Copy elision

Yossi Moalem

Value Categories

- Not something we encounter on our day-day life
  - Even as professional programmers
- We may see them in compilation errors
  - *lvalue required as left operand of assignment*
  - *invalid initialization of non-const reference of type ‘foo&' from an rvalue of type ‘foo‘*

Check Point
SOFTWARE TECHNOLOGIES LTD

Value Category:

- Categories of expressions, not values
- Introduces in CPL, adopted in C, and then in C++
- Refined in C++11
- And again, in C++17

Lvalues and Rvalues

- Originally, Lvalue was on the left side on assignment, while Rvalue on the right
  - Lvalue = Rvalue

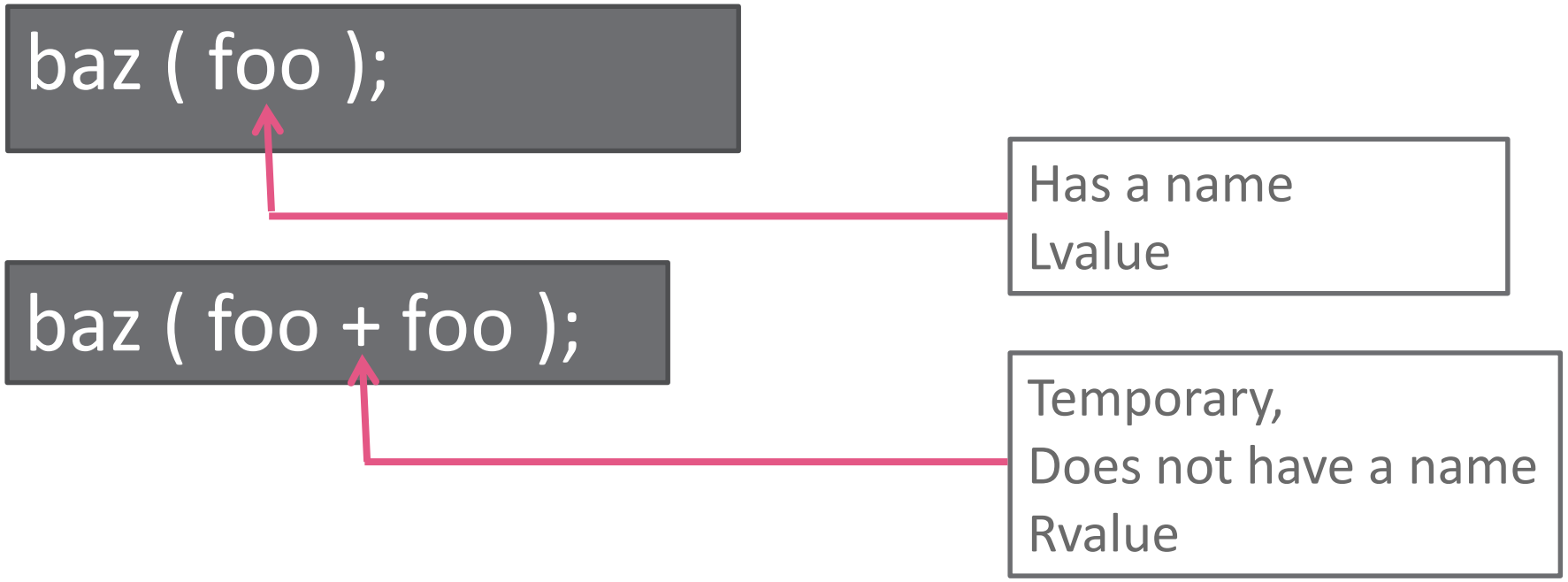
- This is inaccurate:
  - Lvalue may appear on both side
  - Certain Lvalues may not appear on left side

Check Point
SOFTWARE TECHNOLOGIES LTD

Lvalues and Rvalues, C++ 03

Classify based on identity

- Lvalue has identity
- Rvalue, does not have identity



Check Point
SOFTWARE TECHNOLOGIES LTD

What are the value categories

```
Foo(3)


int i = 1;
Foo( i );
foo ( &i );
```

```
struct Foo {
      int baz() {
            return this->_baz;
 }

struct Foo {
      int& baz() {
            return this->_baz;
 }
```

Check Point
SOFTWARE TECHNOLOGIES LTD

BTW, lets have a look at the last example

```
struct Foo {
    ....
    int& baz() {
        return this->_baz;
    }
```

- We return Lvalue, this means we can write:

  Foo.baz() = 3;

  – Yes, we all write such code. When???

Remind me again

Check Point
SOFTWARE TECHNOLOGIES LTD

Pre C++ 11

- Understand remote language features
- Read the standard
- Understand compiler errors
- Impress all your friend
- Be the center of every party

Check Point
SOFTWARE TECHNOLOGIES LTD

Starting C++ 11

- This is simply super important
- Base of key features added in C++ 11

So, lets start with C++ 03!

Is this legal

```
std::string foo() {
        std::string foo {“Foo”};
        return foo;
}

void bar() {
        const std::string & strRef = foo();
}
```

Yes it is legal!

Binding **const reference** on the stack to **temporary** – lengthens the temporary lifetime to that of the reference

Check Point
SOFTWARE TECHNOLOGIES LTD

Is this still legal??

```
std::string foo() {
        std::string Foo {“Foo”};
        return foo;
}

void bar() {
        std::string & strRef = foo();
}
```

Not any more

- invalid initialization of non-const reference of type '…' from an rvalue of type '…'

- And now we know what exactly this this message means…
  - Note: some compilers may allow this. The standard does not **forbid** this.

But why??

```
void foo (double & d){  d++; }


int bar (){
        int intValue = 3;
        //foo(intValue);
}
```

foo would have been called with temporary

intValue would not have been incremented.

Check Point
SOFTWARE TECHNOLOGIES LTD

Binding to Member

```
struct Answer {
Answer ( const string & value) : _value(value)
   {}
const string & _value;
};

void h2g2 () {
Answer answer (“forty two”);
cout <<“The answer is “ << answer._value;
}
```

Binding to Member, Cont'd

- Lifetime extension only takes place when binding to const reference on the stack
- Note: there is no warning here!

- To make things more interesting, this is correct:

*cout <<"The answer is " <<Answer(string("forty two")).*_value;*

Binding to Member #2

```
struct Socket{
    Socket(){cout <<“Opening Socket\n";}
   ~ Socket() { cout<<“Closing Socket \n"; }
};

struct Bar {
    const  Socket& _socket;
    Bar(const  Socket & socket) :
                    _socket(socket)
          { cout << "Holding Socket\n"; }
    ~Bar() { cout << “Releasing Socket" ;    }
};
```

```
int main(){
    Bar b(Socket{});
    cout << "Main finishes\n";
}
```

Binding to Member, Cont'd

- This will output:

  Creating Socket

  Holding Socket

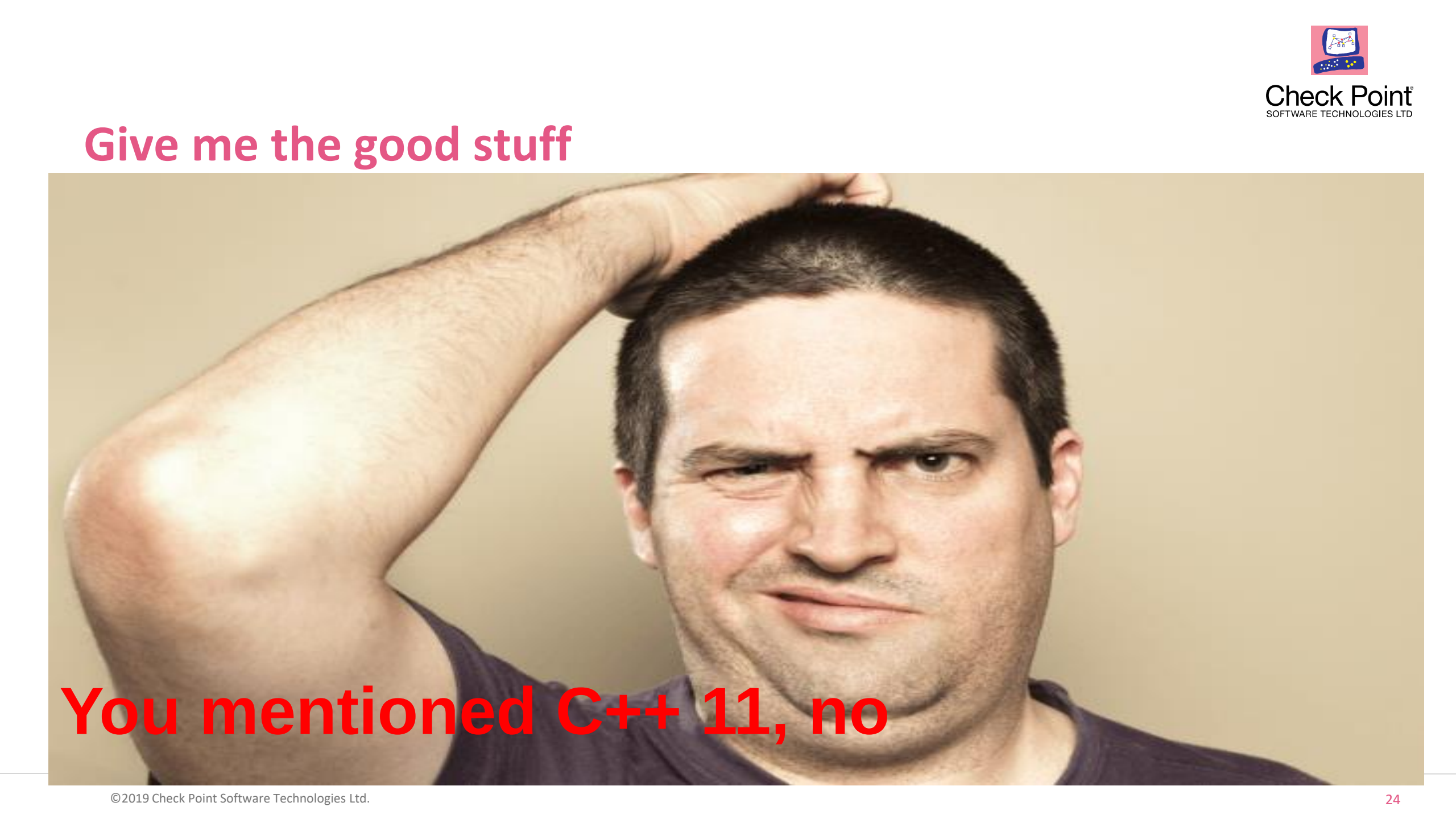
  Closing socket <- Bar is now holding a dangling reference

  Main finishes

  Releasing Socket

Which destructor is called?

```
struct Base{
	~Base() {  cout << "Base";     } // no virtual
};

struct Derived:public Base {
	~Derived()  { cout << "Derived";   }
};

Derived f(){   return Derived(); };

int main(){
	const Base& b = f();
}
```



Give me the good stuff

C++ 11 : Move

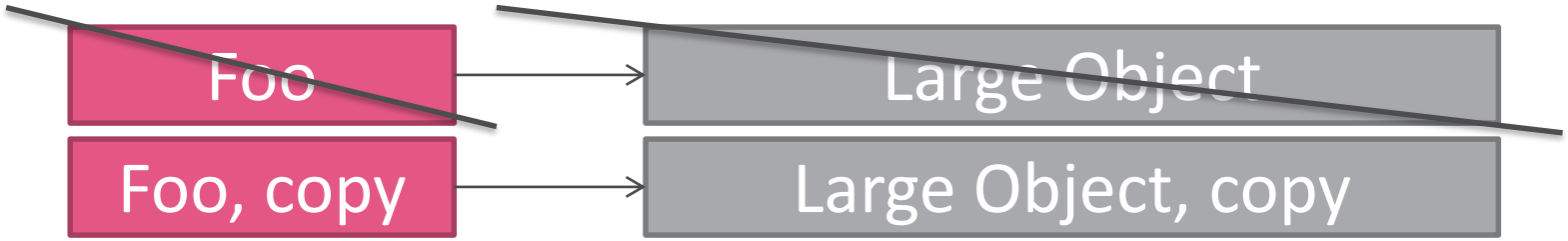
```
{
    Foo foo;
    container.push_back(foo);
    container.push_back(foo + foo);
    container.push_back(foo);
}
```

When we no longer need the object we want to move it, not to copy

What does “moving it” means

Copy:

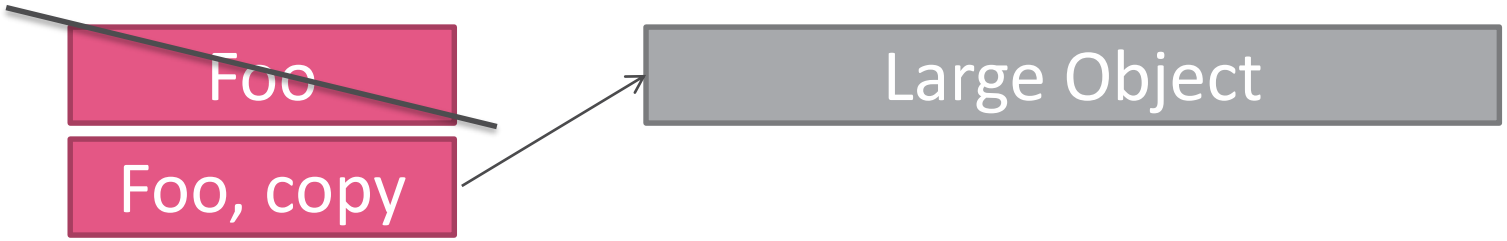
Foo goes out or scope



What does “moving it” means

Move:

Foo goes out or scope



Check Point
SOFTWARE TECHNOLOGIES LTD

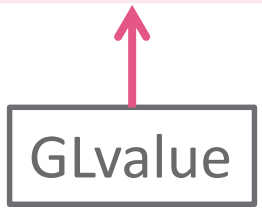
State after move

- Valid
- Maybe (and likely is) inconsistent
- Should be assign-able
- Must not be used
- Must not touch the large object (destruction)

Object category, C++11

- C++ 03 classed expressions based on identity
- C++ 11 added movability to the classification

	Has Identity	Does not have identity
Can be moved from	Xvalue	PRvalue
Cannot be moved from	Lvalue	Not in C++



Check Point
SOFTWARE TECHNOLOGIES LTD

GLvalue – Generalized Lvalue

Any expression that has an identity

- Lvalue:
  - Cannot be moved from
  - Can take address of
  - Original Lvalues could not be moved from

- Xvalue (eXpiring value)
  - Can be moved from
  - Normally, object is going to be expired soon
  - std::move casts into Xvalue

PRvalue – Pure Rvalue

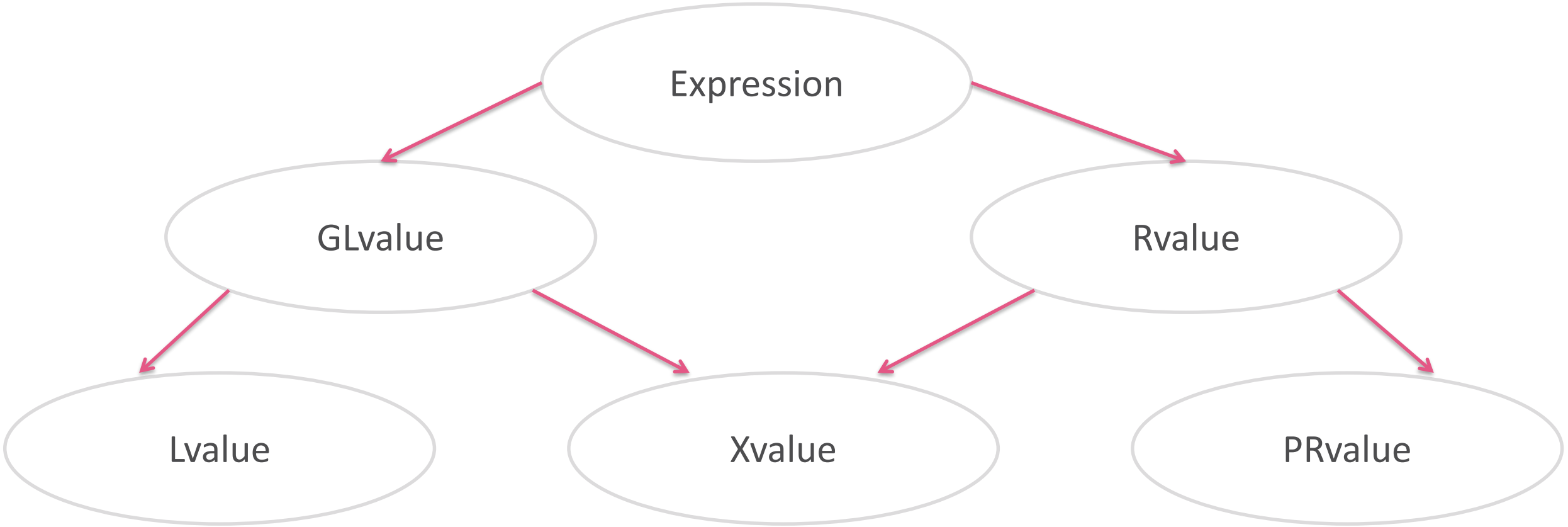
- What Rvalue used to be
- Expression without identity
- Normally used to initialize objects
- For example:
  - Function call that returns non-reference (void/by value)
  - Temporaries
  - Literals (1, true, ‘x’, etc.)

Rvalues

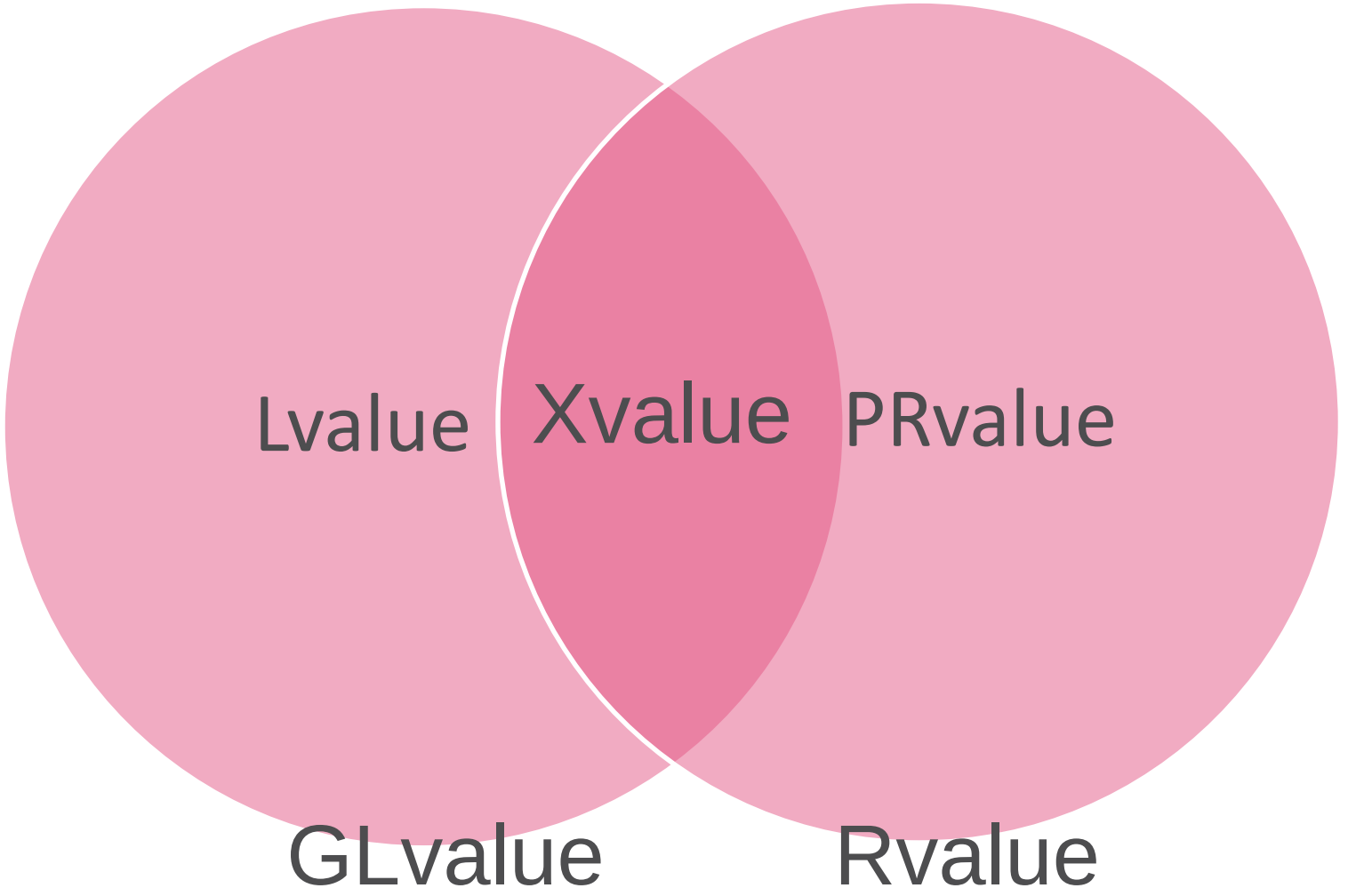
- Generalization of the original Rvalue
- PRvalue or Xvalue
- May, or may not have identity
- Can be moved from

Check Point
SOFTWARE TECHNOLOGIES LTD

The Whole Picture



And, another way to look at it:



Lets practice

```
4
```
PRvalue

```
“Bar”
```
Lvalue

```
Foo foo;
foo;
```
Lvalue

```
std::move(foo);
```
Xvalue

```
struct Foo {
   void bar() {
         this;
    }
};
```
Xvalue

```
int func() { ... }
func();
```
PRvalue

```
int & func() { … }
func();
```
Lvalue

```
int && func() { … }
foo();
```
Xvalue

```
struct Foo {
   int i
} foo;
 foo.i
```
Lvalue

```
&foo.i
```
PRvalue

Back to move

- Move is allowed only if argument is Rvalue
- C++ 11 introduces the ability to receive Rvalue as argument
  - Part of overload resolution

void push(Bar && b) //Will receive Rvalues only. Can move

void push(const Bar & b) //can receive Lvalues and Rvalues. Has to copy

A word on overload resolution

- No && : C++ 03 rules
  - & : can be called with Lvalue only
  - const& : can be called with Lvalues and Rvalues

- Only &&: Move only
  - Only Rvalues can be called
  - Used in unique pointers, string stream etc.

- && and & or const & : distinguish between Lvalue and Rvalue
  - Rvalue version can(and should) move

Another look

```
void useFoo (Foo & );
void useFoo (Foo && );

Foo && getFoo() { …. }

Foo && foo = getFoo();
useFoo (foo);
```

Check Point
SOFTWARE TECHNOLOGIES LTD

Look closer:

void foo (Bar arg)

arg has name -> lvalue

Of type Bar

Check Point
SOFTWARE TECHNOLOGIES LTD

Look closer:

void foo (Bar & arg)

arg has name -> lvalue

Of type Lvalue reference to Bar

Check Point
SOFTWARE TECHNOLOGIES LTD

Look closer:

void foo (Bar && arg)

arg has name -> Lvalue

Of type Rvalue reference to Bar

Back to the example:

```
void useFoo (Foo & foo);
void useFoo (Foo && foo);

Foo && getFoo() { …. }
Foo && foo = getFoo();
useFoo (foo);
```

Check Point
SOFTWARE TECHNOLOGIES LTD

Back to the example:

```
void useFoo (Foo & foo);
void useFoo (Foo && foo);

Foo && getFoo() { …. }
Foo && foo = getFoo();
useFoo (std::move(foo));
```

And a dark corner of C++: Reference Qualifier

```
void F00::doBar() & ; //only LValue "this" can bind

void F00::doBar() && ; //only RValue "this" can bind
```

Hardly the most usable feature added to C++11…

Copy Elision

Most compiler optimizations – as-if

Copy elision:

- Allowed to elide copy, if original is not going to be used
- Even if copy has side effects
- Allowed to change the observable state of the program

Return Value Optimization

- Eliminate creation of temporary returned by function

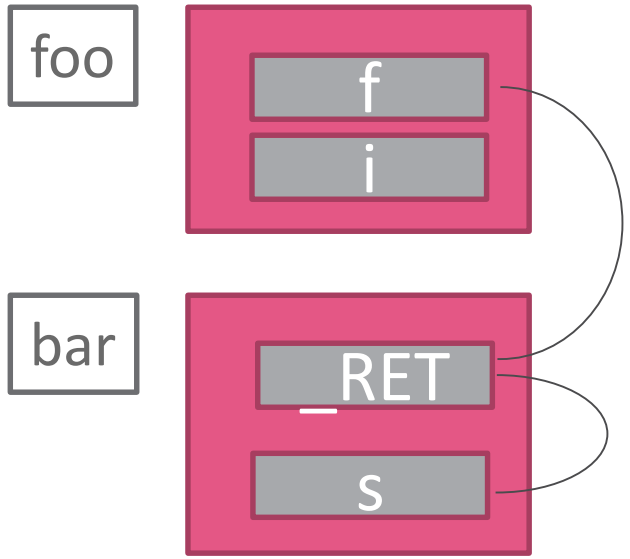
- Copy Ctor must still be accessible
  - But may not be defined.
- Return type and the target must be of the same type
- Return value must be local variable
  - Arguments do not qualify

RVO, Example

```
struct Foo { … };

Foo createFoo() {
  Foo f;
  return f;    //First copy should be here
}

int main() {
  Foo f = createFoo(); //Second copy should be here
  return 0;
}
```

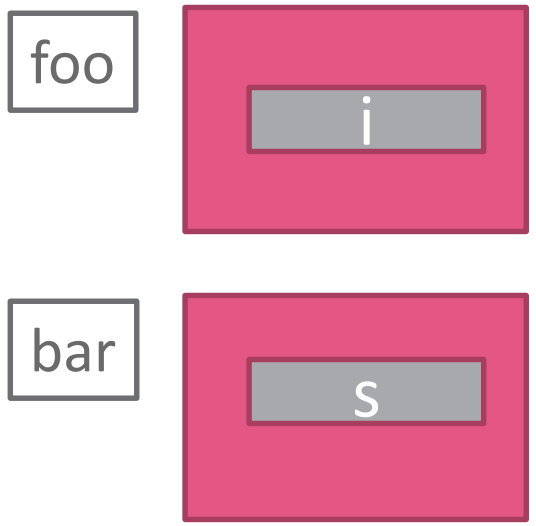
Without RVO

```
string foo () {
    string  f{“lala”};
    int i{42};
    return f;
}

void bar () {
    string s{ foo () } ;
}
```



RVO:

```
string foo () {
      string  f{“lala”};
      int i{42};
      return f;
}

void bar () {
      string s{ foo () } ;
}
```



Check Point
SOFTWARE TECHNOLOGIES LTD

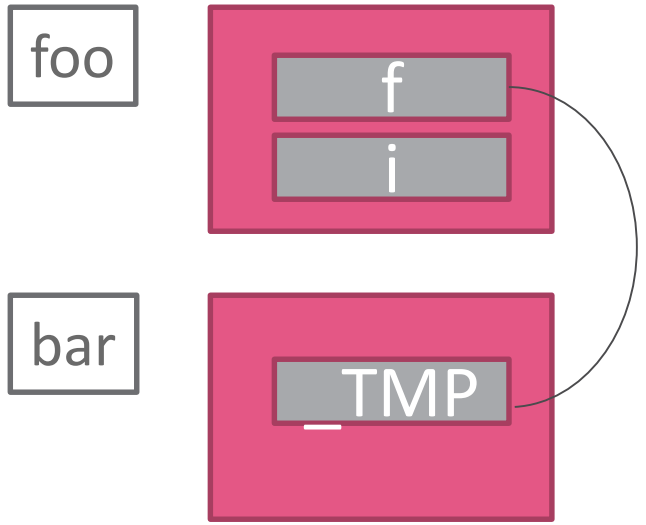
Argument Elision

Temporaries (PRvalues) passed to function by value, can be elided

```
void foo(std::string message) {…}

foo(“Wahoo”);
```

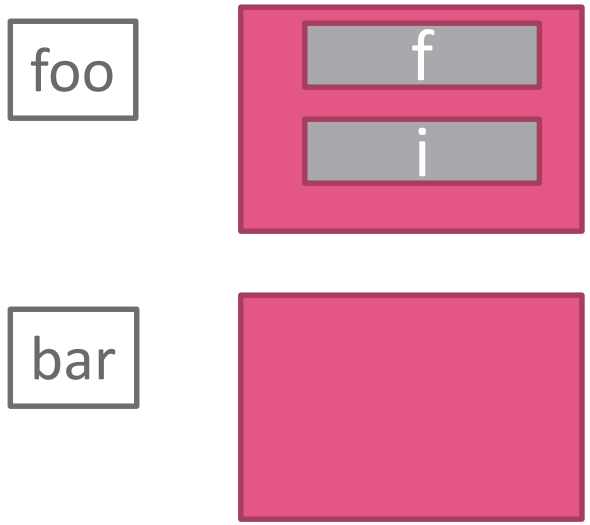
Without argument Elision

```
void foo (string f) {
      int i{42};
      std::cout <<f;
}

void bar () {
     string s{ foo (“lala”) } ;
}
```



Argument Elision

```
void foo (string f) {
      int i{42};
     std::cout <<f;
}

void bar () {
     string s{ “lala” } ;
}
```



Check Point
SOFTWARE TECHNOLOGIES LTD

Throw (starting C++11)

- Non-volatile object
- Automatic
- Not function parameter/catch clause parameter,
- Scope does not extend the innermost try-block (if any),

Catch (starting C++11)

- The same type as the exception object thrown, (ignoring top-level cv-qualification)
- Unless:
  - Change the observable behavior of the program, other than copy elision
  - For example, if the catch clause argument is modified, and the exception object is re-thrown

Check Point
SOFTWARE TECHNOLOGIES LTD

Destruction

- Destruction will take place when the latter object would have been destroyed, had copy not have been elided.

- They should be looked at as two ways to refer to the same object

Direct/Copy initialization

```
struct Foo {
  Foo(int) {}
  Foo(const Foo&) {
        std::cout <<
        "Foo has been copied /n"; }
};

int main() {
  Foo f1(1);
  Foo f2 = 2;
}
```

Direct initialization, Calls Foo::Foo(1);

Copy initialization

- Creates temporary Foo::Foo(2),
- Copies it to f2
- Similar to :

  Foo f2(Foo::Foo(Foo::Foo(2)))

  - What is the difference??
- Compiler may convert this to direct initialization

Return Rvalue, by value

```
Foo bar (Foo&& foo, …) {
  …
  return foo;
}
```

- Inside the function, foo is Lvalue
- Compiler is forced to copy it into return value location
- Use: *return std::move(foo);*
- Fine even if Foo does not have move semantics
- True also for forward reference
  - But need std::forward instead of move.

"This is wonderful, lets put std::move before every return"

Or

- *"I cant remember where I should put std::move, lets put it everywhere"*

Can this hurt?

Well, sure. Otherwise this slide would have been redundant…

Move on Temporary

```
Foo makeFoo(){
  Foo f;
  …
  return f;
}
```

Qualifies for RVO

No moving and no copy will take place

```
Foo makeFoo(){
  Foo f;
  …
  return std::move(f);
}
```

No longer qualifies to RVO.

- It is reference
- Compiler is forced to move/copy

Check Point
SOFTWARE TECHNOLOGIES LTD

Count on compiler optimization?

- If conditions for RVO are met, compiler may:
  - Elide the copy
  - Handled as Rvalue
    - Meaning std::move will be implicitly added.

- By-value arguments will be treated the same
  - Not qualified for RVO, but will be returned as Rvalue.

And in C++ 17

Non copy-able types

```
std::atomic<int> a1 (42);

std::atomic<int> a2 = 42;
```

Final action

```
template<typename F>
struct FinalAction {
    FinalAction(F f): clean{ f }   {}
   ~FinalAction() { clean(); }
    F clean;
};

template<class F>
     FinalAction<F> finally(F f) {
     return FinalAction<F>(f);
}
```

```
void test() {
  int* p=new int{7};
  auto act1 = finally( [&]{
                        delete p;
                        cout<<"Good bye, cruel world\n";
                    } );
}
```

Check Point
SOFTWARE TECHNOLOGIES LTD

The problem

- Compiler may generate 2 temporaries
  - Dtor will be called 3 times!
- We cannot forbid the copy/move ctor
  - Even if compiler will not use them – they must exist
- We must take special care while implementing them
  - And hope they will never be called!

Bind to const reference/rvalue ref

```
template<typename F>
struct Final_action {
    Final_action(F f): clean{ f }   {}
   ~Final_action() { clean(); }
    F clean;
};

template<class F>
    Final_action<F> finally(F f) {
    return {f};
}
```

```
void test() {
 int* p=new int{7};
 auto && act1 = finally( [&]{
                    delete p;
                    cout<<"Good bye, cruel world\n";
                } );
}
```

No initialization
Only reference

No temporary.
Only “recipe“

Move

- Still, we have temporaries
- code that must be maintained
- May not be faster than copy
- Weakens class invariants!

Check Point
SOFTWARE TECHNOLOGIES LTD

Guaranteed Copy elision

- If return statement is PRvalue of the same type as the function return type
- If variable initializer is PRvalue of the same type as the variable type

CV qualification ignored

Guaranteed Copy elision

```
template<typename F>
struct Final_action {
    Final_action(F f): clean{ f }   {}
   ~Final_action() { clean(); }
    F clean;
};

template<class F>
     Final_action<F> finally(F f) {
     return {f};
}
```

```
void test() {
  int* p=new int{7};
 auto  act1 = finally( [&]{
                          delete p;
                          cout<<"Good bye, cruel world\n";
                      } );
}
```

There is no temporary

- So nothing to copy/move from
- Copy/Move Ctor's may not be present/accessible
  - No copy/move can/will take place

```
auto  act1 = finally( .....);
```

Is equivalent to

```
auto  act1 = FinalAction ( ... );
```

Terminology

- "Guaranteed copy elision" is used in the standard
  - But it is not accurate
  - It does not guarantees elision
    - It eliminates them altogether
- The whole meaning of the expression is changed
  - So there is no copy to elide….
- Not an optimization
  - Core language change

Better names:

- “Unmaterialized value returning” (cppreference.com)
  - PRvalues are returned and used without materializing a temporary
- Deferred PRvalues materialization

Why bother with names…

```
template<class F>
    FinalAction<F> finally(F f) {
    return {f};
}
```

```
template<class F>
    FinalAction<F> finally(F f) {
    FinalAction<F> act{f};
   return act;
}
```

```
void test() {
 int* p=new int{7};
 auto  act1 = finally( [&]{
                    delete p;
                    cout<<"Good bye, cruel world\n";
                } );
}
```

And there is more to it…

Andrzej's C++ blog,

Rvalues redefiled