What are copy elision and return value optimization?

Asked 12 years, 2 months ago Modified 1 year, 6 months ago

Viewed 177k times

551

What is copy elision? What is (named) return value optimization? What do they imply?

In what situations can they occur? What are limitations?

- If you were referenced to this question, you're probably looking for **the introduction**.
- For a technical overview, see **the standard reference**.
- See **common cases** here.

c++ optimization c++-faq return-value-optimization copy-elision

Share

Improve this question

Follow

edited May 23, 2017 at 12:03

Community Bot

1 ● 1

asked Oct 18, 2012 at 11:03

Luchian Grigore

258k ● 66 ● 464 ● 630

2 Copy elision is one way to look at it; object elision or object fusion (or confusion) is another view. – curiousguy Aug 25, 2015 at 23:58

1 I found this link helpful. – subtleseeker Mar 2, 2020 at 8:30

5 Answers Sorted by: Highest score (default)

384

Introduction

For a technical overview - skip to this answer.

For common cases where copy elision occurs - skip to this answer.

Copy elision is an optimization implemented by most compilers to prevent extra (potentially expensive) copies in certain situations. It makes returning by value or pass-by-value feasible in practice (restrictions apply).

It's the only form of optimization that elides (ha!) the as-if rule - **copy elision can be applied even if copying/moving the object has side-effects**.

The following example taken from Wikipedia:

```
struct C {
  C() {}
  C(const C&) { std::cout << "A copy was made.\n"; }
};

C f() {
  return C();
}

int main() {
  std::cout << "Hello World!\n";
  C obj = f();
}
```

Depending on the compiler & settings, the following outputs **are all valid**:

> Hello World!
> A copy was made.
> A copy was made.

> Hello World!
> A copy was made.

> Hello World!

This also means fewer objects can be created, so you also can't rely on a specific number of destructors being called. You shouldn't have critical logic inside copy/move-constructors or destructors, as you can't rely on them being called.

If a call to a copy or move constructor is elided, that constructor must still exist and must be accessible. This ensures that copy elision does not allow copying objects which are not normally copyable, e.g. because they have a private or deleted copy/move constructor.

C++17: As of C++17, Copy Elision is guaranteed when an object is returned directly, and in this case, the copy or move constructor need **not** be accessible or present:

```
struct C {
  C() {}
  C(const C&) { std::cout << "A copy was made.\n"; }
};

C f() {
  return C(); //Definitely performs copy elision
}
C g() {
    C c;
    return c; //Maybe performs copy elision
}

int main() {
  std::cout << "Hello World!\n";
  C obj = f(); //Copy constructor isn't called
}
```

Share Improve this answer Follow

edited Jun 8, 2023 at 9:59
Y.T.
2,730 ● 2 ● 16 ● 33

answered Oct 18, 2012 at 11:03
Luchian Grigore
258k ● 66 ● 464 ● 630

5 could you plz explain when is the 2nd output happen and when the 3rd? – zhangxaochen Jun 19, 2014 at 14:00

3 @zhangxaochen when and how the compiler decides to optimize that way. – Luchian Grigore Jun 19, 2014 at 15:11

23 @zhangxaochen, 1st output: copy 1 is from the return to a temp, and copy 2 from temp to obj; 2nd is when one of the above is optimezed, probably the reutnr copy is elided; the thris both are elided – victor Nov 7, 2014 at 16:06

4 Hmm, but in my opinion, this MUST be a feature we can rely on. Because if we can't, it would severely affect the way we implement our functions in modern C++ (RVO vs std::move). During watching some of the CppCon 2014 videos, i really got the impression that all modern compilers always do RVO. Furthermore, I've read somewhere that also without any optimizations on, the compilers apply it. But, of course, I am not sure about it. That's why I am asking. – j00hi Feb 5, 2015 at 8:02

11 @j00hi: Never write move in a return statement - if rvo is not applied, the return value is moved out by default anyway. – MikeMB Mar 10, 2015 at 1:32

Common forms of copy elision

139

For a technical overview - skip to this answer.

For a less technical view & introduction - skip to this answer.

(Named) Return value optimization is a common form of copy elision. It refers to the situation where an object returned by value from a method has its copy elided. The example set forth in the standard illustrates **named return value optimization**, since the object is named.

```
class Thing {
public:
  Thing();
  ~Thing();
  Thing(const Thing&);
};
Thing f() {
  Thing t;
  return t;
}
Thing t2 = f();
```

Regular **return value optimization** occurs when a temporary is returned:

```
class Thing {
public:
  Thing();
  ~Thing();
  Thing(const Thing&);
};
Thing f() {
  return Thing();
}
Thing t2 = f();
```

Other common places where copy elision takes place is when an object is **constructed from a temporary**:

```
class Thing {
public:
  Thing();
  ~Thing();
  Thing(const Thing&);
};
void foo(Thing t);

Thing t2 = Thing();
Thing t3 = Thing(Thing()); // two rounds of elision
foo(Thing()); // parameter constructed from temporary
```

or when an **exception is thrown and caught by value**:

```
struct Thing{
  Thing();
  Thing(const Thing&);
};

void foo() {
  Thing c;
  throw c;
}

int main() {
  try {
    foo();
  }
  catch(Thing c) {
  }
}
```

Common limitations of copy elision are:

- multiple return points
- conditional initialization

Most commercial-grade compilers support copy elision & (N)RVO (depending on optimization settings). C++17 makes many of the above classes of copy elision mandatory.

Share Improve this answer Follow

edited Mar 24, 2022 at 14:18
Raymond Chen
45.2k ● 11 ● 99 ● 144

answered Oct 18, 2012 at 11:04
Luchian Grigore
258k ● 66 ● 464 ● 630

8 I'd be interested in seeing the "Common limitations" bullet points explained just a little bit... what makes these limiting factors? – phonetagger Jan 16, 2013 at 17:54

@phonetagger I linked against the msdn article, hope that clears some stuff out. – Luchian Grigore Jan 16, 2013 at 19:11

127

Standard reference

For a less technical view & introduction - skip to this answer.

For common cases where copy elision occurs - skip to this answer.

Copy elision is defined in the standard in:

12.8 Copying and moving class objects [class.copy]

as

> 31) When certain criteria are met, an implementation is allowed to omit the copy/move construction of a class object, even if the copy/move constructor and/or destructor for the object have side effects. In such cases, the implementation treats the source and target of the omitted copy/move operation as simply two different ways of referring to the same object, and the destruction of that object occurs at the later of the times when the two objects would have been destroyed without the optimization.[123] This elision of copy/move operations, called *copy elision*, is permitted in the following circumstances (which may be combined to eliminate multiple copies):
>
> — in a return statement in a function with a class return type, when the expression is the name of a non-volatile automatic object (other than a function or catch-clause parameter) with the same cvunqualified type as the function return type, the copy/move operation can be omitted by constructing the automatic object directly into the function's return value
>
> — in a throw-expression, when the operand is the name of a non-volatile automatic object (other than a function or catch-clause parameter) whose scope does not extend beyond the end of the innermost enclosing try-block (if there is one), the copy/move operation from the operand

> to the exception object (15.1) can be omitted by constructing the automatic object directly into the exception object
>
> — when a temporary class object that has not been bound to a reference (12.2) would be copied/moved to a class object with the same cv-unqualified type, the copy/move operation can be omitted by constructing the temporary object directly into the target of the omitted copy/move
>
> — when the exception-declaration of an exception handler (Clause 15) declares an object of the same type (except for cv-qualification) as the exception object (15.1), the copy/move operation can be omitted by treating the exception-declaration as an alias for the exception object if the meaning of the program will be unchanged except for the execution of constructors and destructors for the object declared by the exception-declaration.
>
> 123) Because only one object is destroyed instead of two, and one copy/move constructor is not executed, there is still one object destroyed for each one constructed.

The example given is:

```
class Thing {
public:
  Thing();
  ~Thing();
  Thing(const Thing&);
};
Thing f() {
  Thing t;
  return t;
}
Thing t2 = f();
```

and explained:

> Here the criteria for elision can be combined to eliminate two calls to the copy constructor of class `Thing`: the copying of the local automatic object `t` into the temporary object for the return value of function `f()` and the copying of that temporary object into object `t2`. Effectively, the construction of the local object `t` can be viewed as directly initializing the global object `t2`, and that object's destruction will occur at program exit. Adding a move constructor to Thing has the same effect, but it is the move construction from the temporary object to `t2` that is elided.

Share Improve this answer Follow

edited May 23, 2017 at 12:03
Community Bot
1 ● 1

answered Oct 18, 2012 at 11:03
Luchian Grigore
258k ● 66 ● 464 ● 630

3 Is that from the C++17 standard or from an earlier version? – Nils May 7, 2019 at 12:29

1 Why can't function parameter be return value optimized if it's the same type as function's return type? – Sahil Singh Jun 20, 2020 at 19:26

1 This tries to answer - stackoverflow.com/questions/9444485/… – Sahil Singh Jun 20, 2020 at 19:48

3 Is there any type of copy-elision for primitive types? If I have a function that propagates a return value (maybe an error code), will there be any optimisation similar to objects? – WARhead Aug 3, 2020 at 12:50

"Adding a move constructor to Thing has the same effect, but it is the move construction from the temporary object to t2 that is elided." should add ",and the move construction from t to the temporary that that is elided". ? Or am I missing something? – Jake1234 Oct 12, 2022 at 11:51

79

Copy elision is a compiler optimization technique that eliminates unnecessary copying/moving of objects.

In the following circumstances, a compiler is allowed to omit copy/move operations and hence not to call the associated constructor:

1. **NRVO (Named Return Value Optimization)**: If a function returns a class type by value and the return statement's expression is the name of a non-volatile object with automatic storage duration (which isn't a function parameter), then the copy/move that would

be performed by a non-optimising compiler can be omitted. If so, the returned value is constructed directly in the storage to which the function's return value would otherwise be moved or copied.

2. **RVO (Return Value Optimization)**: If the function returns a nameless temporary object that would be moved or copied into the destination by a naive compiler, the copy or move can be omitted as per 1.

```
#include <iostream>
using namespace std;

class ABC
{
public:
    const char *a;
    ABC()
     { cout<<"Constructor"<<endl; }
    ABC(const char *ptr)
     { cout<<"Constructor"<<endl; }
    ABC(ABC  &obj)
     { cout<<"copy constructor"<<endl;}
    ABC(ABC&& obj)
    { cout<<"Move constructor"<<endl; }
    ~ABC()
    { cout<<"Destructor"<<endl; }
};

ABC fun123()
{ ABC obj; return obj; }

ABC xyz123()
{  return ABC(); }

int main()
{
    ABC abc;
    ABC obj1(fun123());    //NRVO
    ABC obj2(xyz123());    //RVO, not NRVO
    ABC xyz = "Stack Overflow";//RVO
    return 0;
}

**Output without -fno-elide-constructors**
root@ajay-PC:/home/ajay/c++# ./a.out
Constructor
Constructor
Constructor
Constructor
Destructor
Destructor
Destructor
Destructor

**Output with -fno-elide-constructors**
root@ajay-PC:/home/ajay/c++# g++ -std=c++11 copy_elisi
constructors
root@ajay-PC:/home/ajay/c++# ./a.out
Constructor
Constructor
Move constructor
Destructor
Move constructor
Destructor
Constructor
Move constructor
Destructor
Move constructor
Destructor
Constructor
Move constructor
Destructor
Destructor
Destructor
Destructor
Destructor
```

Even when copy elision takes place and the copy-/move-constructor is not called, it must be present and accessible (as if no optimization happened at all), otherwise the program is ill-formed.

You should permit such copy elision only in places where it won't affect the observable behavior of your software. Copy elision is the only form of optimization permitted to have (i.e. elide) observable side-effects. Example:

```
#include <iostream>
int n = 0;
class ABC
{  public:
 ABC(int) {}
 ABC(const ABC& a) { ++n; } // the copy constructor ha
};                         // it modifies an object with s

int main()
{
  ABC c1(21); // direct-initialization, calls C::C(42)
  ABC c2 = ABC(21); // copy-initialization, calls C::C

  std::cout << n << std::endl; // prints 0 if the copy
  return 0;
}

Output without -fno-elide-constructors
root@ajay-PC:/home/ayadav# g++ -std=c++11 copy_elision
root@ajay-PC:/home/ayadav# ./a.out
0

Output with -fno-elide-constructors
root@ajay-PC:/home/ayadav# g++ -std=c++11 copy_elision
constructors
root@ajay-PC:/home/ayadav# ./a.out
1
```

GCC provides the `-fno-elide-constructors` option to disable copy elision. If you want to avoid possible copy elision, use `-fno-elide-constructors`.

Now almost all compilers provide copy elision when optimisation is enabled (and if no other option is set to disable it).

Conclusion

With each copy elision, one construction and one matching destruction of the copy are omitted, thus saving CPU time, and one object is not created, thus saving space on the stack frame.

Share Improve this answer

Follow

edited Jul 25, 2021 at 15:09

John
3,424 ● 1 ● 16 ● 44

answered Jan 13, 2015 at 7:26

Ajay yadav
4,481 ● 4 ● 33 ● 42

9 the statement `ABC obj2(xyz123());` is it NRVO or RVO? is it not getting temporary variable/object same as `ABC xyz = "Stack Overflow";//RVO` – Asif Mushtaq Aug 28, 2015 at 2:45

4 To have a more concrete illustration of RVO, you can refer to the assembly that the compiler generates (change the compiler flag -fno-elide-constructors to see the diff). godbolt.org/g/Y2KcdH – Gab是好人 Dec 3, 2016 at 16:18

3 Isn't ABC xyz = "Stack Overflow"; just an implicit call to ABC::ABC(const char *ptr) rather than RVO? – user1079475

Aug 27, 2020 at 17:42

for ABC xyz = "Stack Overflow"; is calls explicitly defined copy constructor, so I am not sure how this can be RVO, and the word R- suggests that there is a return value from function, however constructors don't have return values. – Nusrat Nuriyev Jul 17, 2023 at 19:24

More interestingly, ABC obj1(fun123()) this can be elided, however on C++17 and C++20 leave this as optional, so if -fno-elide-constructors is ON then def + move constructor will be called. – Nusrat Nuriyev Jul 17, 2023 at 19:36

-1

Here I give another example of copy elision that I apparently encountered today.

```
# include <iostream>


class Obj {
public:
  int var1;
  Obj(){
    std::cout<<"In   Obj()"<<"\n";
    var1 =2;
  };
  Obj(const Obj & org){
    std::cout<<"In   Obj(const Obj & org)"<<"\n";
    var1=org.var1+1;
  };
};

int  main(){

  {
    /*const*/ Obj Obj_instance1;  //const doesn't chan
    Obj Obj_instance2;
    std::cout<<"assignment:"<<"\n";
    Obj_instance2=Obj(Obj(Obj(Obj(Obj_instance1))))
    // in fact expected: 6, but got 3, because of 'cop
    std::cout<<"Obj_instance2.var1:"<<Obj_instance2.va
  }

}
```

With the result:

```
In   Obj()
In   Obj()
assignment:
In   Obj(const Obj & org)
Obj_instance2.var1:3
```

Share Improve this answer Follow

answered Oct 15, 2020 at 14:27

K.Karamazen
186 ● 1 ● 8

2 That's already included in Luchian's answer (temporary object passed by value). – Toby Speight Dec 23, 2020 at 13:45

This answer can be a side note to the other answers. It shows an interesting extended case. In C++14, turning on -fno-elide-constructors gives 6 and without the option we get 3. In C++17 and above we get 3 always. – Dhwani Katagade Dec 8, 2022 at 11:17