGoodNumber(true);  //OK

Bool isGoodNumber(char) = delete;
Bool isGoodNumber(bool) = delete;

isGoodNumber('a'); //Error
isGoodNumber(true); //Error
```

Item 11: Prefer **delete** to private undefined ones

- Prefer deleted functions to private undefined ones.
- Any function may be deleted, including non-member functions and template instantiations..

Scoped enums

C++ 98 enum or unscoped enum

```
enum Color {black, white, red};

Color c = white; // OK
auto white = true; // error
```

Scoped enum since C++ 11

```
enum class Color {black, white, red};

Color c = white; // error

Color c = Color::white; //OK
auto white = true; // OK
```

typed enum

- Scoped enum has default underlying type of int
- Scoped enum can thus be forward declared

- Unscoped enum has no default underlying type
- Unscoped enum can not be forward declared unless put a underlying type to it, which is recommended

Item 10: Prefer scoped enums to unscoped enums

- C++98-style enums are now known as unscoped enums.
- Enumerators of scoped enums are visible only within the enum. They convert to other types only with a cast.
- Both scoped and unscoped enums support specification of the underlying type. The default underlying type for scoped enums is int. Unscoped enums have no default underlying type.
- Scoped enums may always be forward-declared. Unscoped enums may be forward-declared only if their declaration specifies an underlying type.

Override

```
class Base{
Public:
    virtual void doWork();
    ...
}

class Derived: public Base{
Public:
    virtual void doWork();
    ...
}

Base * ptr = new Derived;
ptr->doWord();
```

Which function does it call?

Polymorphism -- Override

```
class Base{
Public:
    virtual void doWork();
    ...
}

class Derived: public Base{
Public:
    virtual void doWork();
    ...
}

Base * ptr = new Derived;
ptr->doWord();
```

- The base class function must be virtual.
- The base and derived function names must be identical
- The parameter types of the base and derived functions must be identical.
- The return types and exception specifications of the base and derived functions must be compatible.
- The functions' reference qualifiers must be identical.

Which function does it call?

Function reference qualifier

```
class A{
public:
    ...
    void doWork() &;
    void doWork() &&;
};
```

This version of doWork applies to only when *this is an lvalue

This version of doWork applies to only when *this is an rvalue

```
A a;
A makeA();
```

```
a.doWork();
makeA().doWork();
```

Calls doWork for lvalue

Calls doWork for rvalue

override keywords

```
class Base{
Public:
    virtual void doWork();
    ...
}

class Derived: public Base{
Public:
    virtual void doWork() override;
    ...
}
```

Item 12: Declare overriding functions override

- Declare overriding functions override.
- Member function reference qualifiers make it possible to treat lvalue and rvalue objects (*this) differently.


SUMMARY

- Lambda Expressions in C++
- Smart Pointer
- R-value Reference
- Deducing types
- Moving to modern C++

Item 31: Avoid default capture modes
Item 32: Use init capture to move objects into closures
Item 18: Use std::unique_ptr for exclusive-ownership resource management.
Item 19: Use std::shared_ptr for shared-ownership resource management.
Item 20: Use std::weak_ptr for std::shared_ptr-like pointers that can dangle.
Item 23: Understand std::move and std::forward.
Item 24: Distinguish universal references from rvalue references.
Item 28: Understand reference collapsing.
Item 29: Assume that move operations are not present, not cheap, and not used.
Item 2: Understand auto type deduction
Item 5: Prefer auto to explicit type declarations
Item 8: Prefer nullptr to 0 and NULL
Item 11: Prefer delete to private undefined ones
Item 10: Prefer scoped enums to unscoped enums
Item 12: Declare overriding functions override