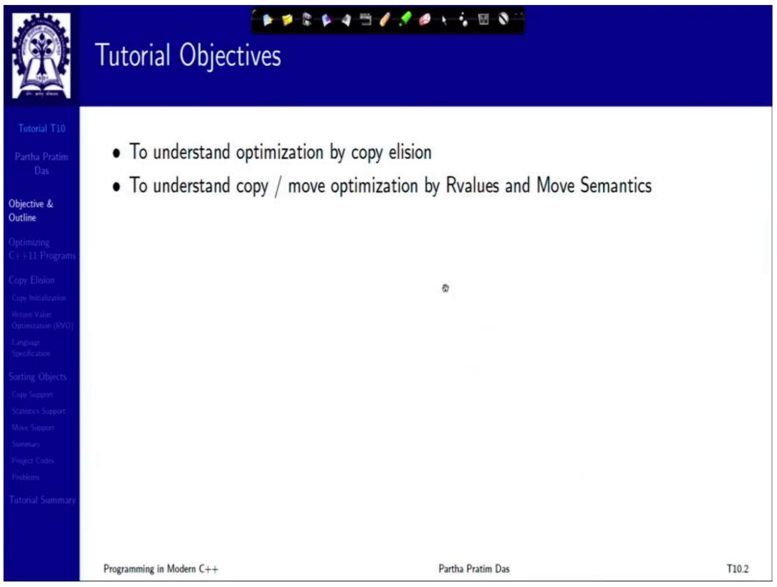
Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Tutorial 10
How to optimize C++ 11 programs using rvalue and move Semantics?

Welcome to Programming in modern C++. We are going to discuss tutorial 10. How to optimize C++ programs using rvalue and move Semantics?

(Refer Slide Time: 0:44)



Tutorial Objectives

- To understand optimization by copy elision
- To understand copy / move optimization by Rvalues and Move Semantics

Tutorial T10

Partha Pratim Das

Objective & Outline

Optimizing C++11 Programs

Copy Elision

Copy Initialization

Return Value Optimization (RVO)

Language Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

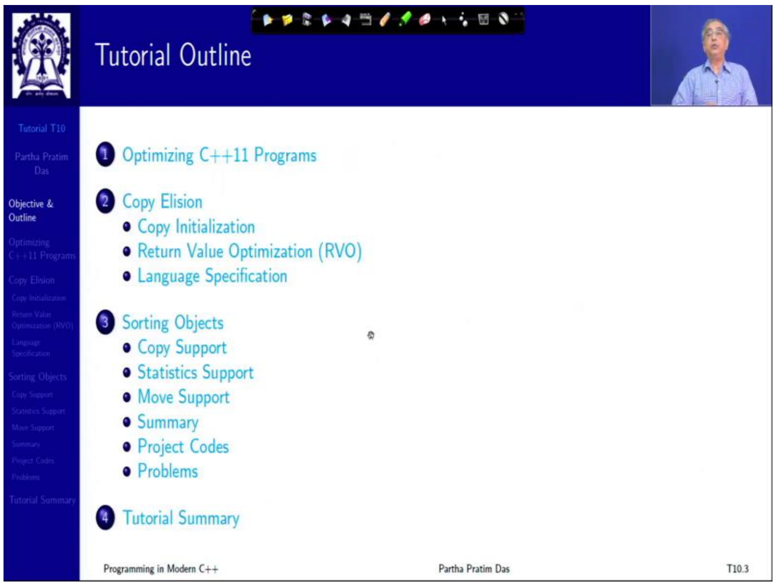
Problems

Tutorial Summary

Programming in Modern C++ Partha Pratim Das T10.2

We have during the course of our discussion on rvalue references universal references and the move operations. We have seen that this support can significantly reduce the necessity to copy particularly the temporary objects. In this tutorial, we are going to discuss a number of examples to illustrate which of these optimizations can as it is be done by the compiler and for which it needs the support from the program to suggest to the compiler as to what can be optimized out that is what can be treated as a temporary object as an rvalue.

(Refer Slide Time: 1:41)



Tutorial Outline

Tutorial T10

Partha Pratim Das

Objective & Outline

Optimizing C++11 Programs

Copy Elision
Copy Initialization
Return Value Optimization (RVO)
Language Specification

Sorting Objects
Copy Support
Statistics Support
Move Support
Summary
Project Codes
Problems

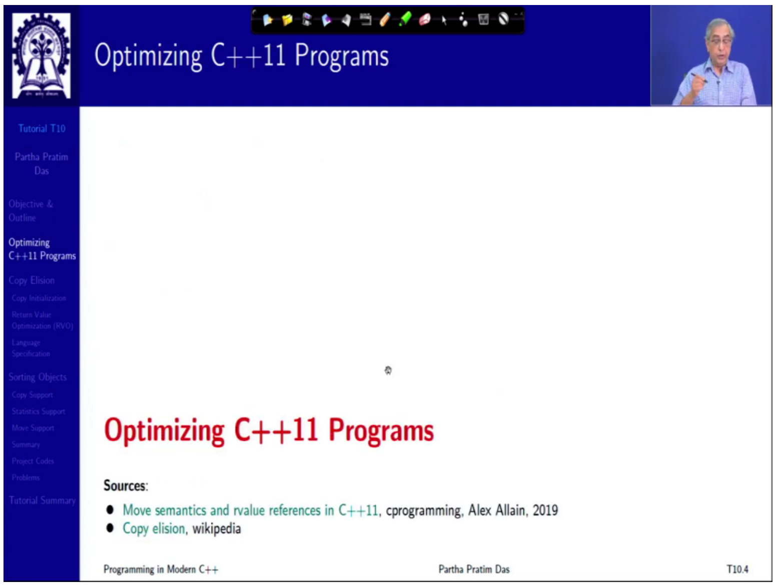
Tutorial Summary

1. Optimizing C++11 Programs
2. Copy Elision
   - Copy Initialization
   - Return Value Optimization (RVO)
   - Language Specification
3. Sorting Objects
   - Copy Support
   - Statistics Support
   - Move Support
   - Summary
   - Project Codes
   - Problems
4. Tutorial Summary

Programming in Modern C++ Partha Pratim Das T10.3

And we will get towards building up a small but complete project of sorting an arbitrary data type containing resources to understand this.

(Refer Slide Time: 01:55)



Optimizing C++11 Programs

Tutorial T10

Partha Pratim Das

Objective & Outline

Optimizing C++11 Programs

Copy Elision
Copy Initialization
Return Value Optimization (RVO)
Language Specification

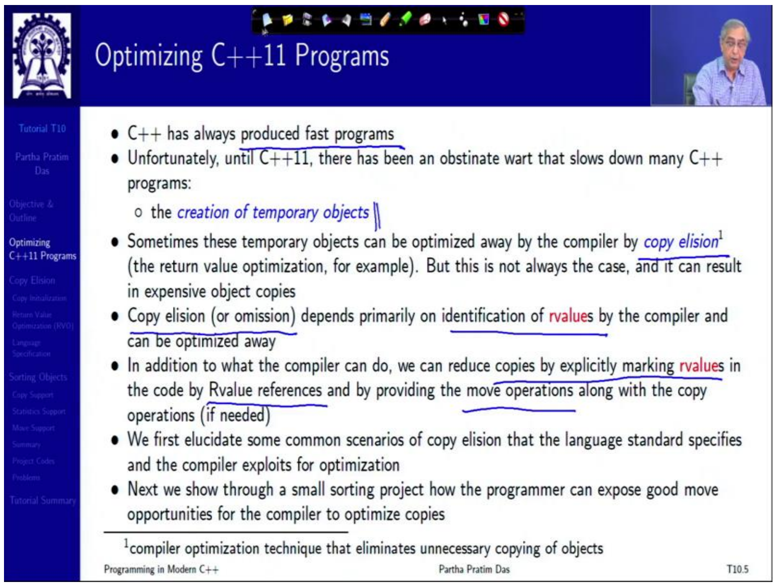
Sorting Objects
Copy Support
Statistics Support
Move Support
Summary
Project Codes
Problems

Tutorial Summary

Optimizing C++11 Programs

Sources:
- Move semantics and rvalue references in C++11, cprogramming, Alex Allain, 2019
- Copy elision, wikipedia

Programming in Modern C++ Partha Pratim Das T10.4



Optimizing C++11 Programs

- C++ has always produced fast programs
- Unfortunately, until C++11, there has been an obstinate wart that slows down many C++ programs:
  - the *creation of temporary objects*
- Sometimes these temporary objects can be optimized away by the compiler by *copy elision*[1] (the return value optimization, for example). But this is not always the case, and it can result in expensive object copies
- Copy elision (or omission) depends primarily on identification of rvalues by the compiler and can be optimized away
- In addition to what the compiler can do, we can reduce copies by explicitly marking rvalues in the code by Rvalue references and by providing the move operations along with the copy operations (if needed)
- We first elucidate some common scenarios of copy elision that the language standard specifies and the compiler exploits for optimization
- Next we show through a small sorting project how the programmer can expose good move opportunities for the compiler to optimize copies

[1] compiler optimization technique that eliminates unnecessary copying of objects

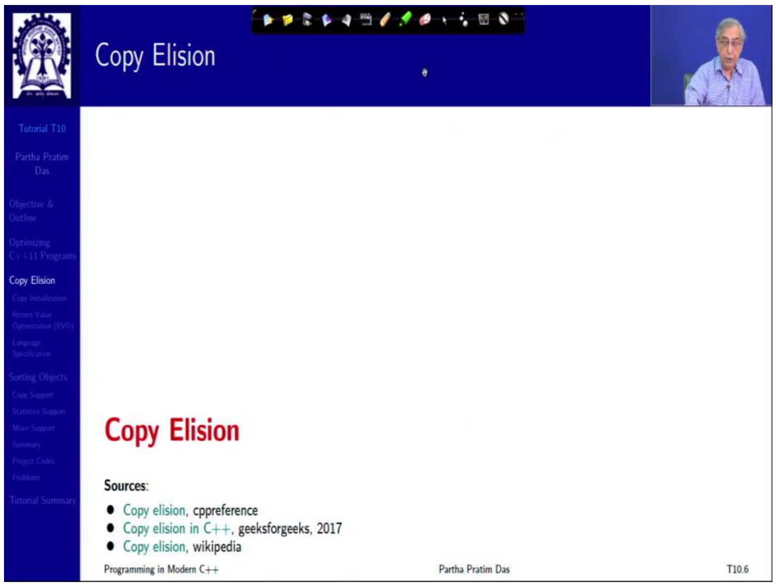
Programming in Modern C++ Partha Pratim Das T10.5

So, if we talk about optimizing C++ program obviously, you know there are several optimizations and C++ has always produced fast programs. But unfortunately, there is one problem area which slows down many C++ program that is creation of temporary objects. C++ to keep its object orientation to keep its implementation of various semantics, need to regularly create temporary objects and delete them.

And when you put those things together, then often you have a lot of temporary objects which are created and deleted. Now, sometimes, these objects can be optimized away by the compiler. And that process is known as copy elision. Copy elision is an optimization technique that remove eliminate the unnecessary copying of objects. Copy elision primarily depends on the identification of rvalues by the compiler, because you know, that rvalues can be removed, can be optimized away once they are used.

Because since they do not have a name, they cannot be used by the programmer, and therefore, after it their use if they are removed, then it does not really impact the correctness of the program. In addition, what the compiler can do is, in addition to what the compiler can do, we can also help the compiler by explicitly marking rvalues. And how do we do that, we can do that by providing rvalue references or by providing move operations along with the copy operations if they are at all needed. So, in this tutorial, we are going to elucidate some of these scenarios of copy elision and the sorting project that I just mentioned.

(Refer Slide Time: 4:00)


Copy Elision

Tutorial T10

Partha Pratim Das

Objective & Outline

Optimizing C++11 Programs

Copy Elision

Sorting Objects

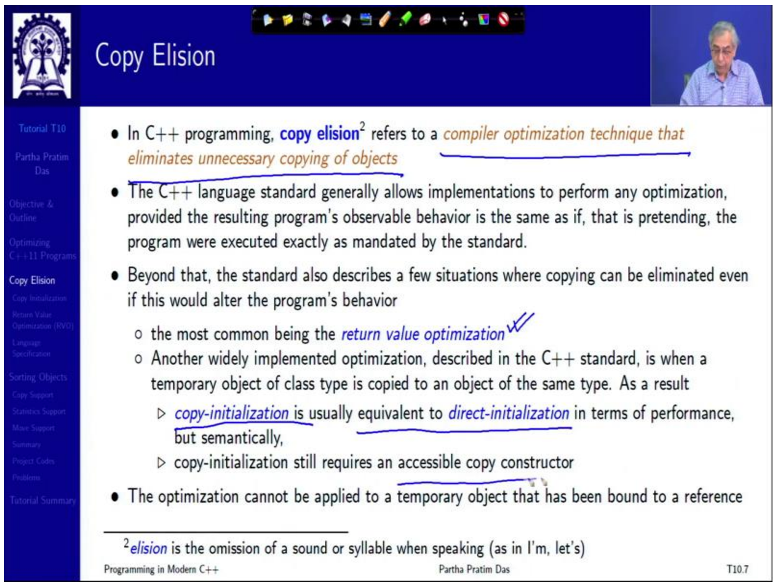
Tutorial Summary

Copy Elision

Sources:

- Copy elision, cppreference
- Copy elision in C++, geeksforgeeks, 2017
- Copy elision, wikipedia

Programming in Modern C++ Partha Pratim Das T10.6


Copy Elision

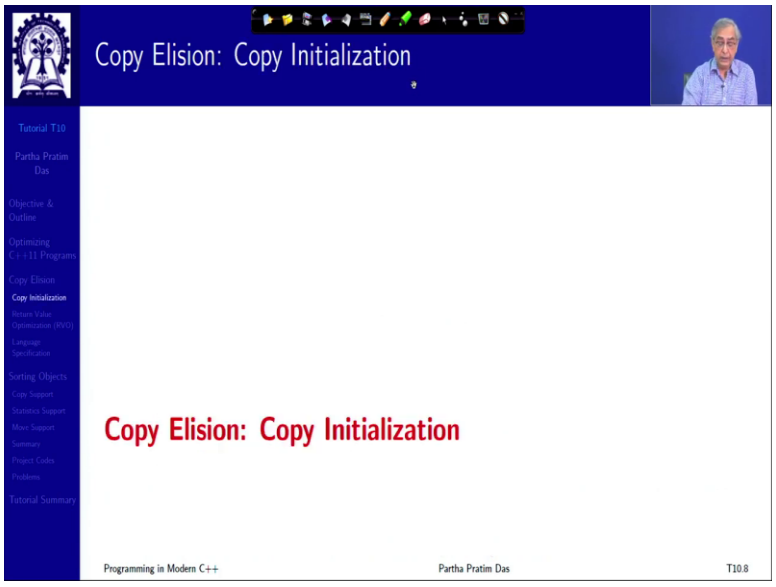
- In C++ programming, **copy elision**[2] refers to a *compiler optimization technique that eliminates unnecessary copying of objects*
- The C++ language standard generally allows implementations to perform any optimization, provided the resulting program's observable behavior is the same as if, that is pretending, the program were executed exactly as mandated by the standard.
- Beyond that, the standard also describes a few situations where copying can be eliminated even if this would alter the program's behavior
  - the most common being the *return value optimization*
  - Another widely implemented optimization, described in the C++ standard, is when a temporary object of class type is copied to an object of the same type. As a result
    - *copy-initialization* is usually equivalent to *direct-initialization* in terms of performance, but semantically,
    - copy-initialization still requires an accessible copy constructor
- The optimization cannot be applied to a temporary object that has been bound to a reference

[2] *elision* is the omission of a sound or syllable when speaking (as in I'm, let's)

Programming in Modern C++ Partha Pratim Das T10.7

So, let us first look at copy elision. As I told it is a compiler optimization technique that eliminates unnecessary copying of objects. And there are several such which exists the most common being the return value optimization. I have talked about this earlier, but in this tutorial, I am going to talk about it in little bit longer length. The other place where copy elision can be used effectively is copy initialization. It is usually equivalent to a direct initialization in terms of performance, but semantically it may be different because it requires an accessible copy constructor. So, let us see what all can be done in terms of copy elision.

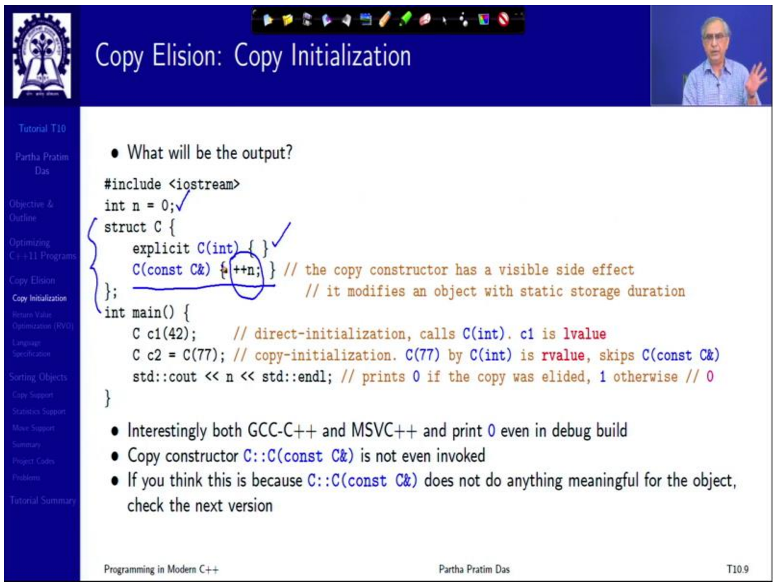
(Refer Slide Time: 4:56)



Copy Elision: Copy Initialization

Tutorial T10
Partha Pratim Das
Objective & Outline
Optimizing C++11 Programs
Copy Elision
Copy Initialization
Return Value Optimization (RVO)
Language Specification
Sorting Objects
Copy Support
Statistics Support
Move Support
Summary
Project Codes
Problems
Tutorial Summary

Copy Elision: Copy Initialization



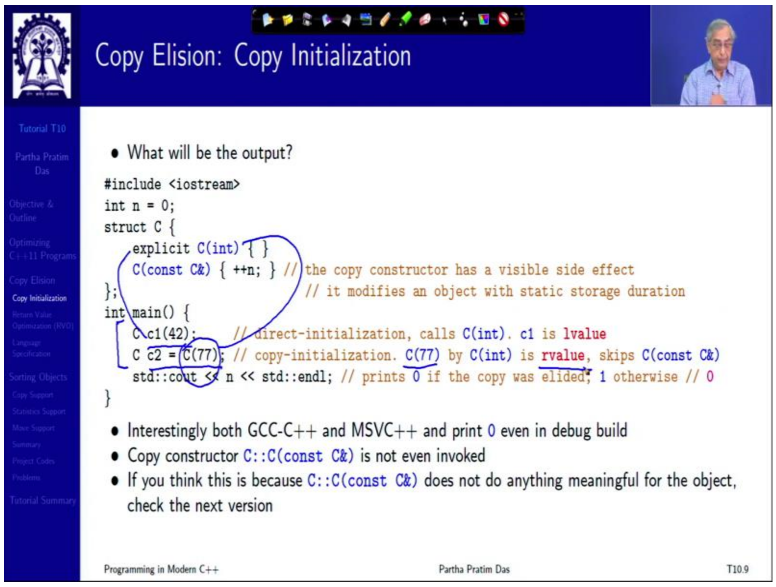
Copy Elision: Copy Initialization

Tutorial T10
Partha Pratim Das
Objective & Outline
Optimizing C++11 Programs
Copy Elision
Copy Initialization
Return Value Optimization (RVO)
Language Specification
Sorting Objects
Copy Support
Statistics Support
Move Support
Summary
Project Codes
Problems
Tutorial Summary

- What will be the output?

```
#include <iostream>
int n = 0;
struct C {
    explicit C(int) { }
    C(const C&) { ++n; } // the copy constructor has a visible side effect
};                       // it modifies an object with static storage duration
int main() {
    C c1(42);      // direct-initialization, calls C(int). c1 is lvalue
    C c2 = C(77); // copy-initialization. C(77) by C(int) is rvalue, skips C(const C&)
    std::cout << n << std::endl; // prints 0 if the copy was elided, 1 otherwise // 0
}
```

- Interestingly both GCC-C++ and MSVC++ and print 0 even in debug build
- Copy constructor `C::C(const C&)` is not even invoked
- If you think this is because `C::C(const C&)` does not do anything meaningful for the object, check the next version



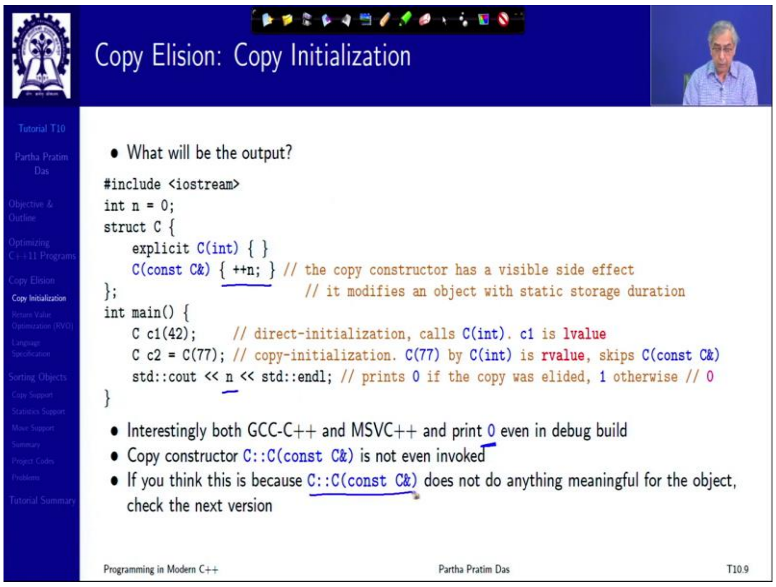
Copy Elision: Copy Initialization

Tutorial T10
Partha Pratim Das
Objective & Outline
Optimizing C++11 Programs
Copy Elision
Copy Initialization
Return Value Optimization (RVO)
Language Specification
Sorting Objects
Copy Support
Statistics Support
Move Support
Summary
Project Codes
Problems
Tutorial Summary

- What will be the output?

```
#include <iostream>
int n = 0;
struct C {
    explicit C(int) { }
    C(const C&) { ++n; } // the copy constructor has a visible side effect
};                       // it modifies an object with static storage duration
int main() {
    C c1(42);      // direct-initialization, calls C(int). c1 is lvalue
    C c2 = C(77); // copy-initialization. C(77) by C(int) is rvalue, skips C(const C&)
    std::cout << n << std::endl; // prints 0 if the copy was elided, 1 otherwise // 0
}
```

- Interestingly both GCC-C++ and MSVC++ and print 0 even in debug build
- Copy constructor `C::C(const C&)` is not even invoked
- If you think this is because `C::C(const C&)` does not do anything meaningful for the object, check the next version


Tutorial T10

Partha Pratim Das

Objective & Outline

Optimizing C++11 Programs

Copy Elision

Copy Initialization

Return Value Optimization (RVO)

Language Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

Copy Elision: Copy Initialization

- What will be the output?

```
#include <iostream>
int n = 0;
struct C {
    explicit C(int) { }
    C(const C&) { ++n; } // the copy constructor has a visible side effect
};                          // it modifies an object with static storage duration
int main() {
    C c1(42);     // direct-initialization, calls C(int). c1 is lvalue
    C c2 = C(77); // copy-initialization. C(77) by C(int) is rvalue, skips C(const C&)
    std::cout << n << std::endl; // prints 0 if the copy was elided, 1 otherwise // 0
}
```

- Interestingly both GCC-C++ and MSVC++ and print 0 even in debug build
- Copy constructor `C::C(const C&)` is not even invoked
- If you think this is because `C::C(const C&)` does not do anything meaningful for the object, check the next version

Programming in Modern C++ Partha Pratim Das T10.9

So, first, let us look at copy initialization of objects. I have a very small class see here most often I have written classes as struct. Because the issue of encapsulation or access restriction is not the focus to discuss whatever we are discussing in terms of structs will apply equally well for the classes altogether. So, we have an explicit constructor, which is a parametric constructor, and it has a copy constructor which does nothing but there is a global variable n which the copy constructor is supposed to increment.

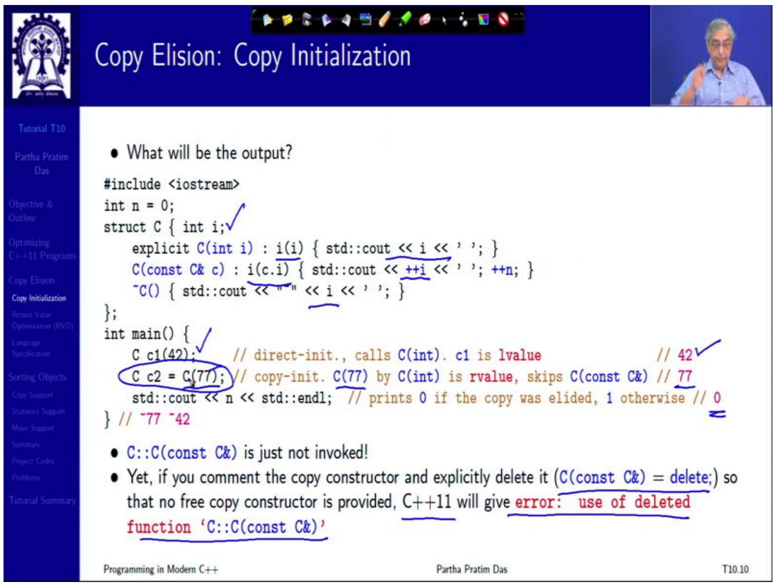
And we do not do we have a parameter constructor we do not have any data, because we are not interested in looking at the data member values and so on immediately. So, in the context of this, we have removed these two-object initialization you can easily make out that this is a direct initialization where we are expected to invoke the parameter constructor and see one after that invocation will become an added value.

The next one, as we will identify is a copy initialization C(77). This particular object will be constructed by the parametric constructor and then from that a copy initialization will be will happen to c2. And this is it is to be noted that C(77) is rvalue here. The interesting thing is if you actually execute this program, and to test whether the copy initialization is or the copy constructor is getting invoked, or how many times it is getting invoked, we are incrementing n every time it is invoked, and we at the end, we print n.

Now, if you execute this program, irrespective of whether you do that in GCC, or in Microsoft Visual C++, whether you do it in release build or in debug build whatever, it will always print 0. And you will get amazed because, you know, it is not only that it is keeping the copy construction, but the copy construction has a side effect it is keeping that as well.

And before you think that this is an error of the compiler, let me tell you that this actually is a language specification. Now, you might think, it is not doing something because the copy constructor does not do anything meaningful. So, let us try to put something meaningful.

(Refer Slide Time: 7:52)



Copy Elision: Copy Initialization

Tutorial T10

Partha Pratim Das

Objective & Outline

Optimizing C++11 Programs

Copy Elision

Copy Initialization

Return Value Optimization (RVO)

Language Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

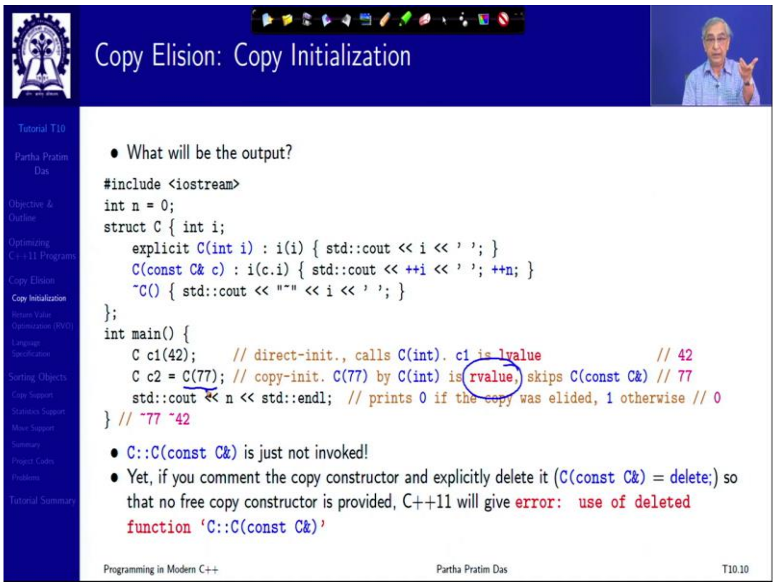
Tutorial Summary

- What will be the output?

```
#include <iostream>
int n = 0;
struct C { int i;
    explicit C(int i) : i(i) { std::cout << i << ' '; }
    C(const C& c) : i(c.i) { std::cout << ++i << ' '; ++n; }
    ~C() { std::cout << "~" << i << ' '; }
};
int main() {
    C c1(42);     // direct-init., calls C(int). c1 is lvalue              // 42
    C c2 = C(77); // copy-init. C(77) by C(int) is rvalue, skips C(const C&) // 77
    std::cout << n << std::endl;  // prints 0 if the copy was elided, 1 otherwise // 0
} // ~77 ~42
```

- C::C(const C&) is just not invoked!
- Yet, if you comment the copy constructor and explicitly delete it (C(const C&) = delete;) so that no free copy constructor is provided, C++11 will give error: use of deleted function 'C::C(const C&)'

Programming in Modern C++ Partha Pratim Das T10.10



Copy Elision: Copy Initialization

Tutorial T10

Partha Pratim Das

Objective & Outline

Optimizing C++11 Programs

Copy Elision

Copy Initialization

Return Value Optimization (RVO)

Language Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

- What will be the output?

```
#include <iostream>
int n = 0;
struct C { int i;
    explicit C(int i) : i(i) { std::cout << i << ' '; }
    C(const C& c) : i(c.i) { std::cout << ++i << ' '; ++n; }
    ~C() { std::cout << "~" << i << ' '; }
};
int main() {
    C c1(42);     // direct-init., calls C(int). c1 is lvalue              // 42
    C c2 = C(77); // copy-init. C(77) by C(int) is rvalue, skips C(const C&) // 77
    std::cout << n << std::endl;  // prints 0 if the copy was elided, 1 otherwise // 0
} // ~77 ~42
```

- C::C(const C&) is just not invoked!
- Yet, if you comment the copy constructor and explicitly delete it (C(const C&) = delete;) so that no free copy constructor is provided, C++11 will give error: use of deleted function 'C::C(const C&)'

Programming in Modern C++ Partha Pratim Das T10.10

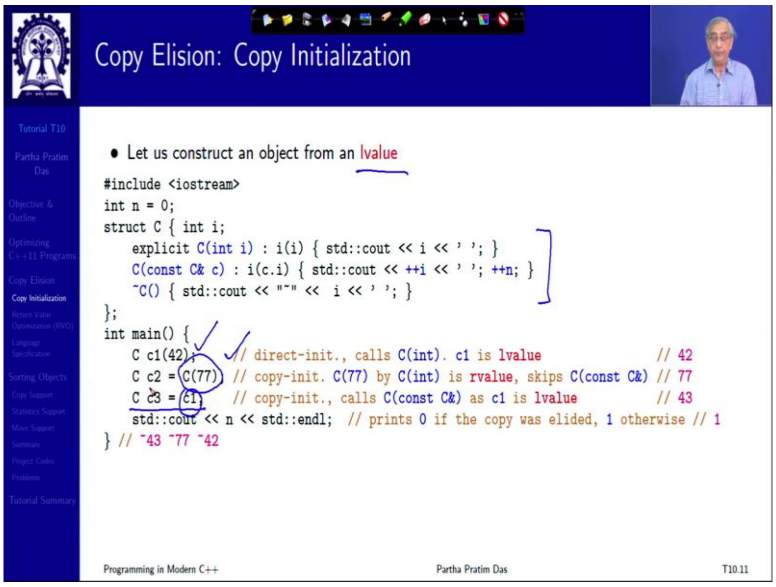
In that let us introduce a data member, let us initialize the data member in the parameter constructor, print the value of the data member. Also, let us copy the appropriate data member value in the copy constructor and print its value with one increment. This increment will tell us that this is getting printed from the copy constructor. Then we have a destructor where all so I am printing the value so that we can trace exactly what is happening.

Now, if you do that, this is the parameterized constructor. This prints 42 as we expect, here, first see 72 is expected to be executed. So, that will call the parametric constructor, so it will print 72. After that you would have expected a copy to happen. But that copy call does not happen. There is no printing of 78 happening from here. So, again even though there is a direct data member to initialize and things to do, those get skipped by the constructor, that is the copy constructor does not happen. And, and therefore prints us 0.

In fact, if you if you comment the copy constructor, and of course if he will say that if a copy comments the copy constructor, then also how do I know that copy constructor is actually invoked or not? Because the compiler will give me a free copy constructor. So, you can comment out the copy constructor and explicitly delete it. You have already learned how to delete a function in a class explicitly delete that so no copy of free copy constructor is provided.

Then the C++ compiler will give an error the use of deleted function. I mean, is not it a little funny because if I provided it does not invoke it, but if I say that it is not there, then it does complain. So, that is what I was saying that here the behavior is like the direct initialization, which is very similar to the copy behavior. But semantically it is different because it, it still wants that a copy constructor must be accessible. So, it has to be accessible then it does not use it. And now in all this, if you have to understand why this is happening, you have to note that the C(77) is rvalue, because it is a temporary object which you have no control to hold on.

(Refer Slide Time: 10:41)

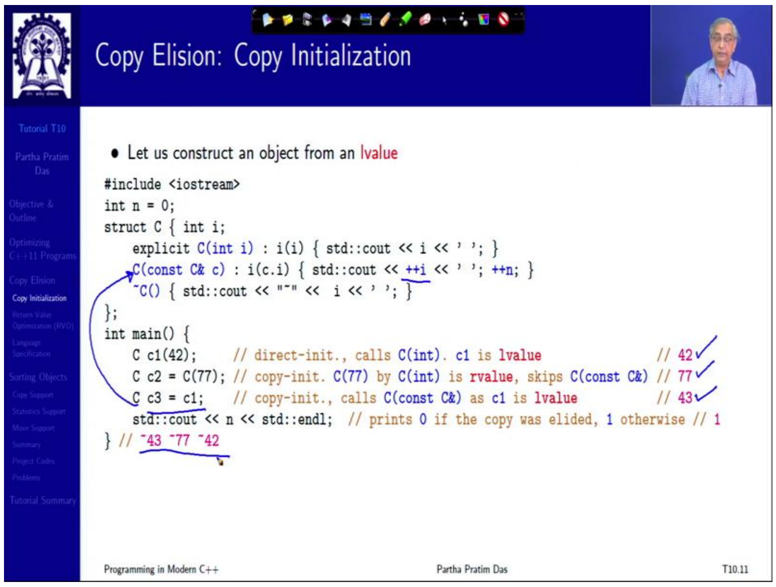


Copy Elision: Copy Initialization

- Let us construct an object from an lvalue

```
#include <iostream>
int n = 0;
struct C { int i;
    explicit C(int i) : i(i) { std::cout << i << ' '; }
    C(const C& c) : i(c.i) { std::cout << ++i << ' '; ++n; }
    ~C() { std::cout << "~" <<  i << ' '; }
};
int main() {
    C c1(42);      // direct-init., calls C(int). c1 is lvalue              // 42
    C c2 = C(77)  // copy-init. C(77) by C(int) is rvalue, skips C(const C&) // 77
    C c3 = c1;    // copy-init., calls C(const C&) as c1 is lvalue          // 43
    std::cout << n << std::endl;  // prints 0 if the copy was elided, 1 otherwise // 1
} // ~43 ~77 ~42
```

Programming in Modern C++ Partha Pratim Das T10.11



Copy Elision: Copy Initialization

Tutorial T10
Partha Pratim Das
Objective & Outline
Optimizing C++11 Programs
Copy Elision
Copy Initialization
Return Value Optimization (RVO)
Language Specification
Sorting Objects
Copy Support
Statistics Support
Move Support
Summary
Project Codes
Problems
Tutorial Summary

- Let us construct an object from an lvalue

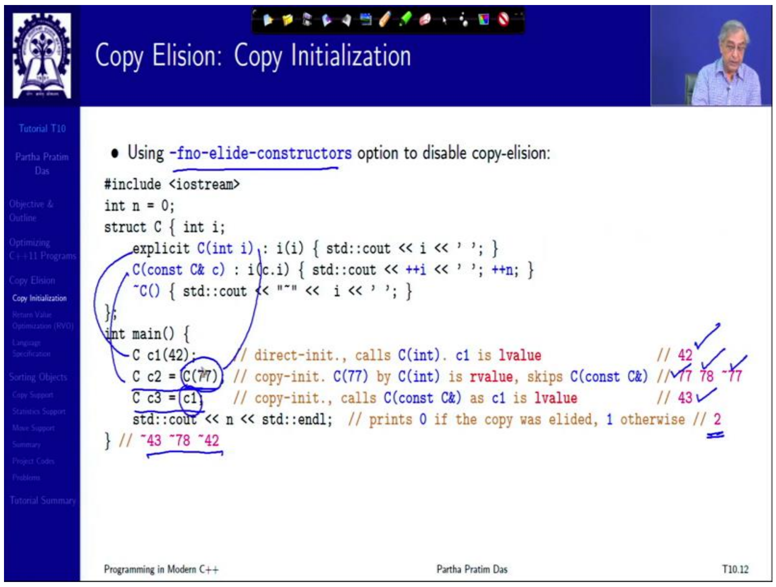
```
#include <iostream>
int n = 0;
struct C { int i;
    explicit C(int i) : i(i) { std::cout << i << ' '; }
    C(const C& c) : i(c.i) { std::cout << ++i << ' '; ++n; }
    ~C() { std::cout << "~" << i << ' '; }
};
int main() {
    C c1(42);     // direct-init., calls C(int). c1 is lvalue                  // 42
    C c2 = C(77); // copy-init. C(77) by C(int) is rvalue, skips C(const C&) // 77
    C c3 = c1;    // copy-init., calls C(const C&) as c1 is lvalue            // 43
    std::cout << n << std::endl;  // prints 0 if the copy was elided, 1 otherwise // 1
} // ~43 ~77 ~42
```

Programming in Modern C++ Partha Pratim Das T10.11

Understanding that, let us see, that if we now change this code slightly to construct an object from lvalue. So, these remain same, all that this is same, this is same, but I add a third initialization for instance c3 from c1 where you can see c1 is lvalue, whereas this one is an rvalue. Now, you see something interesting, it does as before, this also does as before, but as you try to copy constructor, as you try to construct c3 from c1 which is lvalue, it actually invokes the copy constructor.

And as it invokes the copy constructor c1 has a member value 42. So, it increments it to 43. So, it prints 43. And at the end of the program, you will now see 3 objects getting deleted with their respective values destructed with their respective values. So, this is this is the basic copy elision that the compiler does.

(Refer Slide Time: 11:47)



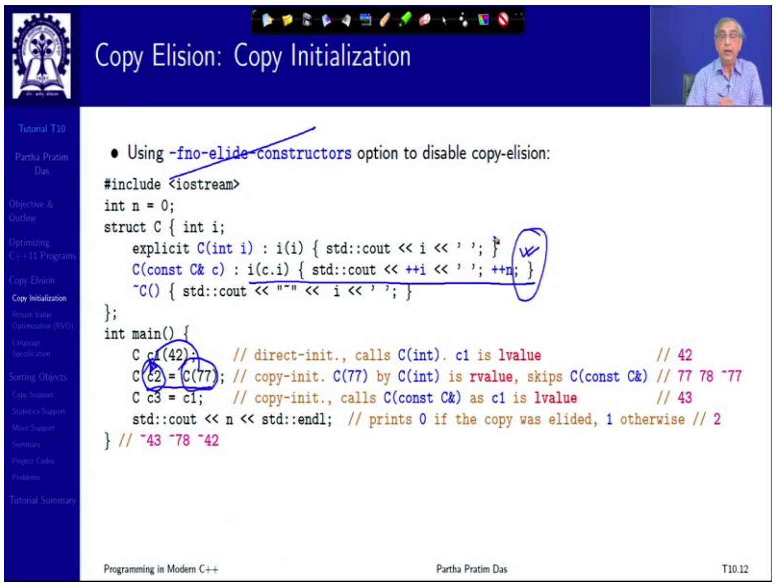
Copy Elision: Copy Initialization

Tutorial T10
Partha Pratim Das
Objective & Outline
Optimizing C++11 Programs
Copy Elision
Copy Initialization
Return Value Optimization (RVO)
Language Specification
Sorting Objects
Copy Support
Statistics Support
Move Support
Summary
Project Codes
Problems
Tutorial Summary

- Using -fno-elide-constructors option to disable copy-elision:

```
#include <iostream>
int n = 0;
struct C { int i;
    explicit C(int i) : i(i) { std::cout << i << ' '; }
    C(const C& c) : i(c.i) { std::cout << ++i << ' '; ++n; }
    ~C() { std::cout << "~" << i << ' '; }
};
int main() {
    C c1(42);     // direct-init., calls C(int). c1 is lvalue                  // 42
    C c2 = C(77); // copy-init. C(77) by C(int) is rvalue, skips C(const C&) // 77 78 ~77
    C c3 = c1;    // copy-init., calls C(const C&) as c1 is lvalue            // 43
    std::cout << n << std::endl;  // prints 0 if the copy was elided, 1 otherwise // 2
} // ~43 ~78 ~42
```

Programming in Modern C++ Partha Pratim Das T10.12



Copy Elision: Copy Initialization

Tutorial T10

Partha Pratim Das

Objective & Outline

Optimizing C++11 Programs

Copy Elision

Copy Initialization

Return Value Optimization (RVO)

Language Specification

Sorting Objects

Copy Support

Stateless Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

- Using -fno-elide-constructors option to disable copy-elision:

```
#include <iostream>
int n = 0;
struct C { int i;
    explicit C(int i) : i(i) { std::cout << i << ' '; }
    C(const C& c) : i(c.i) { std::cout << ++i << ' '; ++n; }
    ~C() { std::cout << "~" <<  i << ' '; }
};
int main() {
    C c1(42);     // direct-init., calls C(int). c1 is lvalue                 // 42
    C c2 = C(77); // copy-init. C(77) by C(int) is rvalue, skips C(const C&) // 77 78 ~77
    C c3 = c1;    // copy-init., calls C(const C&) as c1 is lvalue            // 43
    std::cout << n << std::endl;  // prints 0 if the copy was elided, 1 otherwise // 2
} // ~43 ~78 ~42
```

Programming in Modern C++ Partha Pratim Das T10.12

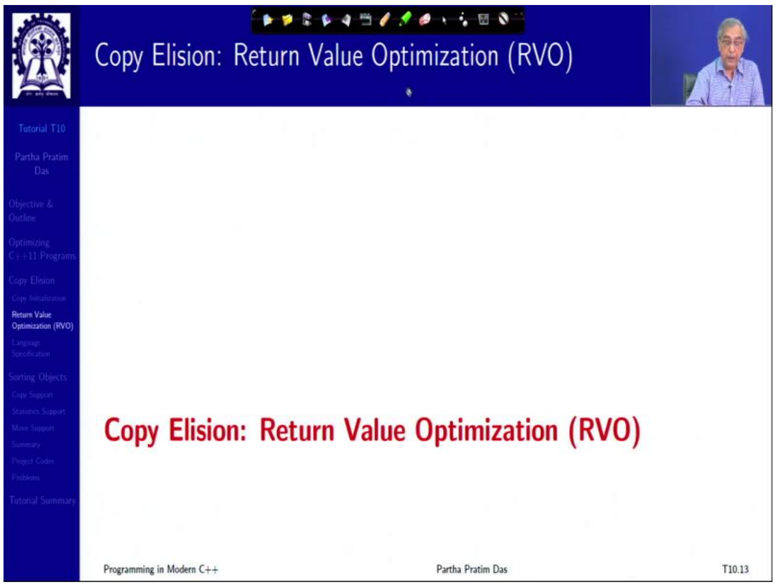
If you want to see that well, if the compiler were not aligning the copies, then how would you look like in GCC you have a nice way you have a flag by this minus f means flag, fno-elide-constructors, then it will disable the copy elision and you can see exactly as you expected it to see. So, this will invoke the parametric constructor, this will then invoke again the parametric constructor, that then it will invoke the copy constructor, you get 78.

And after having invoked that, the lvalue C(77). The temporary object which was created has no use for any further. So, that temporary object is now deleted right here. And then this is a copy construction that you have seen earlier. So, you are left with you have created 4 objects and you have deleted 1. So, you are left with 3 objects that are deleted at the end, you will be able to see that the value of n is 2 because you have invoked the copy constructor twice once here and once here.

So, in copy elision, basically what you are doing when you do not have this flag on what you are doing is when this object is constructed as an rvalue a temporary object, you do not actually copy to c2 rather what you what the compiler does for addition is compiler takes the address of the location of c2 and constructs a temporary object directly there. So, though it is a temporary object, it is not constructing and then copying it, but it is directly constructing it right here, which is actually semantically correct.

Now, that justifies why this call is not made. And why does it still need the copy constructor definition? Because it needs to know whether it is allowed to make such a copy. Though it is, it is constructing this temporary itself in the location of c2 semantically it is a copy operation. So, it wants to know whether that copy operation is allowed. But it is optimizing by directly doing that construction. That is the reason it does not need to call this particular copy constructor. So, that is the kind of copy elision optimization that happens.

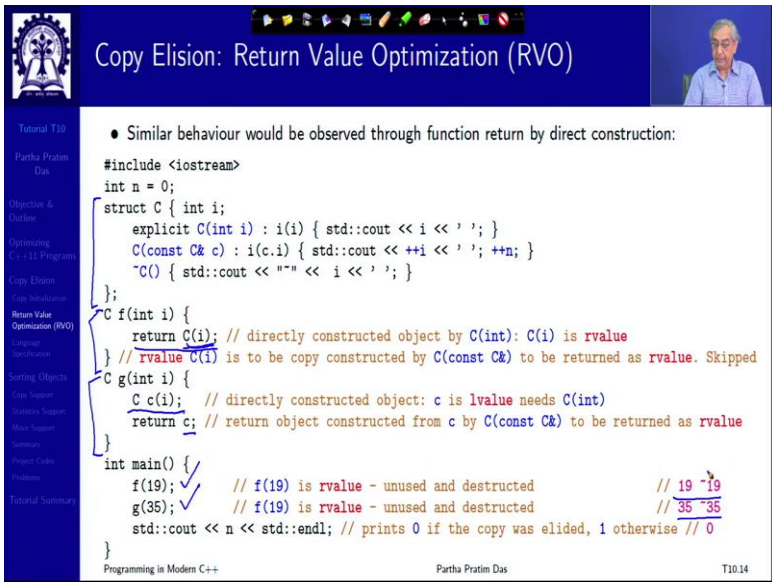
(Refer Slide Time: 14:29)



Copy Elision: Return Value Optimization (RVO)

Copy Elision: Return Value Optimization (RVO)

Programming in Modern C++ | Partha Pratim Das | T10.13



Copy Elision: Return Value Optimization (RVO)

- Similar behaviour would be observed through function return by direct construction:

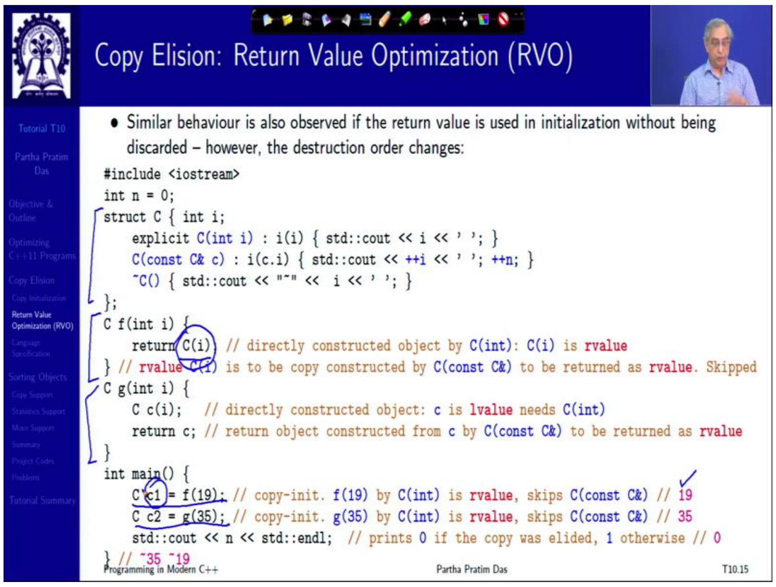
```
#include <iostream>
int n = 0;
struct C { int i;
    explicit C(int i) : i(i) { std::cout << i << ' '; }
    C(const C& c) : i(c.i) { std::cout << ++i << ' '; ++n; }
    ~C() { std::cout << "~" <<  i << ' '; }
};
C f(int i) {
    return C(i); // directly constructed object by C(int): C(i) is rvalue
} // rvalue C(i) is to be copy constructed by C(const C&) to be returned as rvalue. Skipped
C g(int i) {
    C c(i);   // directly constructed object: c is lvalue needs C(int)
    return c; // return object constructed from c by C(const C&) to be returned as rvalue
}
int main() {
    f(19);      // f(19) is rvalue - unused and destructed                // 19 ~19
    g(35);      // f(19) is rvalue - unused and destructed                // 35 ~35
    std::cout << n << std::endl; // prints 0 if the copy was elided, 1 otherwise // 0
}
```

Programming in Modern C++ | Partha Pratim Das | T10.14

Let us take another example of copy elision which is return value optimization. We have the same class, I define two functions f() and g(), f() construction object by parametric constructor and directly returns in by value. And here it constructs into a temporary into a local object, and then returns that by value, in both cases, you will expect that there will be a copy construction, there will be a direct construction by the parameter constructor and a copy constructor by return by value try to invoke these two functions, you will find that there is just the direct construction happening, no copies are been made.

Which you can argue that both of these are actually rvalues and the compiler sees that this rvalue has no use, it has not been assigned to anything or passed to some other function or a method. So, it deduces that it is enough to just construct the object, but it then does not need to actually copy it, it has no further use. So, that temporary object can be deleted. So, that is example.

(Refer Slide Time: 15:58)



Copy Elision: Return Value Optimization (RVO)

Tutorial T10

Partha Pratim Das

Objective & Outline

Optimizing C++11 Programs

Copy Elision

Copy Initialization

Return Value Optimization (RVO)

Language Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

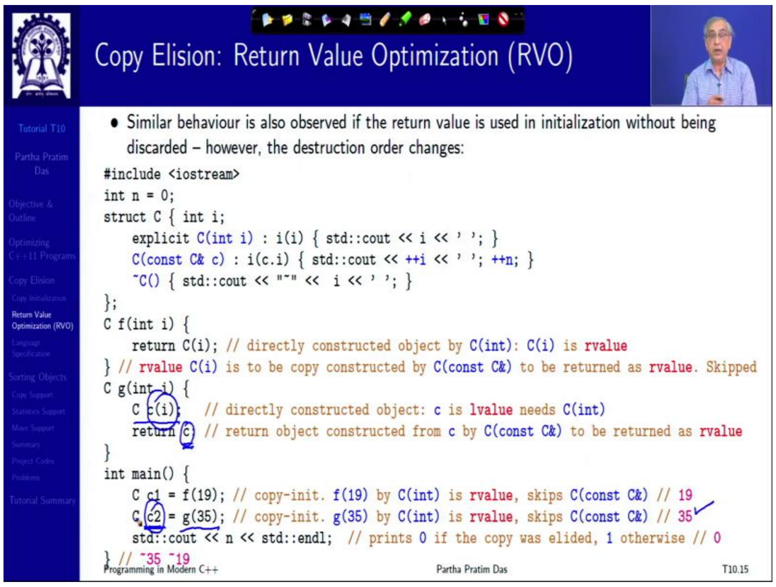
Problems

Tutorial Summary

- Similar behaviour is also observed if the return value is used in initialization without being discarded – however, the destruction order changes:

```
#include <iostream>
int n = 0;
struct C { int i;
    explicit C(int i) : i(i) { std::cout << i << ' '; }
    C(const C& c) : i(c.i) { std::cout << ++i << ' '; ++n; }
    ~C() { std::cout << "~" << i << ' '; }
};
C f(int i) {
    return C(i); // directly constructed object by C(int): C(i) is rvalue
} // rvalue C(i) is to be copy constructed by C(const C&) to be returned as rvalue. Skipped
C g(int i) {
    C c(i);   // directly constructed object: c is lvalue needs C(int)
    return c; // return object constructed from c by C(const C&) to be returned as rvalue
}
int main() {
    C c1 = f(19); // copy-init. f(19) by C(int) is rvalue, skips C(const C&) // 19
    C c2 = g(35); // copy-init. g(35) by C(int) is rvalue, skips C(const C&) // 35
    std::cout << n << std::endl;  // prints 0 if the copy was elided, 1 otherwise // 0
} // ~35 ~19
```

Programming in Modern C++ Partha Pratim Das T10.15



Copy Elision: Return Value Optimization (RVO)

Tutorial T10

Partha Pratim Das

Objective & Outline

Optimizing C++11 Programs

Copy Elision

Copy Initialization

Return Value Optimization (RVO)

Language Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

- Similar behaviour is also observed if the return value is used in initialization without being discarded – however, the destruction order changes:

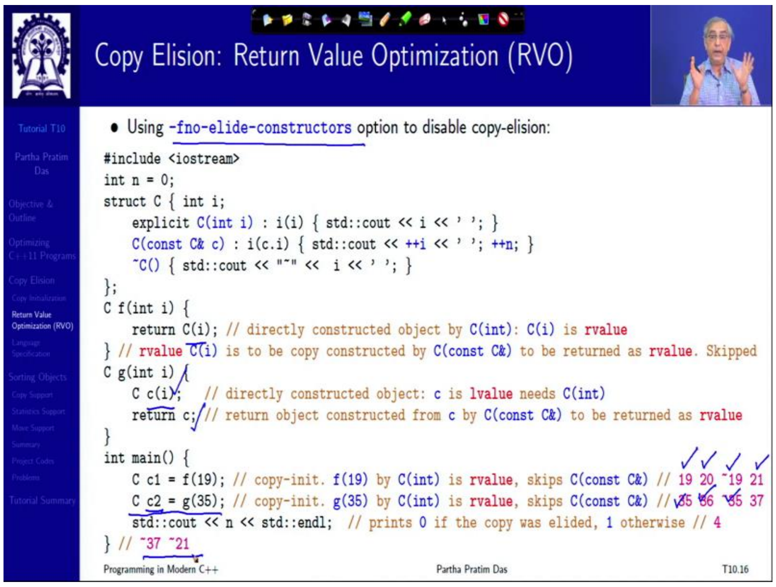
```
#include <iostream>
int n = 0;
struct C { int i;
    explicit C(int i) : i(i) { std::cout << i << ' '; }
    C(const C& c) : i(c.i) { std::cout << ++i << ' '; ++n; }
    ~C() { std::cout << "~" << i << ' '; }
};
C f(int i) {
    return C(i); // directly constructed object by C(int): C(i) is rvalue
} // rvalue C(i) is to be copy constructed by C(const C&) to be returned as rvalue. Skipped
C g(int i) {
    C c(i);   // directly constructed object: c is lvalue needs C(int)
    return c; // return object constructed from c by C(const C&) to be returned as rvalue
}
int main() {
    C c1 = f(19); // copy-init. f(19) by C(int) is rvalue, skips C(const C&) // 19
    C c2 = g(35); // copy-init. g(35) by C(int) is rvalue, skips C(const C&) // 35
    std::cout << n << std::endl;  // prints 0 if the copy was elided, 1 otherwise // 0
} // ~35 ~19
```

Programming in Modern C++ Partha Pratim Das T10.15

Now, let us just give it a little twist, give the same this is same, this is same, except that now instead of just invoking the function, I am invoking the function to initialize two objects c1 and c2, you will see the same behavior 19 constructed. So, this is constructed. But the copy constructor is not invoked. Why is the copy constructor not invoked? Because it does the copy elision that is it constructs this C(i) which is local to function f() in the address of c1 itself.

So, that it does not unnecessarily need to do the copy and then delete that temporary object, it saves on both of that. Similarly, when you do g(35), you have constructed C, so you get 35 and this return by value of this variable c is again optimized out by doing an addition that is the compiler goes to the extent to see that this c will actually get copies to c2. So, when it is constructing the C(i), it is constructing in the address value of c2 itself. Of course, now, since both of these have actually been instantiated into two local variables have made, the elisions will happen at the end.

(Refer Slide Time: 17:31)



Copy Elision: Return Value Optimization (RVO)

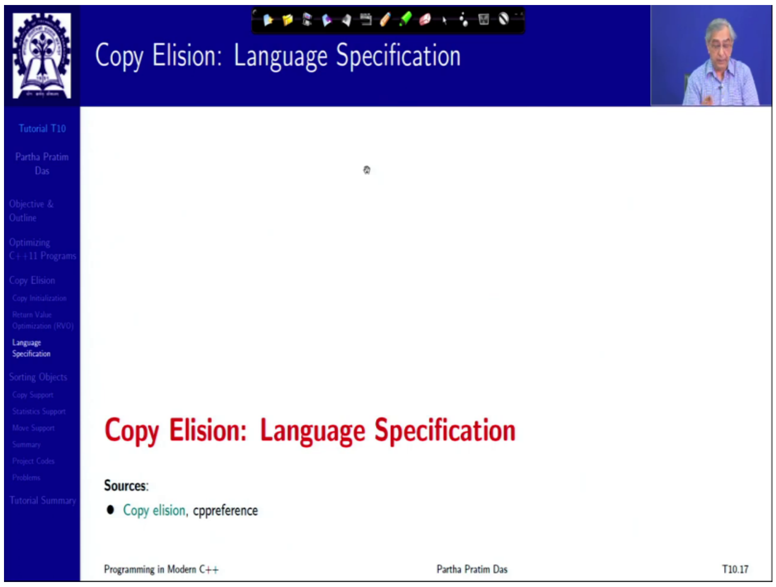
- Using -fno-elide-constructors option to disable copy-elision:

```
#include <iostream>
int n = 0;
struct C { int i;
    explicit C(int i) : i(i) { std::cout << i << ' '; }
    C(const C& c) : i(c.i) { std::cout << ++i << ' '; ++n; }
    ~C() { std::cout << "~" <<  i << ' '; }
};
C f(int i) {
    return C(i); // directly constructed object by C(int): C(i) is rvalue
} // rvalue C(i) is to be copy constructed by C(const C&) to be returned as rvalue. Skipped
C g(int i) {
    C c(i);   // directly constructed object: c is lvalue needs C(int)
    return c; // return object constructed from c by C(const C&) to be returned as rvalue
}
int main() {
    C c1 = f(19); // copy-init. f(19) by C(int) is rvalue, skips C(const C&) // 19 20 ~19 21
    C c2 = g(35); // copy-init. g(35) by C(int) is rvalue, skips C(const C&) // 35 36 ~35 37
    std::cout << n << std::endl;  // prints 0 if the copy was elided, 1 otherwise // 4
} // ~37 ~21
```

Programming in Modern C++ Partha Pratim Das T10.16

You can again use this flag to force that the compiler does not do any such copy elision if you do that, then in this case, it will first construct this object then it will copy then it will delete the temporary object and then it will again copy into c1 you know you can see that. Two copies and one elision are happening additionally here whereas when you do the other thing, you see the same behavior it first constructs then it copies return by value, then it distracts the local object c and then it copies into c2 finally, c1 and c2 are destroyed. So, you can see that copy elision and how much of you know effort the copy the additional you know task which is unnecessary can be actually handled by the copy elision optimization.

(Refer Slide Time: 18:42)



Copy Elision: Language Specification

Tutorial T10

Partha Pratim Das

Objective & Outline

Optimizing C++11 Programs

Copy Elision

Copy Initialization

Return Value Optimization (RVO)

Language Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

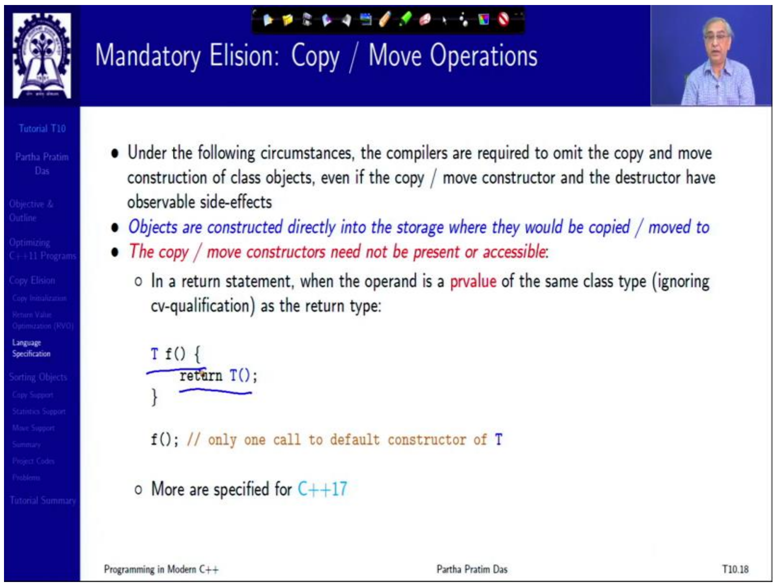
Tutorial Summary

Copy Elision: Language Specification

Sources:

- Copy elision, cppreference

Programming in Modern C++ Partha Pratim Das T10.17



Mandatory Elision: Copy / Move Operations

- Under the following circumstances, the compilers are required to omit the copy and move construction of class objects, even if the copy / move constructor and the destructor have observable side-effects
- *Objects are constructed directly into the storage where they would be copied / moved to*
- *The copy / move constructors need not be present or accessible:*
  - In a return statement, when the operand is a prvalue of the same class type (ignoring cv-qualification) as the return type:

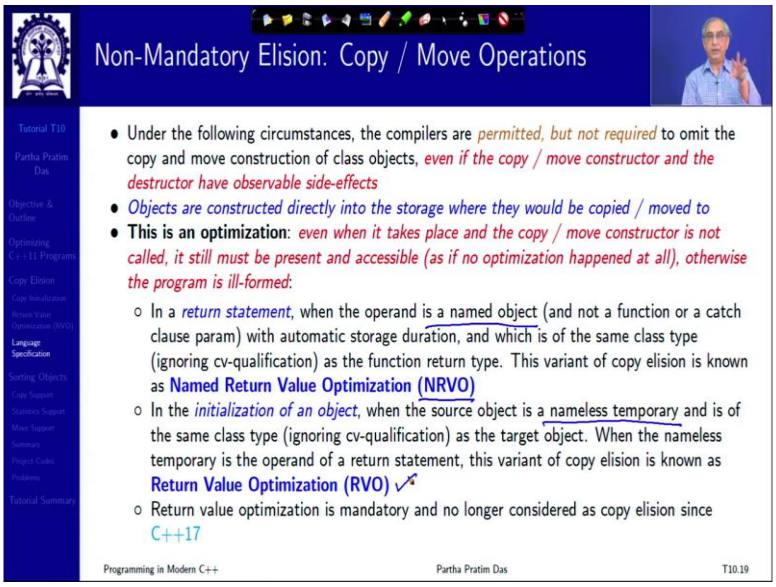
```
T f() {
    return T();
}

f(); // only one call to default constructor of T
```

  - More are specified for C++17

Programming in Modern C++ Partha Pratim Das T10.18

So, these are not just optimizations by the by the compiler, but they are language specified also for example, in C++03 itself you had this kind of optimizations given that if you do return T construction by value, then only one call to the default constructor will happen exactly as we have seen.

(Refer Slide Time: 19:13)



Non-Mandatory Elision: Copy / Move Operations

- Under the following circumstances, the compilers are *permitted, but not required* to omit the copy and move construction of class objects, *even if the copy / move constructor and the destructor have observable side-effects*
- *Objects are constructed directly into the storage where they would be copied / moved to*
- **This is an optimization**: *even when it takes place and the copy / move constructor is not called, it still must be present and accessible (as if no optimization happened at all), otherwise the program is ill-formed*:
  - In a *return statement*, when the operand is a named object (and not a function or a catch clause param) with automatic storage duration, and which is of the same class type (ignoring cv-qualification) as the function return type. This variant of copy elision is known as **Named Return Value Optimization (NRVO)**
  - In the *initialization of an object*, when the source object is a nameless temporary and is of the same class type (ignoring cv-qualification) as the target object. When the nameless temporary is the operand of a return statement, this variant of copy elision is known as **Return Value Optimization (RVO)** ✓
  - Return value optimization is mandatory and no longer considered as copy elision since C++17

Programming in Modern C++ Partha Pratim Das T10.19

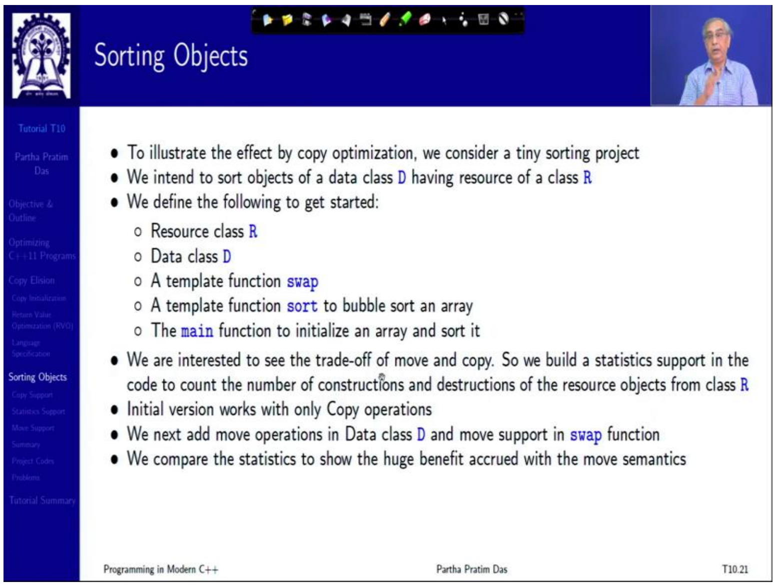
Similarly, there are it can it can in a return statement, if your operand is a named object that is lvalue then you will have NRVO Named Return Value Optimization. If it is the initialization of your object with a nameless temporary that rvalue you will have rvalue optimization. And in C++17 this rvalue optimization is no more considered a copy elision it is mandatory and it is different or optimization altogether. But C++17 we are not discussing at that length.

(Refer Slide Time: 19:56)

Sorting Objects

Sorting Objects

Programming in Modern C++ Partha Pratim Das T10.20



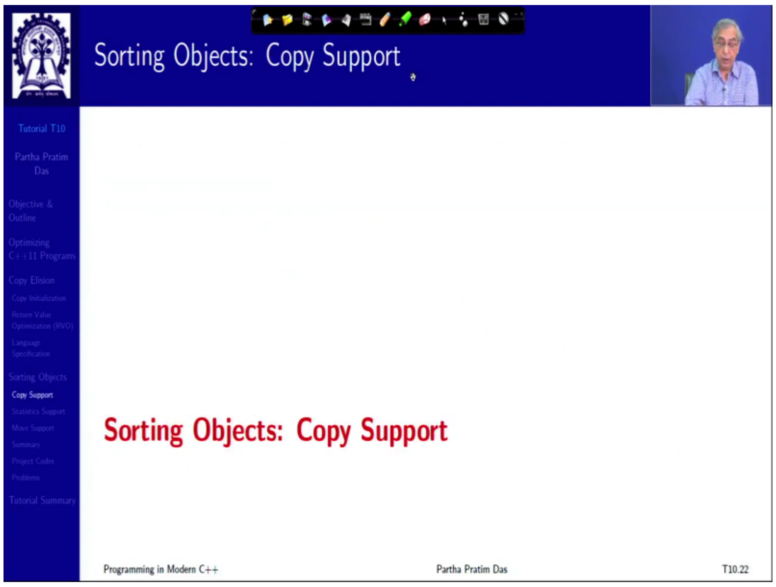
Sorting Objects

- To illustrate the effect by copy optimization, we consider a tiny sorting project
- We intend to sort objects of a data class D having resource of a class R
- We define the following to get started:
  - Resource class R
  - Data class D
  - A template function `swap`
  - A template function `sort` to bubble sort an array
  - The `main` function to initialize an array and sort it
- We are interested to see the trade-off of move and copy. So we build a statistics support in the code to count the number of constructions and destructions of the resource objects from class R
- Initial version works with only Copy operations
- We next add move operations in Data class D and move support in `swap` function
- We compare the statistics to show the huge benefit accrued with the move semantics

Programming in Modern C++ Partha Pratim Das T10.21

So, having said that now we are set to just show you a sample, you know sorting project, where you can see this benefit. So, we have a resource class R. So, it is just a dummy resource class, which will just have an integer a data class, which has a resource of the resource class R, and we have templatized swap function, a bubble sort function templatized to sort an array and a main function to give the calls. So, let us see how does that work.

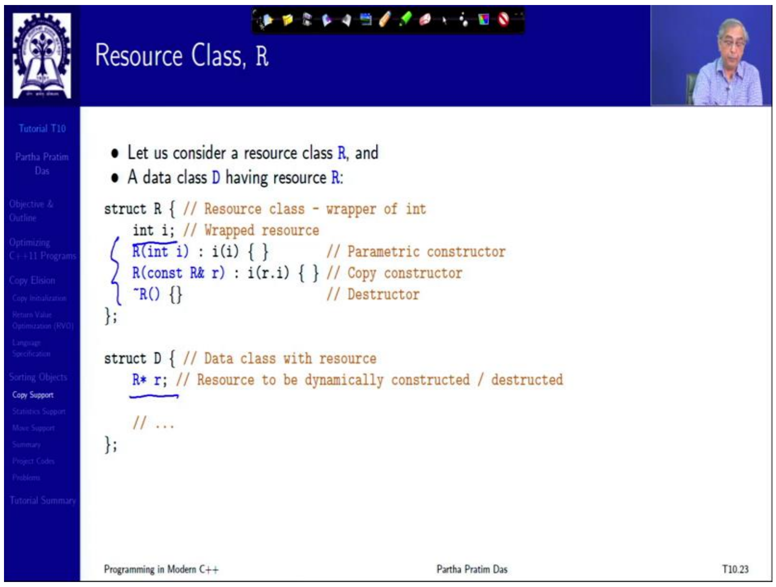
(Refer Slide Time: 20:28)



Sorting Objects: Copy Support

Sorting Objects: Copy Support

Programming in Modern C++ Partha Pratim Das T10.22



Resource Class, R

Tutorial T10
Partha Pratim Das
Objective & Outline
Optimizing C++11 Programs
Copy Elision
Copy Initialization
Return Value Optimization (RVO)
Language Specification
Sorting Objects
Copy Support
Statistics Support
Move Support
Summary
Project Codes
Problems
Tutorial Summary

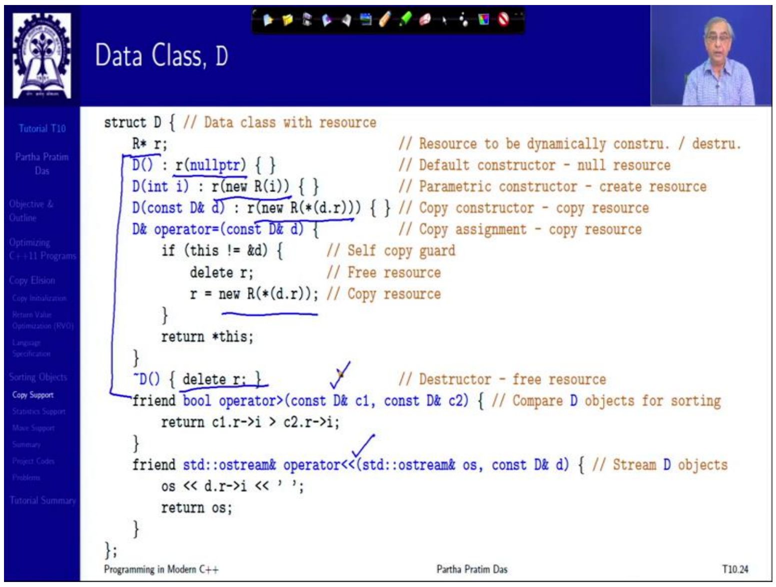
- Let us consider a resource class R, and
- A data class D having resource R:

```
struct R { // Resource class - wrapper of int
    int i; // Wrapped resource
    R(int i) : i(i) { }        // Parametric constructor
    R(const R& r) : i(r.i) { } // Copy constructor
    ~R() {}                    // Destructor
};

struct D { // Data class with resource
    R* r; // Resource to be dynamically constructed / destructed

    // ...
};
```

Programming in Modern C++ Partha Pratim Das T10.23

So, first we start with the functions with copy, I mean classes with copy support, everything is with copy support. So, this is my resource class, it has constructors and destructors as expected, and my data class has a resource of that resource class are so, it is a it holds a pointer to that resource. So, every time it has to get a proper resource object constructed to get this value.

(Refer Slide Time: 21:00)



Data Class, D

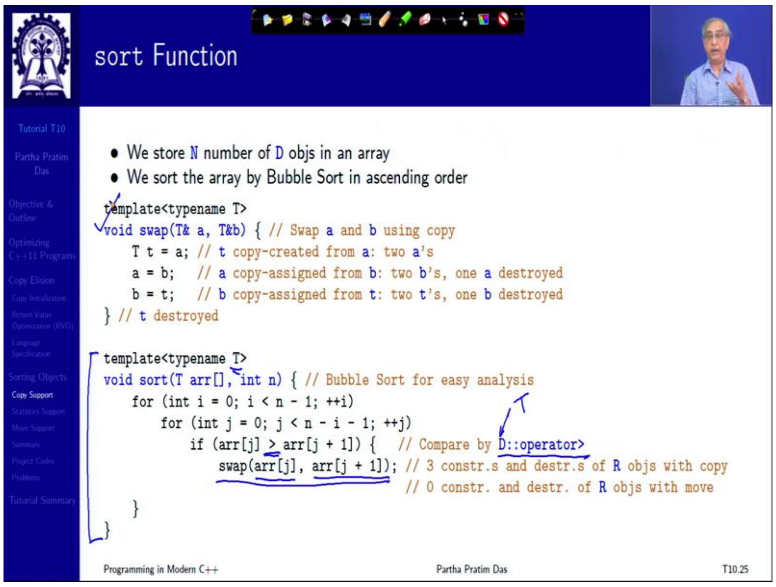
Tutorial T10
Partha Pratim Das
Objective & Outline
Optimizing C++11 Programs
Copy Elision
Copy Initialization
Return Value Optimization (RVO)
Language Specification
Sorting Objects
Copy Support
Statistics Support
Move Support
Summary
Project Codes
Problems
Tutorial Summary

```
struct D { // Data class with resource
    R* r;                                 // Resource to be dynamically constru. / destru.
    D() : r(nullptr) { }                  // Default constructor - null resource
    D(int i) : r(new R(i)) { }            // Parametric constructor - create resource
    D(const D& d) : r(new R(*(d.r))) { }  // Copy constructor - copy resource
    D& operator=(const D& d) {            // Copy assignment - copy resource
        if (this != &d) {      // Self copy guard
            delete r;          // Free resource
            r = new R(*(d.r)); // Copy resource
        }
        return *this;
    }
    ~D() { delete r; }                    // Destructor - free resource
    friend bool operator>(const D& c1, const D& c2) { // Compare D objects for sorting
        return c1.r->i > c2.r->i;
    }
    friend std::ostream& operator<<(std::ostream& os, const D& d) { // Stream D objects
        os << d.r->i << ' ';
        return os;
    }
};
```

Programming in Modern C++ Partha Pratim Das T10.24

So, in the data class it has appointed to this so, it has all its usual operations of you know default will set it to null. Otherwise, if it is parameterized, then it will construct an object with value R->i will allocate for that, and put that pointer in case of a copy construction, it will copy the value of the resource from the source object and construct a new object through that is a deep copy will get done, this is exactly what happens in the copy assignment operator delete will have to release this resource. So, the standard scenario and then we have a we have friend operator out stream function to just you know, do our understanding of what is going on. And I have provided a operator greater than comparison operator of two D objects so that I can do sorting based on that.

(Refer Slide Time: 22:15)



sort Function

- We store N number of D objs in an array
- We sort the array by Bubble Sort in ascending order

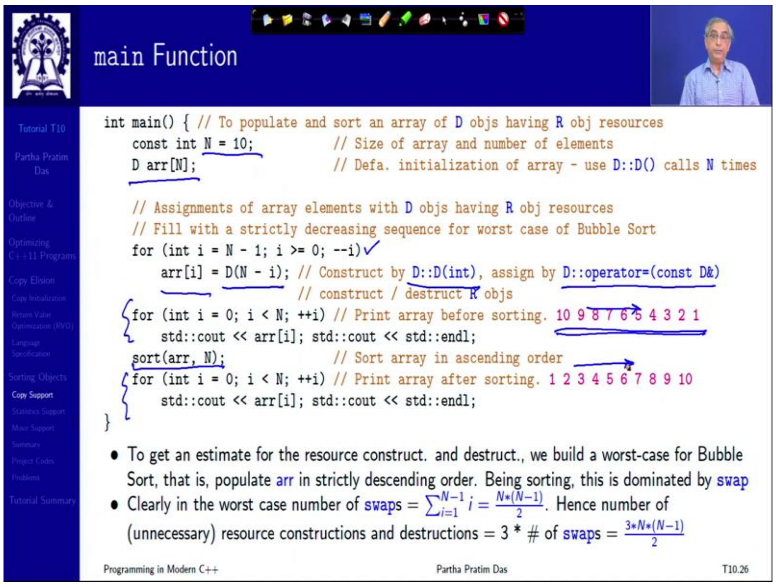
```
template<typename T>
void swap(T& a, T&b) { // Swap a and b using copy
    T t = a; // t copy-created from a: two a's
    a = b;   // a copy-assigned from b: two b's, one a destroyed
    b = t;   // b copy-assigned from t: two t's, one b destroyed
} // t destroyed

template<typename T>
void sort(T arr[], int n) { // Bubble Sort for easy analysis
    for (int i = 0; i < n - 1; ++i)
        for (int j = 0; j < n - i - 1; ++j)
            if (arr[j] > arr[j + 1]) {   // Compare by D::operator>
                swap(arr[j], arr[j + 1]); // 3 constr.s and destr.s of R objs with copy
                                          // 0 constr. and destr. of R objs with move
        }
}
```

Programming in Modern C++ Partha Pratim Das T10.25

So, now, to do the sort function, what do we need, this is a this is a bubble sort function it issues that the data values are of type T, and the rest of it is Bubble sort function which is which, you know, so I am not going to explain the reason I have chosen Bubble sort is it is easy for analysis. And I am going to construct a worst case for Bubble sort that is elements are sorted in a certain order and I want to sort them in in the opposite order.

So, this is a comparison which will use the comparison operator actually I should write here not D I should write here T, T::operator greater than and that will need to swap two objects. Now, when you want to swap these two objects, then naturally I make use of this swap function, which is standard templatized swap function which as you know, does multiple 3 copies. So, 3 constructors and destructors of the resource object with copy will be will have to be done with move the scenario will change.

(Refer Slide Time: 23:30)



main Function

```
int main() { // To populate and sort an array of D objs having R obj resources
    const int N = 10;           // Size of array and number of elements
    D arr[N];                   // Defa. initialization of array - use D::D() calls N times

    // Assignments of array elements with D objs having R obj resources
    // Fill with a strictly decreasing sequence for worst case of Bubble Sort
    for (int i = N - 1; i >= 0; --i)
        arr[i] = D(N - i); // Construct by D::D(int), assign by D::operator=(const D&)
                           // construct / destruct R objs
    for (int i = 0; i < N; ++i) // Print array before sorting. 10 9 8 7 6 5 4 3 2 1
        std::cout << arr[i]; std::cout << std::endl;
    sort(arr, N);               // Sort array in ascending order
    for (int i = 0; i < N; ++i) // Print array after sorting. 1 2 3 4 5 6 7 8 9 10
        std::cout << arr[i]; std::cout << std::endl;
}
```

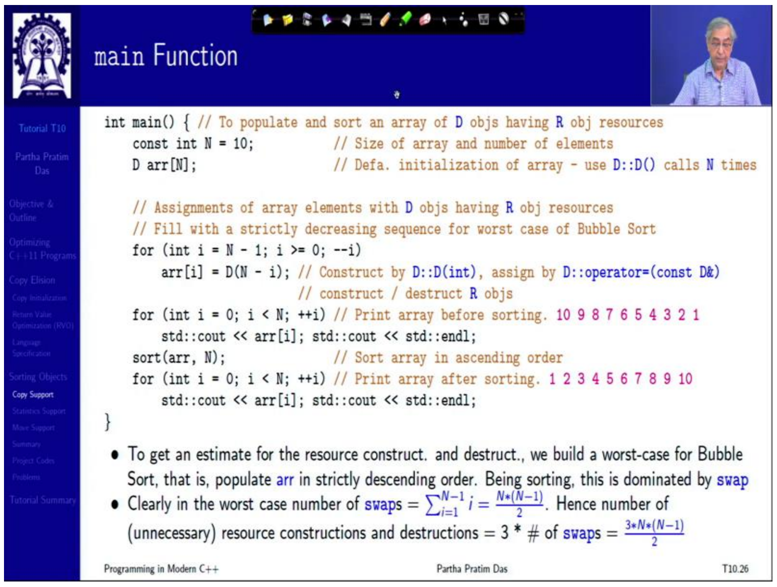
- To get an estimate for the resource construct. and destruct., we build a worst-case for Bubble Sort, that is, populate arr in strictly descending order. Being sorting, this is dominated by swap
- Clearly in the worst case number of swaps $= \sum_{i=1}^{N-1} i = \frac{N*(N-1)}{2}$. Hence number of (unnecessary) resource constructions and destructions $= 3 * \#$ of swaps $= \frac{3*N*(N-1)}{2}$

Programming in Modern C++ Partha Pratim Das T10.26

Now, let us see what happens in the main function I declare an array this is the number of elements size of the array and the number of elements I assume them to be same, I declare an array which means that the default constructor of the D class is used to fill the array with D objects having null resource pointers, then I fill up this array using which with D(N – i) objects, so you can see that I am actually filling them up in the increasing order in the decreasing order.

So, 10-1, so this is the order in which that will be filled up, because I said I want to create a worst case. So, every time I do this, I will construct this D(N – i) by the parameterized constructor, then I will have it have to do the assignment using the copy constructor to construct and deconstruct the R objects as they exist. This is just to print the array so that they can see what is in then I call the sort function. The template is sort function and called sort output now and sort what does the sort function do? It tries to sort them in the increasing order, but this is strictly in the decreasing order. And I want to make them into increasing order. So, I want to do just the opposite. So, which means that if I go back to the sort function, it will be easier to see.

(Refer Slide Time: 25:11)

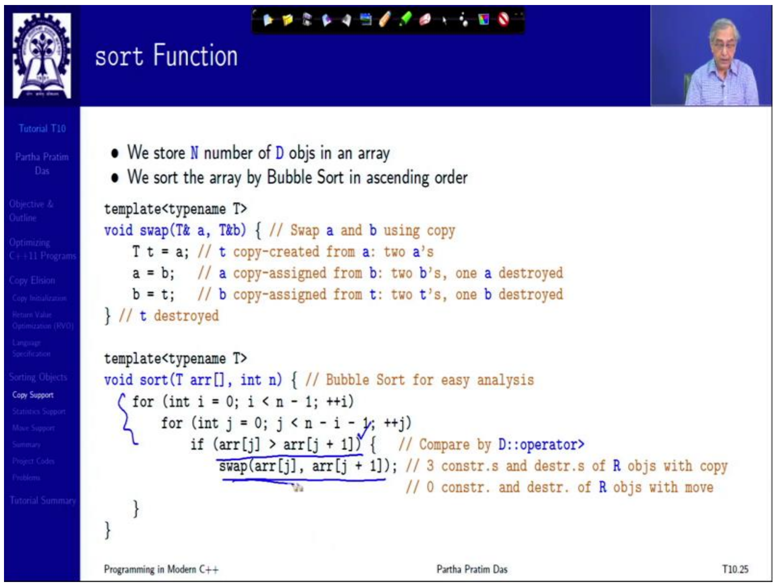


main Function

```
int main() { // To populate and sort an array of D objs having R obj resources
    const int N = 10;           // Size of array and number of elements
    D arr[N];                   // Defa. initialization of array - use D::D() calls N times

    // Assignments of array elements with D objs having R obj resources
    // Fill with a strictly decreasing sequence for worst case of Bubble Sort
    for (int i = N - 1; i >= 0; --i)
        arr[i] = D(N - i); // Construct by D::D(int), assign by D::operator=(const D&)
                           // construct / destruct R objs
    for (int i = 0; i < N; ++i) // Print array before sorting. 10 9 8 7 6 5 4 3 2 1
        std::cout << arr[i]; std::cout << std::endl;
    sort(arr, N);               // Sort array in ascending order
    for (int i = 0; i < N; ++i) // Print array after sorting. 1 2 3 4 5 6 7 8 9 10
        std::cout << arr[i]; std::cout << std::endl;
}
```

- To get an estimate for the resource construct. and destruct., we build a worst-case for Bubble Sort, that is, populate arr in strictly descending order. Being sorting, this is dominated by swap
- Clearly in the worst case number of swaps = $\sum_{i=1}^{N-1} i = \frac{N*(N-1)}{2}$. Hence number of (unnecessary) resource constructions and destructions = 3 * # of swaps = $\frac{3*N*(N-1)}{2}$

Programming in Modern C++ Partha Pratim Das T10.26



sort Function

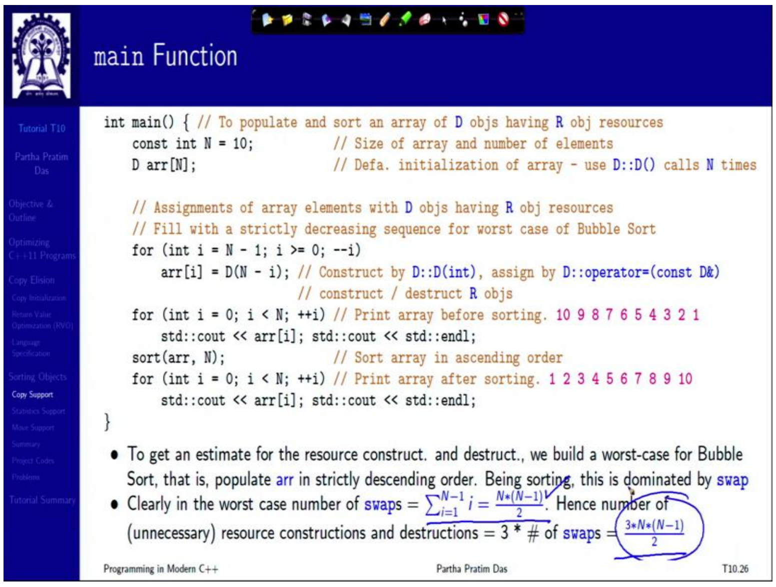
- We store N number of D objs in an array
- We sort the array by Bubble Sort in ascending order

```
template<typename T>
void swap(T& a, T&b) { // Swap a and b using copy
    T t = a; // t copy-created from a: two a's
    a = b;   // a copy-assigned from b: two b's, one a destroyed
    b = t;   // b copy-assigned from t: two t's, one b destroyed
} // t destroyed

template<typename T>
void sort(T arr[], int n) { // Bubble Sort for easy analysis
    for (int i = 0; i < n - 1; ++i)
        for (int j = 0; j < n - i - 1; ++j)
            if (arr[j] > arr[j + 1]) {   // Compare by D::operator>
                swap(arr[j], arr[j + 1]); // 3 constr.s and destr.s of R objs with copy
                                          // 0 constr. and destr. of R objs with move
    }
}
```

Programming in Modern C++ Partha Pratim Das T10.25

If you look into this sort function, then you will see that since it is strictly in the decreasing order, and I want to make it, so, every time I do this comparison, this is going to succeed that is, that is the reason I am using Bubble sort. So, very easy, you know, worst case. So, since they are the numbers are decreasing, and I want them increasing, so, every time I compare the consecutive pairs, they are in the wrong order. So, this will always be true, which means that as many times as these loops go on this swap function will be called.

(Refer Slide Time: 25:50)



main Function

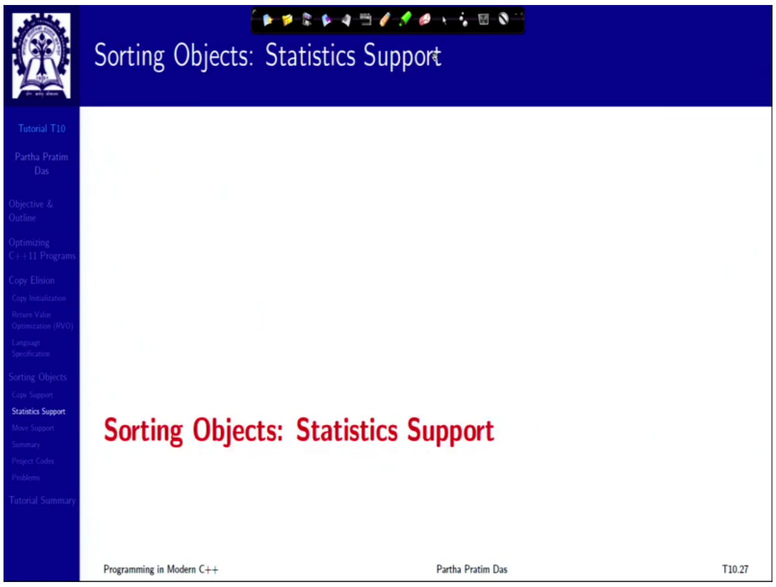
```
int main() { // To populate and sort an array of D objs having R obj resources
    const int N = 10;           // Size of array and number of elements
    D arr[N];                   // Defa. initialization of array - use D::D() calls N times

    // Assignments of array elements with D objs having R obj resources
    // Fill with a strictly decreasing sequence for worst case of Bubble Sort
    for (int i = N - 1; i >= 0; --i)
        arr[i] = D(N - i); // Construct by D::D(int), assign by D::operator=(const D&)
                           // construct / destruct R objs
    for (int i = 0; i < N; ++i) // Print array before sorting. 10 9 8 7 6 5 4 3 2 1
        std::cout << arr[i]; std::cout << std::endl;
    sort(arr, N);               // Sort array in ascending order
    for (int i = 0; i < N; ++i) // Print array after sorting. 1 2 3 4 5 6 7 8 9 10
        std::cout << arr[i]; std::cout << std::endl;
}
```

- To get an estimate for the resource construct. and destruct., we build a worst-case for Bubble Sort, that is, populate arr in strictly descending order. Being sorting, this is dominated by swap
- Clearly in the worst case number of swaps = $\underline{\sum_{i=1}^{N-1} i = \frac{N*(N-1)}{2}}$. Hence number of (unnecessary) resource constructions and destructions = 3 * # of swaps = $\boxed{\frac{3*N*(N-1)}{2}}$

Programming in Modern C++ Partha Pratim Das T10.26

So, if I put a little algorithm analysis into this, then this basically means that, in every loop, it will depend on the value of i, how many times it happens and it goes from 1 to N - 1, hence, the total N the number of times the swapping will be done is N to N - 1 by 2. So, in every swap, I have 3 assignments, copy assignments, so what will happen is I will have 3 resource objects created and free resource object destruct. So, this is, this is the amount of construction destruction of the resource objects that I am going to do.

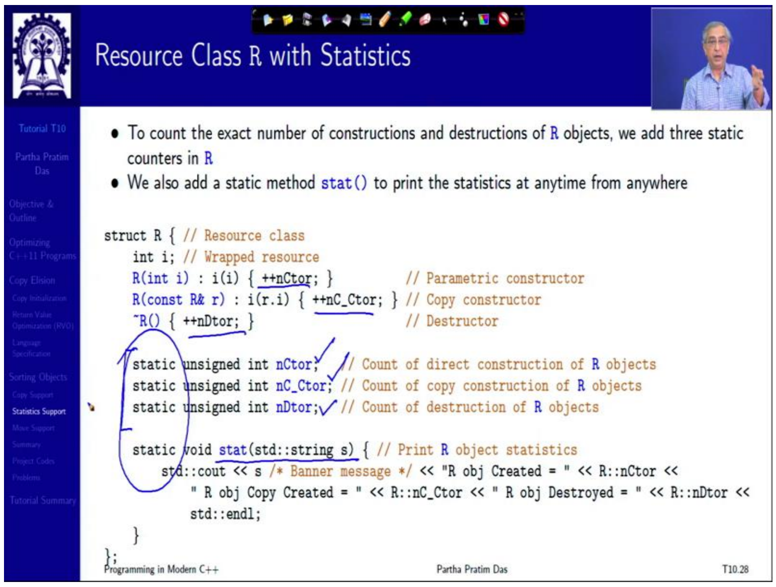
(Refer Slide Time: 26:39)



Sorting Objects: Statistics Support

Sorting Objects: Statistics Support

Programming in Modern C++ Partha Pratim Das T10.27



Resource Class R with Statistics

Tutorial T10

Partha Pratim Das

Objective & Outline

Optimizing C++11 Programs

Copy Elision

Copy Initialization

Return Value Optimization (RVO)

Language Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

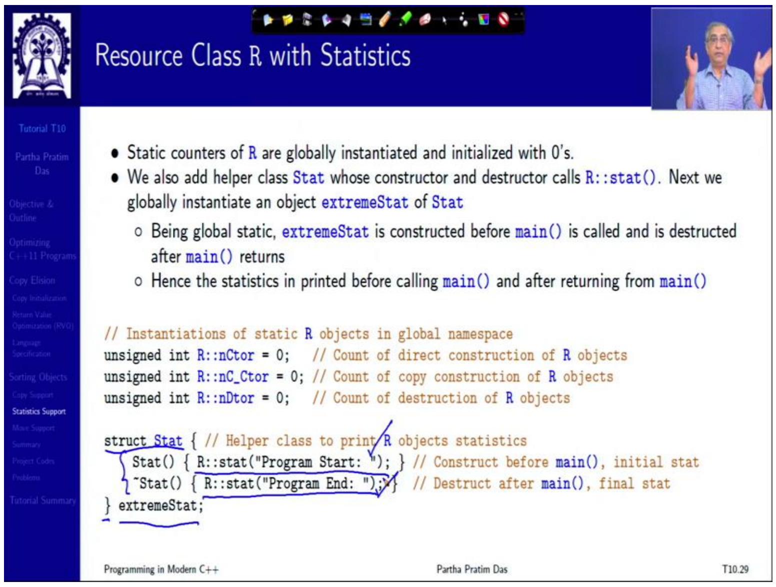
- To count the exact number of constructions and destructions of R objects, we add three static counters in R
- We also add a static method stat() to print the statistics at anytime from anywhere

```
struct R { // Resource class
    int i; // Wrapped resource
    R(int i) : i(i) { ++nCtor; }            // Parametric constructor
    R(const R& r) : i(r.i) { ++nC_Ctor; } // Copy constructor
    ~R() { ++nDtor; }                       // Destructor

    static unsigned int nCtor;   // Count of direct construction of R objects
    static unsigned int nC_Ctor; // Count of copy construction of R objects
    static unsigned int nDtor;   // Count of destruction of R objects

    static void stat(std::string s) { // Print R object statistics
        std::cout << s /* Banner message */ << "R obj Created = " << R::nCtor <<
            " R obj Copy Created = " << R::nC_Ctor << " R obj Destroyed = " << R::nDtor <<
            std::endl;
    }
};
```

Programming in Modern C++ Partha Pratim Das T10.28

Now, to this was analytical, so, if we if I really want to see the number that is happening, I need to provide some support for the statistics. So, that you can these are techniques you can do in in any, any class to if you are working on the performance. So, I put some counters accounted to count the number of times direct constructed reasons number of times a copy constructor is used and the number of times the destructor is used and the respective counters are incremented in the respective function calls. And then I have a stat function, which basically prints the values of these counters at any point of time, all of these are static, because they are not bound to any particular R object, but should be usable anywhere.

(Refer Slide Time: 27:30)



Resource Class R with Statistics

Tutorial T10

Partha Pratim Das

Objective & Outline

Optimizing C++11 Programs

Copy Elision

Copy Initialization

Return Value Optimization (RVO)

Language Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

- Static counters of R are globally instantiated and initialized with 0's.
- We also add helper class Stat whose constructor and destructor calls R::stat(). Next we globally instantiate an object extremeStat of Stat
  - Being global static, extremeStat is constructed before main() is called and is destructed after main() returns
  - Hence the statistics in printed before calling main() and after returning from main()

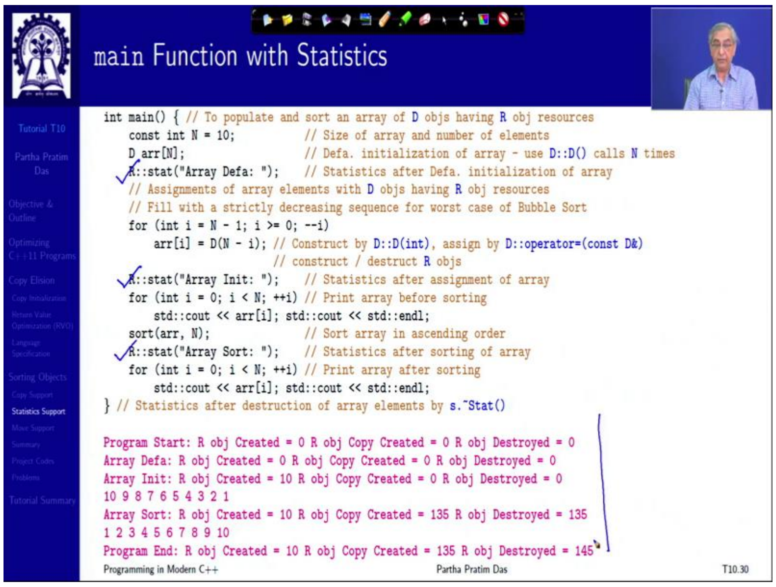
```
// Instantiations of static R objects in global namespace
unsigned int R::nCtor = 0;   // Count of direct construction of R objects
unsigned int R::nC_Ctor = 0; // Count of copy construction of R objects
unsigned int R::nDtor = 0;   // Count of destruction of R objects

struct Stat { // Helper class to print R objects statistics
    Stat() { R::stat("Program Start: "); } // Construct before main(), initial stat
    ~Stat() { R::stat("Program End: "); }  // Destruct after main(), final stat
} extremeStat;
```

Programming in Modern C++ Partha Pratim Das T10.29

Now, since I have these static members, so I will need to instantiate them in the global space, and I instantiate and initialize them with 0. In addition, I do something, interesting is I define another class struct in whose constructor I call this stat function, say with program start and the stat function again in the destructor with program end and define a variable of instantiate it in the global.

So, what will happen, this is a static object, extreme stat will become a static object. Therefore, as you know, static objects must be constructed before main starts, and they are distracted after main ends. So, this function will get called before main starts and this function will get called after main returns. So, I will be able to get the first and the final statistics. So, that is just a just a nice technique to create statistics.

(Refer Slide Time: 28:36)

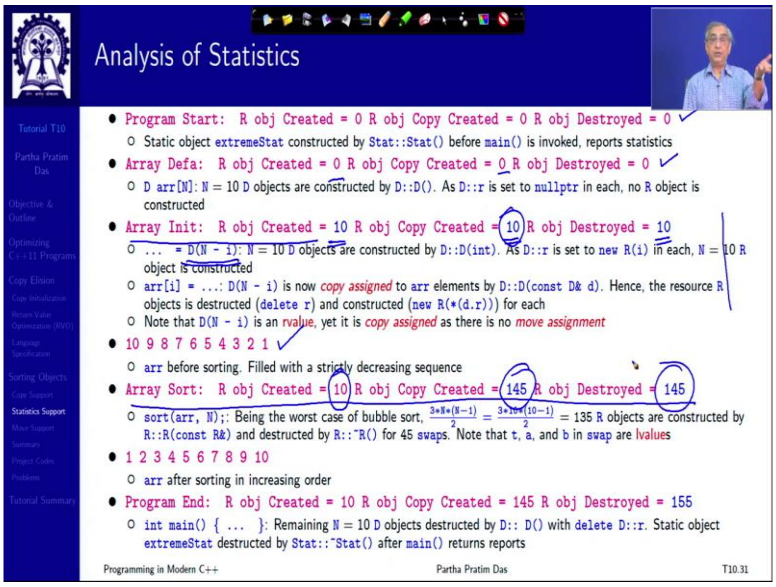


main Function with Statistics

```
int main() { // To populate and sort an array of D objs having R obj resources
    const int N = 10;           // Size of array and number of elements
    D arr[N];                   // Defa. initialization of array - use D::D() calls N times
    R::stat("Array Defa: ");    // Statistics after Defa. initialization of array
    // Assignments of array elements with D objs having R obj resources
    // Fill with a strictly decreasing sequence for worst case of Bubble Sort
    for (int i = N - 1; i >= 0; --i)
        arr[i] = D(N - i); // Construct by D::D(int), assign by D::operator=(const D&)
                           // construct / destruct R objs
    R::stat("Array Init: ");    // Statistics after assignment of array
    for (int i = 0; i < N; ++i) // Print array before sorting
        std::cout << arr[i]; std::cout << std::endl;
    sort(arr, N);               // Sort array in ascending order
    R::stat("Array Sort: ");    // Statistics after sorting of array
    for (int i = 0; i < N; ++i) // Print array after sorting
        std::cout << arr[i]; std::cout << std::endl;
} // Statistics after destruction of array elements by s.~Stat()
```

```
Program Start: R obj Created = 0 R obj Copy Created = 0 R obj Destroyed = 0
Array Defa: R obj Created = 0 R obj Copy Created = 0 R obj Destroyed = 0
Array Init: R obj Created = 10 R obj Copy Created = 0 R obj Destroyed = 0
10 9 8 7 6 5 4 3 2 1
Array Sort: R obj Created = 10 R obj Copy Created = 135 R obj Destroyed = 135
1 2 3 4 5 6 7 8 9 10
Program End: R obj Created = 10 R obj Copy Created = 135 R obj Destroyed = 145
```

Programming in Modern C++ Partha Pratim Das T10.30



Analysis of Statistics

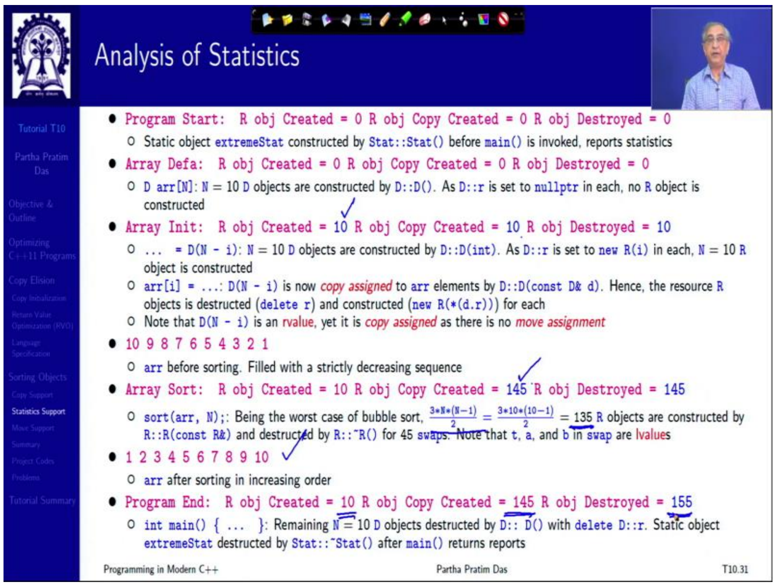
- Program Start: R obj Created = 0 R obj Copy Created = 0 R obj Destroyed = 0 ✓
  - Static object extremeStat constructed by Stat::Stat() before main() is invoked, reports statistics
- Array Defa: R obj Created = 0 R obj Copy Created = 0 R obj Destroyed = 0 ✓
  - D arr[N]: N = 10 D objects are constructed by D::D(). As D::r is set to nullptr in each, no R object is constructed
- Array Init: R obj Created = 10 R obj Copy Created = 10 R obj Destroyed = 10
  - ... = D(N - i): N = 10 D objects are constructed by D::D(int). As D::r is set to new R(i) in each, N = 10 R object is constructed
  - arr[i] = ...: D(N - i) is now *copy assigned* to arr elements by D::D(const D& d). Hence, the resource R objects is destructed (delete r) and constructed (new R(*(d.r))) for each
  - Note that D(N - i) is an rvalue, yet it is *copy assigned* as there is no *move assignment*
- 10 9 8 7 6 5 4 3 2 1 ✓
  - arr before sorting. Filled with a strictly decreasing sequence
- Array Sort: R obj Created = 10 R obj Copy Created = 145 R obj Destroyed = 145
  - sort(arr, N);: Being the worst case of bubble sort, $\frac{3*N*(N-1)}{2} = \frac{3*10*(10-1)}{2}$ = 135 R objects are constructed by R::R(const R&) and destructed by R::~R() for 45 swaps. Note that t, a, and b in swap are lvalues
- 1 2 3 4 5 6 7 8 9 10
  - arr after sorting in increasing order
- Program End: R obj Created = 10 R obj Copy Created = 145 R obj Destroyed = 155
  - int main() { ... }: Remaining N = 10 D objects destructed by D:: D() with delete D::r. Static object extremeStat destructed by Stat::~Stat() after main() returns reports

Programming in Modern C++ Partha Pratim Das T10.31

So, I do the same thing in the main after the default initialization, I print statistics after filling that arr[i], statistics after sorting, I print statistics and I get all of these. So, if you just recall what will this mean this will mean that initially everything is 0, because no objects are created so far, then after the default, again, no R objects are created, because the D objects initialize the resource as null pointer.

Then after I initialize the array by this every time, I construct a D object with parameter in R object will get initialized will get created so I have 10 creations of these and then they will be copied 10 times to arr. So, there are 10 copies and those temporaries will be deleted. So, there are 10 deletions. So, if you look into that and then I have this, then have the sort. And in sort you can see that creation still remains to be 10. But I have 145 copies created instead of 10. So, 135 more copies have been created these are because of the swap and those are created and destroyed. So, just see how west fall it is.

(Refer Slide Time: 30:14)



Analysis of Statistics

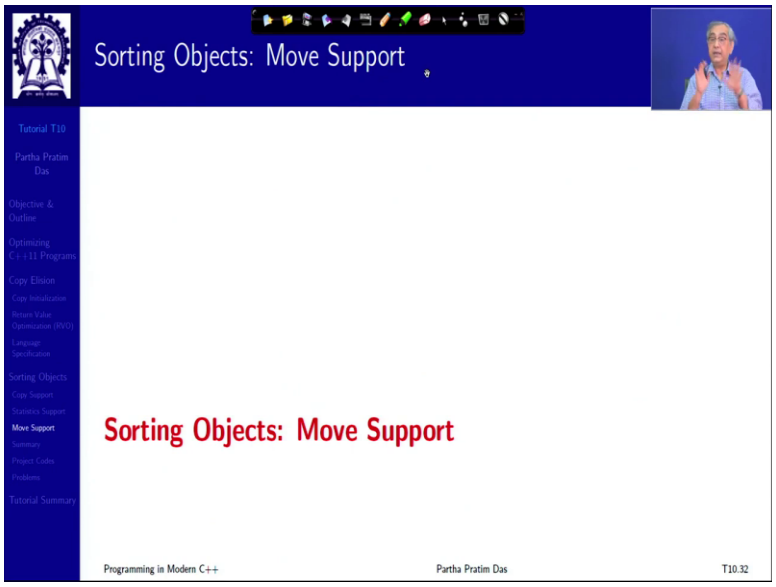
- Program Start: R obj Created = 0 R obj Copy Created = 0 R obj Destroyed = 0
  - Static object `extremeStat` constructed by `Stat::Stat()` before `main()` is invoked, reports statistics
- Array Defa: R obj Created = 0 R obj Copy Created = 0 R obj Destroyed = 0
  - `D arr[N]`: N = 10 D objects are constructed by `D::D()`. As `D::r` is set to `nullptr` in each, no R object is constructed
- Array Init: R obj Created = 10 R obj Copy Created = 10 R obj Destroyed = 10
  - `... = D(N - i)`: N = 10 D objects are constructed by `D::D(int)`. As `D::r` is set to `new R(i)` in each, N = 10 R object is constructed
  - `arr[i] = ...: D(N - i)` is now *copy assigned* to `arr` elements by `D::D(const D& d)`. Hence, the resource R objects is destructed (`delete r`) and constructed (`new R(*(d.r))`) for each
  - Note that `D(N - i)` is an rvalue, yet it is *copy assigned* as there is no *move assignment*
- 10 9 8 7 6 5 4 3 2 1
  - `arr` before sorting. Filled with a strictly decreasing sequence
- Array Sort: R obj Created = 10 R obj Copy Created = 145 R obj Destroyed = 145
  - `sort(arr, N);`: Being the worst case of bubble sort, $\frac{3*N*(N-1)}{2} = \frac{3*10*(10-1)}{2} = 135$ R objects are constructed by `R::R(const R&)` and destructed by `R::~R()` for 45 swaps. Note that t, a, and b in `swap` are lvalues
- 1 2 3 4 5 6 7 8 9 10
  - `arr` after sorting in increasing order
- Program End: R obj Created = 10 R obj Copy Created = 145 R obj Destroyed = 155
  - `int main() { ... }`: Remaining N = 10 D objects destructed by `D::~D()` with `delete D::r`. Static object `extremeStat` destructed by `Stat::~Stat()` after `main()` returns reports

Programming in Modern C++ Partha Pratim Das T10.31

And if you can tallied with the analytical formula, we have just derived you will find that this is 135 which tallies with the fact that 10 plus 135 is 145 then you have the adding and finally, you have the total copies created at 145 direct creation is 10 from here and therefore, the 10 objects in the array that existed when the main ends those will get destructed. So, you will have 155 destructions.

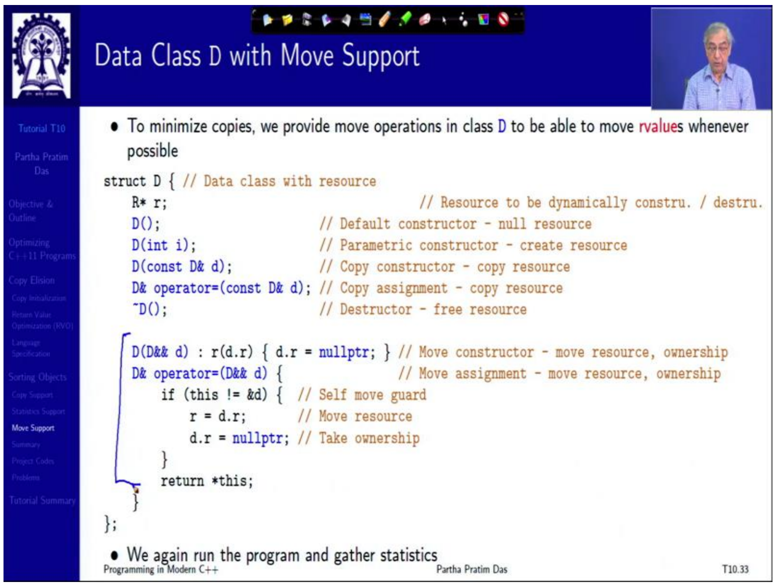
So, you can see that you actually needed 10 objects to be created and 10 objects to be distracted, but you have along with that you have created 145 objects Additionally, R objects additionally and 145 R objects you have destructed.

(Refer Slide Time: 31:04)



Sorting Objects: Move Support

Sorting Objects: Move Support

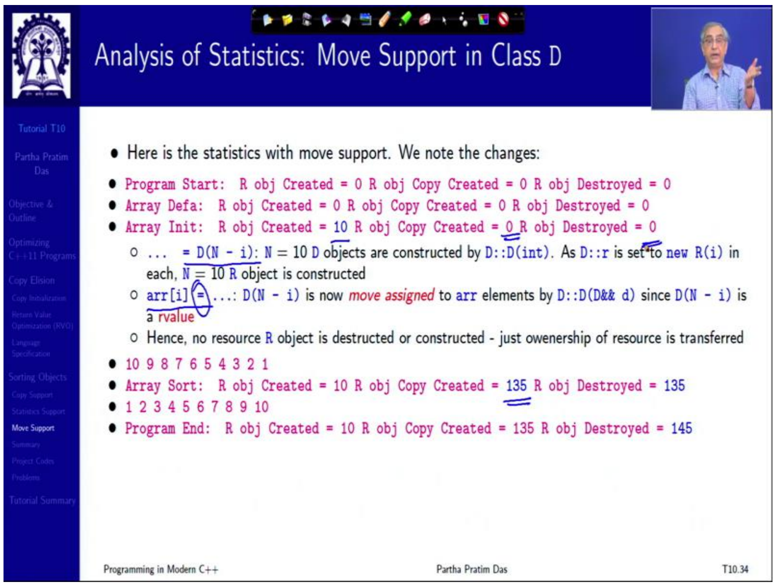


Data Class D with Move Support

- To minimize copies, we provide move operations in class D to be able to move rvalues whenever possible

```
struct D { // Data class with resource
    R* r;                                  // Resource to be dynamically constru. / destru.
    D();                    // Default constructor - null resource
    D(int i);               // Parametric constructor - create resource
    D(const D& d);          // Copy constructor - copy resource
    D& operator=(const D& d); // Copy assignment - copy resource
    ~D();                   // Destructor - free resource

    D(D&& d) : r(d.r) { d.r = nullptr; } // Move constructor - move resource, ownership
    D& operator=(D&& d) {              // Move assignment - move resource, ownership
        if (this != &d) {  // Self move guard
            r = d.r;       // Move resource
            d.r = nullptr; // Take ownership
        }
        return *this;
    }
};
```

- We again run the program and gather statistics



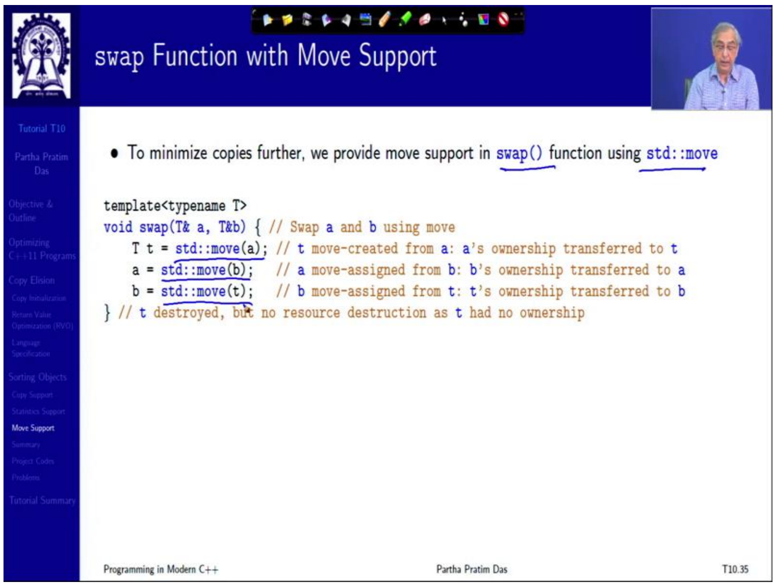
Analysis of Statistics: Move Support in Class D

- Here is the statistics with move support. We note the changes:
- `Program Start:  R obj Created = 0 R obj Copy Created = 0 R obj Destroyed = 0`
- `Array Defa:  R obj Created = 0 R obj Copy Created = 0 R obj Destroyed = 0`
- `Array Init:  R obj Created = 10 R obj Copy Created = 0 R obj Destroyed = 0`
  - `...  = D(N - i)`: N = 10 D objects are constructed by `D::D(int)`. As `D::r` is set to `new R(i)` in each, N = 10 R object is constructed
  - `arr[i] = ...`: `D(N - i)` is now *move assigned* to `arr` elements by `D::D(D&& d)` since `D(N - i)` is a rvalue
  - Hence, no resource R object is destructed or constructed - just owenership of resource is transferred
- `10 9 8 7 6 5 4 3 2 1`
- `Array Sort:  R obj Created = 10 R obj Copy Created = 135 R obj Destroyed = 135`
- `1 2 3 4 5 6 7 8 9 10`
- `Program End:  R obj Created = 10 R obj Copy Created = 135 R obj Destroyed = 145`

So, that is the basic problem with temporaries. So, how do you get rid of this you know that you can give a move support. So, you say I will not copy, if not required, I will not copy the D objects, I will move them. So, I have a move constructor and a move assignment operator added now. So, how will that impact, it will impact in the sense that if I run the program again, when I am doing arr[i] = D(N – i), D(N – i) is a constructed object and D(N – i) is an rvalue.

So, in this assignment, it is not the copy assignment which will bind it is a move assignment that will bind and in the move assignment, I will simply move the resource I do not need to copy the resource. So, after 10 when I do this, I will have only 135 copies made instead of I was earlier having 145, I will have 135 only. Because these 10 copies that were required here copies and destruction that were required here is saved. So, some benefit has been obtained.

(Refer Slide Time: 32:28)



swap Function with Move Support

- To minimize copies further, we provide move support in swap() function using std::move

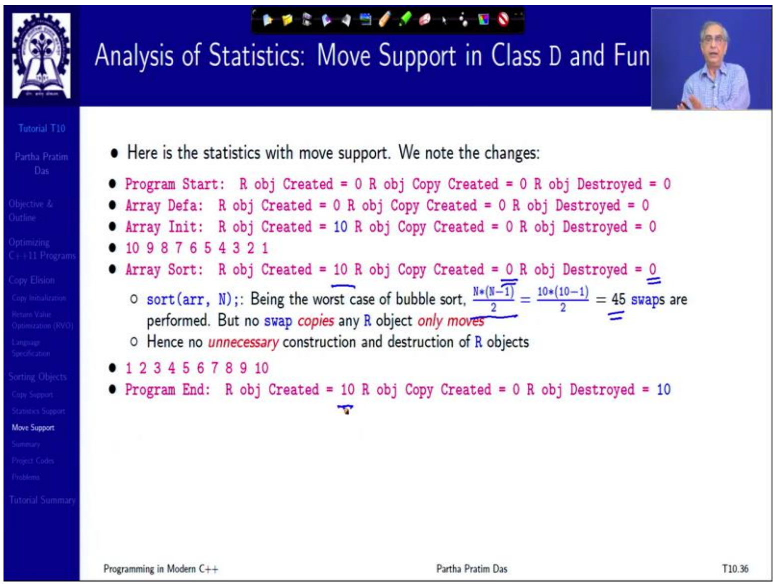
```
template<typename T>
void swap(T& a, T&b) { // Swap a and b using move
    T t = std::move(a); // t move-created from a: a's ownership transferred to t
    a = std::move(b);   // a move-assigned from b: b's ownership transferred to a
    b = std::move(t);   // b move-assigned from t: t's ownership transferred to b
} // t destroyed, but no resource destruction as t had no ownership
```

Tutorial T10
Partha Pratim Das
Objective & Outline
Optimizing C++11 Programs
Copy Elision
Copy Initialization
Return Value Optimization (RVO)
Language Specification
Sorting Objects
Copy Support
Statistics Support
Move Support
Summary
Project Codes
Problems
Tutorial Summary

Programming in Modern C++ Partha Pratim Das T10.35

Now, next we can actually the culprit is a swap function. So, what I can do is I can use std::move from the library in the swap function and make all of that move. So, that the values just keep on keep rotating around they are not copy created. So, if we do that, then naturally all of these will be the first one the second one the third one all our move assignments move of values.

(Refer Slide Time: 33:04)



Analysis of Statistics: Move Support in Class D and Fun

- Here is the statistics with move support. We note the changes:
- Program Start: R obj Created = 0 R obj Copy Created = 0 R obj Destroyed = 0
- Array Defa: R obj Created = 0 R obj Copy Created = 0 R obj Destroyed = 0
- Array Init: R obj Created = 10 R obj Copy Created = 0 R obj Destroyed = 0
- 10 9 8 7 6 5 4 3 2 1
- Array Sort: R obj Created = 10 R obj Copy Created = 0 R obj Destroyed = 0
  - sort(arr, N);: Being the worst case of bubble sort, $\frac{N*(N-1)}{2} = \frac{10*(10-1)}{2} = 45$ swaps are performed. But no swap *copies* any R object *only moves*
  - Hence no *unnecessary* construction and destruction of R objects
- 1 2 3 4 5 6 7 8 9 10
- Program End: R obj Created = 10 R obj Copy Created = 0 R obj Destroyed = 10

Programming in Modern C++ Partha Pratim Das T10.36

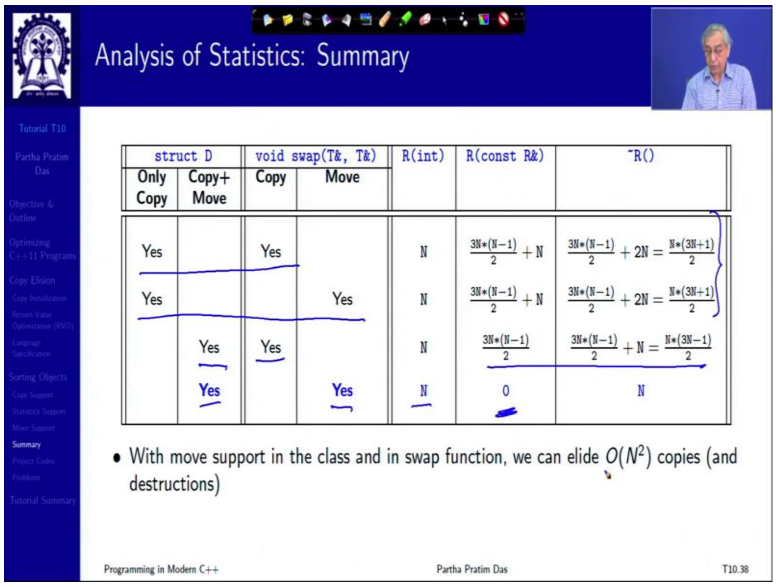
So, for that, no, R object needs to be created or destroyed. Therefore, if you with this modified swap function, if you run you will see that in short, there is no copy created of the R object neither of neither therefore, they are destroyed. So, if you analyze all that you need is 45 swaps, which needed earlier 135 construction destruction, but all those 40, all those unnecessary construction destructions have been optimized out now, at the end, you have just 10 constructions for the 10 objects and 10 destructions for the 10 objects the resources are optimally used.

(Refer Slide Time: 33:54)

Sorting Objects: Summary

Sorting Objects: Summary

Programming in Modern C++ Partha Pratim Das T10.37



Analysis of Statistics: Summary

Tutorial T10

Partha Pratim Das

Objective & Outline

Optimizing C++11 Programs

Copy Elision

Copy Initialization

Return Value Optimization (RVO)

Language Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

`struct D`		`void swap(T&, T&)`		`R(int)`	`R(const R&)`	`~R()`
Only Copy	Copy+ Move	Copy	Move			
Yes		Yes		N	$\frac{3N*(N-1)}{2} + N$	$\frac{3N*(N-1)}{2} + 2N = \frac{N*(3N+1)}{2}$
Yes			Yes	N	$\frac{3N*(N-1)}{2} + N$	$\frac{3N*(N-1)}{2} + 2N = \frac{N*(3N+1)}{2}$
	Yes	Yes		N	$\frac{3N*(N-1)}{2}$	$\frac{3N*(N-1)}{2} + N = \frac{N*(3N-1)}{2}$
	Yes		Yes	N	0	N

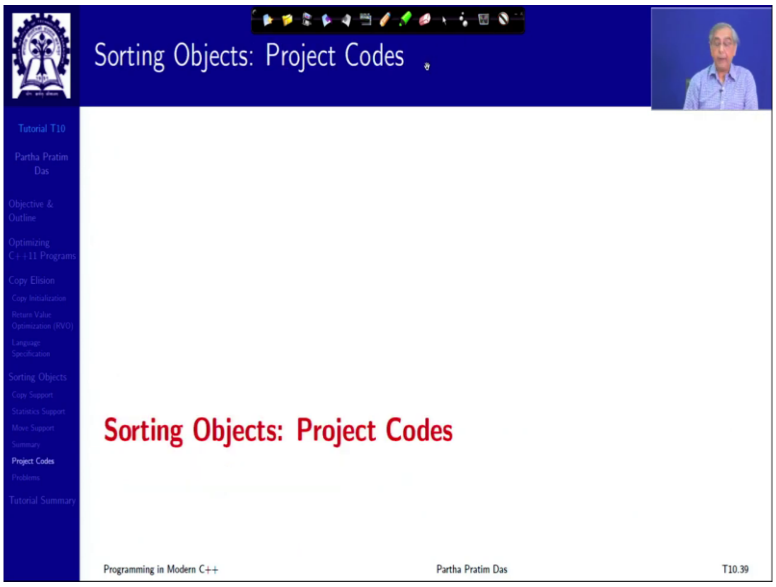
- With move support in the class and in swap function, we can elide $O(N^2)$ copies (and destructions)

Programming in Modern C++ Partha Pratim Das T10.38

So, that is the director demonstration of the benefit of what you get by providing the move support. Here I have given a table for your understanding that there are two places to support this in this particular project one is in the class D you can have only copy support or copy and move support and in the swap function, you can only copy version of soft function or move version and naturally for combination.

So, if you do say if you do not have a copy move support in D then irrespective of which swap function you use, you have the same complexity if you have move support in D but not in the swap, you get little less but you still have a lot of unnecessary copies happening. Whereas if you move supporting both, then you have no extra copies being done. So, ordering swap copies are now really optimized out. So, that is a big optimization that the program can guide to guide the compiler to do.

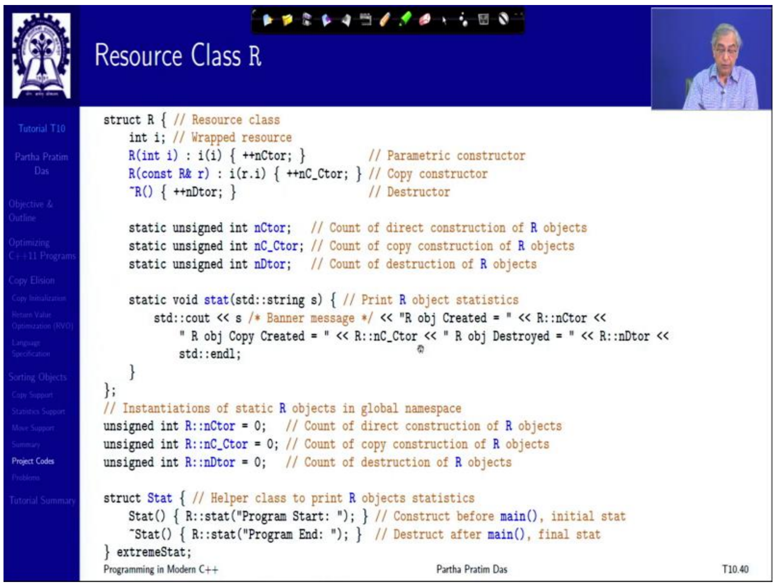
(Refer Slide Time: 35:08)



Sorting Objects: Project Codes

Sorting Objects: Project Codes

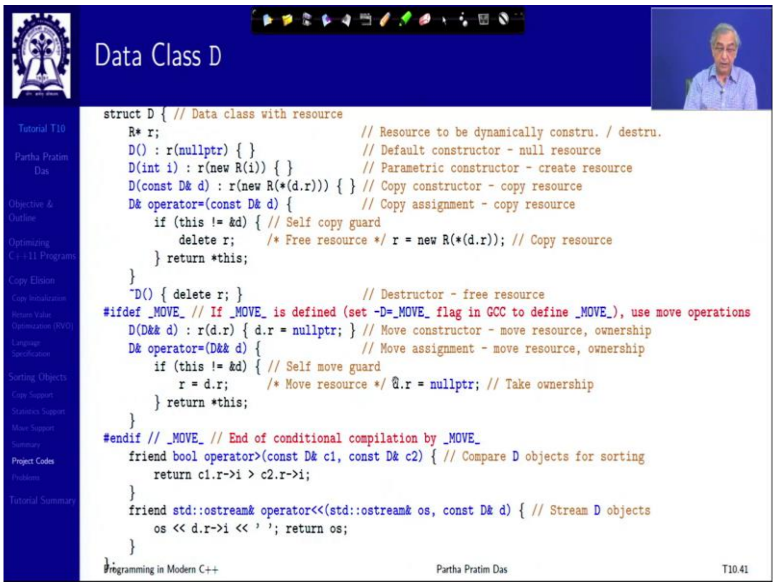
Programming in Modern C++ Partha Pratim Das T10.39



Resource Class R

```cpp
struct R { // Resource class
    int i; // Wrapped resource
    R(int i) : i(i) { ++nCtor; }          // Parametric constructor
    R(const R& r) : i(r.i) { ++nC_Ctor; } // Copy constructor
    ~R() { ++nDtor; }                     // Destructor

    static unsigned int nCtor;   // Count of direct construction of R objects
    static unsigned int nC_Ctor; // Count of copy construction of R objects
    static unsigned int nDtor;   // Count of destruction of R objects

    static void stat(std::string s) { // Print R object statistics
        std::cout << s /* Banner message */ << "R obj Created = " << R::nCtor <<
            " R obj Copy Created = " << R::nC_Ctor << " R obj Destroyed = " << R::nDtor <<
            std::endl;
    }
};
// Instantiations of static R objects in global namespace
unsigned int R::nCtor = 0;   // Count of direct construction of R objects
unsigned int R::nC_Ctor = 0; // Count of copy construction of R objects
unsigned int R::nDtor = 0;   // Count of destruction of R objects

struct Stat { // Helper class to print R objects statistics
    Stat() { R::stat("Program Start: "); } // Construct before main(), initial stat
    ~Stat() { R::stat("Program End: "); }  // Destruct after main(), final stat
} extremeStat;
```

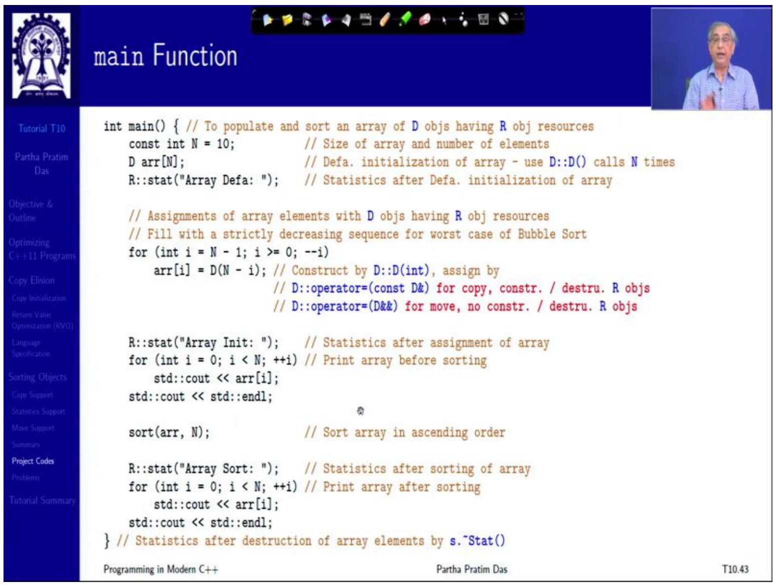
Programming in Modern C++ Partha Pratim Das T10.40



Data Class D

```cpp
struct D { // Data class with resource
    R* r;                                // Resource to be dynamically constru. / destru.
    D() : r(nullptr) { }                 // Default constructor - null resource
    D(int i) : r(new R(i)) { }           // Parametric constructor - create resource
    D(const D& d) : r(new R(*(d.r))) { } // Copy constructor - copy resource
    D& operator=(const D& d) {           // Copy assignment - copy resource
        if (this != &d) { // Self copy guard
            delete r;     /* Free resource */ r = new R(*(d.r)); // Copy resource
        } return *this;
    }
    ~D() { delete r; }                   // Destructor - free resource
#ifdef _MOVE_ // If _MOVE_ is defined (set -D=_MOVE_ flag in GCC to define _MOVE_), use move operations
    D(D&& d) : r(d.r) { d.r = nullptr; } // Move constructor - move resource, ownership
    D& operator=(D&& d) {                // Move assignment - move resource, ownership
        if (this != &d) { // Self move guard
            r = d.r;      /* Move resource */ d.r = nullptr; // Take ownership
        } return *this;
    }
#endif // _MOVE_ // End of conditional compilation by _MOVE_
    friend bool operator>(const D& c1, const D& c2) { // Compare D objects for sorting
        return c1.r->i > c2.r->i;
    }
    friend std::ostream& operator<<(std::ostream& os, const D& d) { // Stream D objects
        os << d.r->i << ' '; return os;
    }
};
```

Programming in Modern C++ Partha Pratim Das T10.41



main Function

Tutorial T10

Partha Pratim Das

Objective & Outline

Optimizing C++11 Programs

Copy Elision

Copy Initialization

Return Value Optimization (RVO)

Language Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

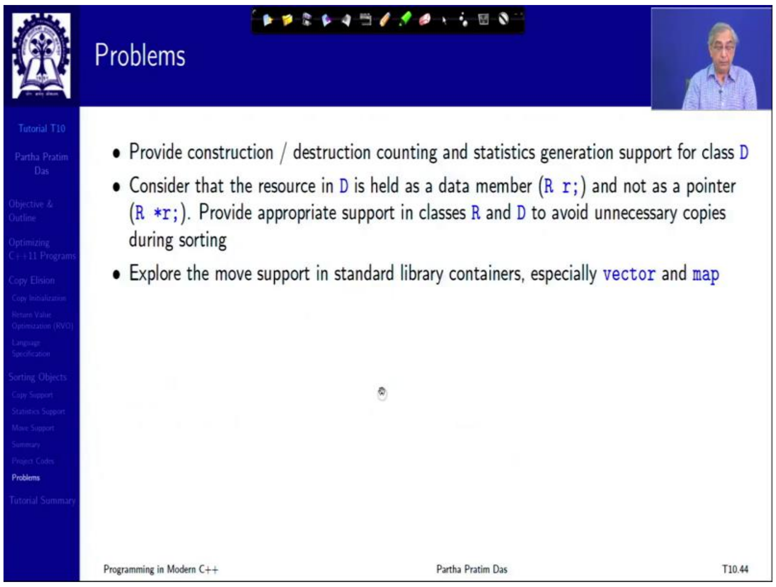
Tutorial Summary

```
int main() { // To populate and sort an array of D objs having R obj resources
    const int N = 10;           // Size of array and number of elements
    D arr[N];                   // Defa. initialization of array - use D::D() calls N times
    R::stat("Array Defa: ");    // Statistics after Defa. initialization of array

    // Assignments of array elements with D objs having R obj resources
    // Fill with a strictly decreasing sequence for worst case of Bubble Sort
    for (int i = N - 1; i >= 0; --i)
        arr[i] = D(N - i); // Construct by D::D(int), assign by
                           // D::operator=(const D&) for copy, constr. / destru. R objs
                           // D::operator=(D&&) for move, no constr. / destru. R objs

    R::stat("Array Init: ");    // Statistics after assignment of array
    for (int i = 0; i < N; ++i) // Print array before sorting
        std::cout << arr[i];
    std::cout << std::endl;

    sort(arr, N);               // Sort array in ascending order

    R::stat("Array Sort: ");    // Statistics after sorting of array
    for (int i = 0; i < N; ++i) // Print array after sorting
        std::cout << arr[i];
    std::cout << std::endl;
} // Statistics after destruction of array elements by s.~Stat()
```

Programming in Modern C++ Partha Pratim Das T10.43

And this is just one example I have shown, you can see this in the whole. In the next couple of slides, I will not go through I have just put together the entire code one after the other, as it finally stands out this struct R, the struct stat, the struct D, the two versions of the swap function, the sort function, and finally the main function.

(Refer Slide Time: 35:38)



Problems

Tutorial T10

Partha Pratim Das

Objective & Outline

Optimizing C++11 Programs

Copy Elision

Copy Initialization

Return Value Optimization (RVO)

Language Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

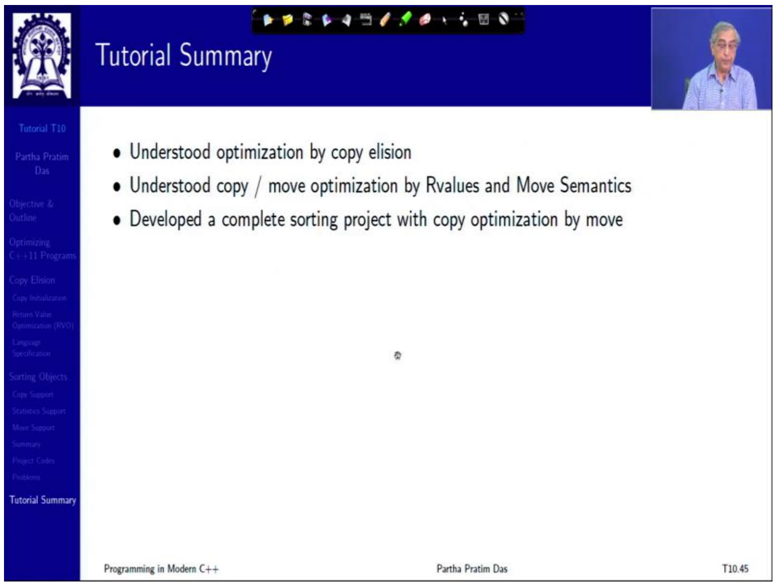
Tutorial Summary

- Provide construction / destruction counting and statistics generation support for class D
- Consider that the resource in D is held as a data member (`R r;`) and not as a pointer (`R *r;`). Provide appropriate support in classes R and D to avoid unnecessary copies during sorting
- Explore the move support in standard library containers, especially `vector` and `map`

Programming in Modern C++ Partha Pratim Das T10.44

So, you can put them together into one source file and execute with move, without move and so on and experiment further. And here are some problems that you can try out like providing construction destruction counting, and statistic generation support for the class D objects also. We are just done for class on the resource object, you can change the resource from being a pointed resource to an actual data member resource, and we will need to make changes in the class R as well. To optimize unnecessary copies out, you can explore on the move support, which are available in containers in the standard library, particularly vector and map and so on.

(Refer Slide Time: 36:25)



Tutorial Summary

- Understood optimization by copy elision
- Understood copy / move optimization by Rvalues and Move Semantics
- Developed a complete sorting project with copy optimization by move

So, in this tutorial, we have tried to explain the optimization by copy elision. And we have understood the copy and move optimization by rvalue and move semantics through a small example with a complete project. Thank you very much. I hope you this will be useful for you in understanding the rvalue and move semantics and their one of their use in this context. So, see you in the next discussion.