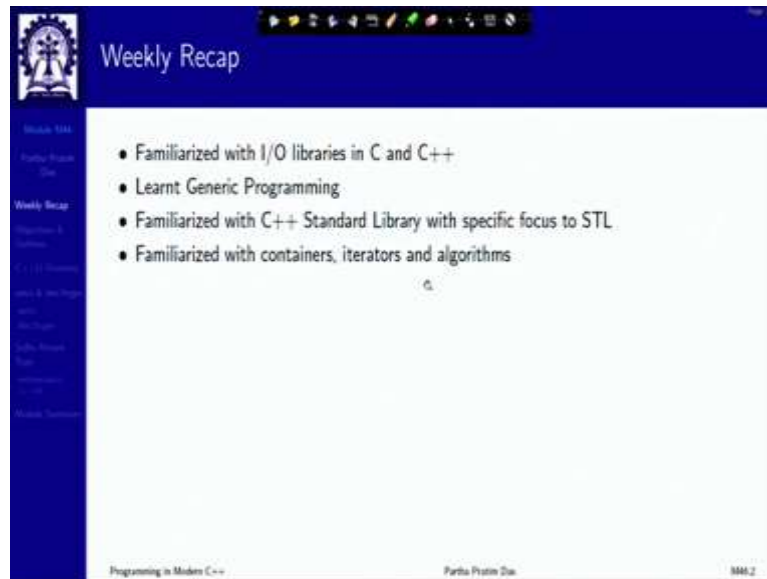


Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 46
C++ 11 and beyond: General Features: Part 1

Welcome to Programming in Modern C++. We are going to start week 10 with module 46.

(Refer Slide Time: 0:36)



In the last week we have spent the time on standard library. We are familiarized with I/O libraries both in C as well as C++. We discussed C because those can also be used in C++. We learnt about generic programming, which is a very great style to adopt in terms of C++, and then we have familiarized with C++ standard library with specific focus to Standard Template Library or STL and familiarized with containers, iterators and algorithms.

(Refer Slide Time: 1:13)

The slide is titled "Module Objectives" and features a blue header with a logo on the left and a small video inset of a speaker on the right. The main content area is white and contains a bulleted list of objectives. The footer includes the text "Programming in Modern C++", "Partho Pratim Das", and "MM-1".

- Getting familiar with C++11 and beyond: C++14, C++17, C++20, ...
- Introducing following C++11 general features;
 - auto
 - decltype
 - suffix return type

Going forward from this week, from this module and for the remaining part of the course that is the three weeks, we will primarily focus on the modern part of C++. So, far we have been discussing about C++ 03 primarily. Now, we will start getting familiar with C++ 11 and beyond. These discussions will be primarily on C++, but we will keep on mentioning additional features that came in the subsequent versions, in 14, 17, and 20. In this module we will introduce few general features of C++ auto, decltype and suffix return type.

(Refer Slide Time: 1:59)

The slide is titled "Module Outline" and features a blue header with a logo on the left and a small video inset of a speaker on the right. The main content area is white and contains a numbered list of five items. The footer includes the text "Programming in Modern C++", "Partho Pratim Das", and "MM-1".

- 1 Weekly Recap
- 2 Major C++11 Features
- 3 auto & decltype
 - auto
 - decltype
- 4 Suffix Return Type
 - decltype(auto): C++14
- 5 Module Summary

So, this is the outline which will be available on the left panel as usual.

(Refer Slide Time: 2:04)

Major C++11 Features

Sources:

- C++11 Language Extensions — General Features, isocpp.org
- C++11, cppreference.com
- Scott Meyers on C++
- How is C++ in 2019?, Quora, 2019
- C++20/17/14/11, [github](https://github.com)

Programming in Modern C++ Parth Pratap Dixi 344.5

C++ Standards

C++98	C++11	C++14	C++17	C++20
1998	2011	2014	2017	2020
Templates	Move Semantics	Reader-Writer Locks	Fold Expressions	Coroutines
STL with Containers and Algorithms	Unified Initialization	Generic Lambda Functions	constexpr #	Modules
Strings	auto and decltype		Structured Binding	Concepts
I/O Streams	Lambda Functions		std::string_view	Ranges Library
	constexpr		Parallel Algorithms of the STL	
	Multi-threading and Memory Model		File System Library	
	Regular Expressions		std::any, std::optional, and std::variant	
	Smart Pointers			
	Hash Tables			
	std::array			
ISO/IEC 14882:1998	ISO/IEC 14882:2011	ISO/IEC 14882:2014	ISO/IEC 14882:2017	ISO/IEC 14882:2020

Fixes on C++98: C++03; ISO/IEC 14882:2003, 2003
 Latest Version as of Sep-21: C++20; ISO/IEC 14882:2020, 2020
 Programming in Modern C++ Parth Pratap Dixi 344.6

So, first let us look at the major C++ features. And just to remind you these are the different standards that we have been talking about; C++ 98 or its corrected version C++ 03 has been in our discussion for all the time. C++ 11 is what we are going to discuss now with references to 14, 17 and 20, the latest version.

(Refer Slide Time: 2:31)

Major C++11 Features: Core language features

- auto and decltype
- trailing (suffix) return type
- list initialization (`initializer_list`)
- uniform initialization: brace-or-equal initializers
- enum class: scoped enums
- constexpr and literal types
- noexcept specifier and operator
- nullptr
- defaulted and deleted functions
- delegating and inherited constructors
- range-for (based on Boost)
- static_assert (based on Boost)
- Unicode string literals
- user-defined literals
- rvalue references
- move constructors and move assignment operators
- lambda expressions
- multithreaded memory model
- thread-local storage
- GC interface (removed in C++23)
- long long, char16_t and char32_t
- final and override
- type aliases
- variadic templates
- generalized (non-trivial) unions
- generalized PODs (trivial types and standard-layout types)
- attributes
- alignof and alignas

In C++ 11 several core language features have been added onto C++ 03. Some of them were like to make it a better C++, like when we studied, started studying C++ 03 we noticed that there are quite a few features which just made C++ a better language over C. Similarly, C++ 11 also has a number of features which make it a better C++.

But there are number of features which are fundamentally very different, fundamentally, I should say very significant and some of those I have shown in terms of the bold, the rvalue reference, move constructor and move assignment operator semantics, the lambda expressions, multithreaded memory model, concurrency support and so on.

And there are several others which are either convenience features that make C++ more amenable to generic programming and easier to write correct code or they are features that are required for these advanced semantics to be provided in C++ 11. We will go over, not necessarily each one of the features, but we will take up most the major features when we discuss.

(Refer Slide Time: 3:54)

The slide is titled "Major C++11 Features: Library features". It is divided into two columns. The left column, titled "Header", lists various C++11 headers. The right column, titled "Library features", lists various features and standard library additions.

Header	Library features
• <array>	• atomic operations library
• <atomic>	• <code>emplace()</code> and other use of rvalue references throughout all parts of the existing library
• <cfenv>	• <code>std::unique_ptr</code>
• <chrono>	• <code>std::move_iterator</code>
• <cstdint>	• <code>std::initializer_list</code>
• <condition_variable>	• stateful and scoped allocators
• <condint>	• <code>std::forward_list</code>
• <csurbar>	• chrono & ratio library
• <forward_list>	• new algorithms:
• <future>	◦ <code>std::all_of</code> , <code>std::any_of</code> , <code>std::none_of</code>
• <initializer_list>	◦ <code>std::find_if</code> , <code>std::copy_if</code> , <code>std::copy_n</code>
• <mutex>	◦ <code>std::move</code> , <code>std::move_backward</code>
• <mutex>	◦ <code>std::random_shuffle</code> , <code>std::shuffle</code>
• <random>	◦ <code>std::inplace_merge</code> , <code>std::partition</code> , <code>std::partition_copy</code> , <code>std::partition_point</code>
• <ratio>	◦ <code>std::inserter</code> , <code>std::inserter_until</code> , <code>std::in3eq</code> , <code>std::in3eq_until</code>
• <regex>	◦ <code>std::minmax_element</code>
• <scoped_allocator>	◦ <code>std::inplace_merge</code>
• <system_error>	◦ <code>std::iota</code>
• <thread>	◦ <code>std::iota</code>
• <thread>	◦ <code>std::iota</code>
• <tuple>	• Unicode conversion facets
• <type_traits>	• thread library
• <type_traits>	• <code>std::function</code>
• <unordered_map>	• <code>std::exception_ptr</code>
• <unordered_set>	• <code>std::error_code</code> and <code>std::error_condition</code>
	• iterator improvements: <code>std::begin</code> , <code>std::end</code> , <code>std::next</code> , <code>std::prev</code>
	• Unicode conversion functions

Besides the core language there have been significant additions to the C++ standard library. So, if we look at standard library, on the left column there is a list of headers which have been added to C++ 11 standard library. And naturally they have lot of different concepts added, so some of the important features are of the library as added to C++ have been given on the right column. We will discuss some of them as we discuss the different features.

(Refer Slide Time: 4:30)

The slide is titled "auto & decltype". It features a large red heading "auto & decltype" and a list of sources for further information.

Sources:

- auto and decltype isocpp.org
- Placeholder type specifiers (since C++11) and decltype specifier, cppreference.com
- C++ auto and decltype Explained

Now, to start with C++ general features, we start with auto and decltype for type deduction.

(Refer Slide Time: 4:38)

The slide is titled "auto" and features a list of bullet points explaining the evolution of the `auto` keyword in C++ from C++03 to C++11. The text is annotated with blue handwritten marks: a checkmark next to the first bullet point, a bracket under `auto x1 = 10;`, a circled `int` next to `// x1: int`, and a bracket under the iterator type in the second code example.

- In C++03, `auto` designated an object with *automatic storage type*. That is now *deprecated*. We must specify the type of an object at declaration though the declaration may include an initializer with type.
- In C++11 `auto` variables get the type from their *initializing expression*:
`auto x1 = 10; // x1: int`
`std::map<int, std::string> m;`
`auto i1 = m.begin(); // i1: std::map<int, std::string>::iterator`
- `const/volatile` and `reference/pointer` adornments may be added:
`const auto *x2 = &x1; // x2: const int*`
`const auto& i2 = m; // i2: const std::map<int, std::string>&`
- To get a `const_iterator`, use `cbegin` (or `cend`, `cbegin`, and `crend`) container function:
`auto ci = m.cbegin(); // ci: std::map<int, std::string>::const_iterator`
- Type deduction for `auto` is akin to that for template parameters:
`template<typename T> void f(T t);`
`f(expr); // deduce T's type from expr`
`auto v = expr; // do essentially the same thing for v's type`

This slide is identical to the one above but includes additional blue handwritten annotations: a checkmark next to the first bullet point, a bracket under `auto i1 = m.begin();`, and a bracket under the iterator type `std::map<int, std::string>::iterator` in the second code example.

- In C++03, `auto` designated an object with *automatic storage type*. That is now *deprecated*. We must specify the type of an object at declaration though the declaration may include an initializer with type.
- In C++11 `auto` variables get the type from their *initializing expression*:
`auto x1 = 10; // x1: int`
`std::map<int, std::string> m;`
`auto i1 = m.begin(); // i1: std::map<int, std::string>::iterator`
- `const/volatile` and `reference/pointer` adornments may be added:
`const auto *x2 = &x1; // x2: const int*`
`const auto& i2 = m; // i2: const std::map<int, std::string>&`
- To get a `const_iterator`, use `cbegin` (or `cend`, `cbegin`, and `crend`) container function:
`auto ci = m.cbegin(); // ci: std::map<int, std::string>::const_iterator`
- Type deduction for `auto` is akin to that for template parameters:
`template<typename T> void f(T t);`
`f(expr); // deduce T's type from expr`
`auto v = expr; // do essentially the same thing for v's type`

auto

- In C++03, auto designated an object with *automatic storage type*. That is now *deprecated*. We must specify the type of an object at declaration though the declaration may include an initializer with type
- In C++11 auto variables get the type from their *initializing expression*:


```
auto x1 = 10; // x1: int
```

```
std::map<int, std::string> m;
auto i1 = m.begin(); // i1: std::map<int, std::string>::iterator
```
- const/volatile and reference/pointer adornments may be added:


```
const auto *x2 = &x1; // x2: const int*
const auto& i2 = i1; // i2: const std::map<int, std::string>&
```
- To get a const_iterator, use cbegin (or cend, crbegin, and crend) container function:


```
auto ci = m.cbegin(); // ci: std::map<int, std::string>::const_iterator
```
- Type deduction for auto is akin to that for template parameters:


```
template<typename T> void f(T t);
f(expr); // deduce T's type from expr
auto v = expr; // do essentially the same thing for v's type
```

auto

- In C++03, auto designated an object with *automatic storage type*. That is now *deprecated*. We must specify the type of an object at declaration though the declaration may include an initializer with type
- In C++11 auto variables get the type from their *initializing expression*:


```
auto x1 = 10; // x1: int
```

```
std::map<int, std::string> m;
auto i1 = m.begin(); // i1: std::map<int, std::string>::iterator
```
- const/volatile and reference/pointer adornments may be added:


```
const auto *x2 = &x1; // x2: const int*
const auto& i2 = i1; // i2: const std::map<int, std::string>&
```
- To get a const_iterator, use cbegin (or cend, crbegin, and crend) container function:


```
auto ci = m.cbegin(); // ci: std::map<int, std::string>::const_iterator
```
- Type deduction for auto is akin to that for template parameters:


```
template<typename T> void f(T t);
f(expr); // deduce T's type from expr
auto v = expr; // do essentially the same thing for v's type
```

auto

- In C++03, auto designated an object with *automatic storage type*. That is now *deprecated*. We must specify the type of an object at declaration though the declaration may include an initializer with type
- In C++11 auto variables get the type from their *initializing expression*:


```
auto x1 = 10; // x1: int
```

```
std::map<int, std::string> m;
auto i1 = m.begin(); // i1: std::map<int, std::string>::iterator
```
- const/volatile and reference/pointer adornments may be added:


```
const auto *x2 = &x1; // x2: const int*
const auto& i2 = i1; // i2: const std::map<int, std::string>&
```
- To get a const_iterator, use cbegin (or cend, crbegin, and crend) container function:


```
auto ci = m.cbegin(); // ci: std::map<int, std::string>::const_iterator
```
- Type deduction for auto is akin to that for template parameters:


```
template<typename T> void f(T t); ✓
f(expr); // deduce T's type from expr ✓
auto v = expr; // do essentially the same thing for v's type
```

Auto as you know was a keyword in C++ 03, it used to mean automatic storage type. This is one of the very few rare features which have been deprecated from C++ 03 to C++ 11, so that meaning of automatic storage type is no more applicable in C++ 11. In C++ 03 while we are specifying an object if it is necessary to declare its type, that we know.

It is a strongly typed language, so we must declare the type and we may include an initialization with the type as well. In C++ 11 there is some liberation being given from that requirement if a variable has an initializing expression which usually most variables will have, recommendedly all variables must have, then it is possible to just write auto in the place of the type specifier.

And the type will be deduced by the compiler from the initial, type of the initializing expression. So, we see the simplest example here, I am just writing auto at this point and my initializing expression is a integer literal, int literal, therefore, I am not writing explicitly. So, this is equivalent to writing int x initialize 10, but I am not writing this int, instead I am just writing auto and the compiler will be able to deduce this.

So, this deduction is very similar to template parameter deduction that we had talked about earlier. And we will soon see, here actually it does not show a great advantage, but soon we can see that it has several advantages. For example, consider this map. We have just studied map in the, in terms of the containers, so it is a map whose key value type is int and value type is string, the map m, so as a map it will always have an iterator.

So, you know that map contain a type has an iterator type, we have discussed this while discussing map. Now, this is a long expression and it is quite possible that while writing this you will forget some details and the compilation will fail. So, what we can do? We can simply write auto for this. So, what it will do?

You are writing m.begin, m.begin returns an iterator, so the type of m.begin is the iterator of the map type that you have given here and the compiler will deduce the entire type expression on the right from this auto, so that is a big convenience to really have. You can also have constant and volatile or reference or pointer adornments that you may add, so you can say const auto* &x.

Where x1 has a auto type to 10, so x1 is int, so when you do this, you are basically taking the address of an int, so you have int* and then you are adorning with const, so you have a const

int* as a type of x2. Similarly, you can do a const reference in this way to the entire map, which becomes a convenience.

So, you can have const iterator types done as auto in the similar way except that you have to remember in place of begin, you have to begin, you have to use the iterator member function for constant iterator which is cbegin or cend, any one of those you can use, or creverse begin or creverse end, both of, all of them have const iterator type so...

You can use it simply in terms of the way the template type deduction is done, so given a template function f with type parameter T, f expression will deduce the type of T from the type of expression. This is the template type deduction and exactly the similar thing will be done by auto. So, auto in most, most, most of the cases is very simply template type deduction given to any normal part of the declaration code.

(Refer Slide Time: 9:17)

auto

- For variables not explicitly declared to be a reference:
 - Top-level consts / volatiles in the initializing type are ignored
 - Array and function names in initializing types decay to pointers

```
const std::list<int> li; ✓  
auto v1 = li;           // v1: std::list<int> ✓  
auto& v2 = li;         // v2: const std::list<int& ✓
```

```
float data[BufSize];  
auto v3 = data;        // v3: float*  
auto& v4 = data;       // v4: float (&)[BufSize]
```

- Both direct and copy initialization syntax are permitted

```
auto v1(expr);        // direct initialization syntax  
auto v2 = expr;       // copy initialization syntax
```

For auto, both syntaxes have the same meaning

- auto is closely related to decltype and has extensive use in templates and generic lambdas

```
template<class T, class U> void multiply(const vector<T>& vt, const vector<U>& vu) {  
    // ...  
    auto tap = vt[i]*vu[i]; // Compiler knows the type of tap: product of T by a U  
    // ...  
}
```

auto

- For variables not explicitly declared to be a reference:
 - Top-level consts / volatiles in the initializing type are ignored
 - Array and function names in initializing types decay to pointers

```

const std::list<int> li;
auto v1 = li;           // v1: std::list<int>
auto& v2 = li;         // v2: const std::list<int>&

float data[BufSize];
auto v3 = data;       // v3: float*
auto& v4 = data;     // v4: float (&)[BufSize]

```

- Both direct and copy initialization syntax are permitted

```

auto v1(expr);       // direct initialization syntax
auto v2 = expr;     // copy initialization syntax

```

For auto, both syntaxes have the same meaning

- auto is closely related to decltype and has extensive use in templates and generic lambdas

```

template<class T, class U> void multiply(const vector<T>& vt, const vector<U>& vu) {
    // ...
    auto tmp = vt[i]*vu[i]; // Compiler knows the type of tmp: product of T by a U
    // ...
}

```

auto

- For variables not explicitly declared to be a reference:
 - Top-level consts / volatiles in the initializing type are ignored
 - Array and function names in initializing types decay to pointers

```

const std::list<int> li;
auto v1 = li;           // v1: std::list<int>
auto& v2 = li;         // v2: const std::list<int>&

float data[BufSize];
auto v3 = data;       // v3: float*
auto& v4 = data;     // v4: float (&)[BufSize]

```

- Both direct and copy initialization syntax are permitted

```

auto v1(expr);       // direct initialization syntax
auto v2 = expr;     // copy initialization syntax

```

For auto, both syntaxes have the same meaning

- auto is closely related to decltype and has extensive use in templates and generic lambdas

```

template<class T, class U> void multiply(const vector<T>& vt, const vector<U>& vu) {
    // ...
    auto tmp = vt[i]*vu[i]; // Compiler knows the type of tmp: product of T by a U
    // ...
}

```

*vt[i] * vu[i]*

So, variables, for variables that are not explicitly declared to be a reference, the top-level constant volatile in the initialization are ignored, and array and function names, when they are being initialized, used in initialization will decay to corresponding pointer type. So, this is, auto will take the basic type that it has. So, I have a const list of int, but when I do auto v1 of that, I actually get just the list of int, I do not get the const part of it, so that is stripped off.

Similarly, if I, but if I do a reference which is, that is why it is not explicitly declared to be reference, but when it is explicitly declared to be reference it will take the entire type and then add the reference to it. So, this is the basic connotation of or the way auto puts the type. Another example, there is a array of size, BufSize of elements type float.

If you just do auto it will decay to pointer. So, element type is, it is an array of float, so it will become float* as you know. But if you use a reference, then it will preserve the entire type and give a reference to that. So, the type is float, array BufSize and you are doing a reference so you get a reference here, so this is the type that will get deduced.

Using auto, you can use both the direct initialization as well as copy initialization as you know, and in auto both these syntaxes would have the same meaning. We will see in some other context, slowly things will become different. Now, auto is related to another feature called decltype, which has extensive use in templates.

Just to start you know introducing the problem, suppose I have a multiply function, which takes elements of two different types T and U, I do not know what these T and U are and multiply takes, for these two types it takes two references of constant vectors of T type and constant vectors of U type. So, basically, possibly I am trying to do an inner product of two vectors and so on whatever.

Now, the question is what is the type of the product of the elements of vt and vu, so an element of vt is vt[i], an element of vu is vu[i] and I am multiplying them as I am doing here. The question is what is the type of this product? That depends on the type T and U and the question is how do you express that. And this is, I mean, this use of auto here can be a big convenience because otherwise you will have to spend a lot of effort to really write a type expression which will work for any type T and any type U.

(Refer Slide Time: 12:43)



auto

- For variables not explicitly declared to be a reference:
 - Top-level consts / volatiles in the initializing type are ignored
 - Array and function names in initializing types decay to pointers

```
const std::list<int> li;
auto v1 = li;           // v1: std::list<int>
auto& v2 = li;         // v2: const std::list<int>&

float data[BufSize];
auto v3 = data;        // v3: float*
auto& v4 = data;       // v4: float (&)[BufSize]
```

- Both direct and copy initialization syntax are permitted

```
auto v1(expr); ✓      // direct initialization syntax
auto v2 = expr; ✓    // copy initialization syntax
```

For auto, both syntaxes have the same meaning

- auto is closely related to decltype and has extensive use in templates and generic lambdas

```
template<class T, class U> void multiply(const vector<T>& vt, const vector<U>& vu) {
    // ...
    auto tap = vt[i]*vu[i]; // Compiler knows the type of tap: product of T by a U
    // ...
}
```

decltype

- `decltype` yields the type of an expression without evaluating it


```
int x, *ptr;
decltype(x) i1;           // i1's type is int
decltype(ptr) p1;        // p1's type is int*
std::size_t sz = sizeof(decltype(ptr)[4]); // sz = sizeof(int); ptr[4] not evaluated
```
- Fairly intuitive, but some quirks, for example, parentheses can matter:


```
struct S { double d; };
const S* p;
decltype(p->d) x1;        // double
decltype((p->d)) x2;      // const double&
```

 - Quirks rarely relevant (and can be looked up when necessary)
- Can simplify complex type expressions


```
void f(const vector<int>& a, vector<float>& b) {
    typedef decltype(a[0]+b[0]) Tmp; // Type deduced by int + float
    for (int i=0; i<b.size(); ++i) {
        Tmp p = new Tmp(a[i]+b[i]);
        // ...
    }
    // ...
}
```

decltype

- `decltype` yields the type of an expression without evaluating it


```
int x, *ptr;
decltype(x) i1;           // i1's type is int
decltype(ptr) p1;        // p1's type is int*
std::size_t sz = sizeof(decltype(ptr)[4]); // sz = sizeof(int); ptr[4] not evaluated
```
- Fairly intuitive, but some quirks, for example, parentheses can matter:


```
struct S { double d; };
const S* p;
decltype(p->d) x1;        // double
decltype((p->d)) x2;      // const double&
```

 - Quirks rarely relevant (and can be looked up when necessary)
- Can simplify complex type expressions


```
void f(const vector<int>& a, vector<float>& b) {
    typedef decltype(a[0]+b[0]) Tmp; // Type deduced by int + float
    for (int i=0; i<b.size(); ++i) {
        Tmp p = new Tmp(a[i]+b[i]);
        // ...
    }
    // ...
}
```

decltype

- `decltype` yields the type of an expression without evaluating it


```
int x, *ptr;
decltype(x) i1;           // i1's type is int
decltype(ptr) p1;        // p1's type is int*
std::size_t sz = sizeof(decltype(ptr)[4]); // sz = sizeof(int); ptr[4] not evaluated
```
- Fairly intuitive, but some quirks, for example, parentheses can matter:


```
struct S { double d; };
const S* p;
decltype(p->d) x1;        // double
decltype((p->d)) x2;      // const double&
```

 - Quirks rarely relevant (and can be looked up when necessary)
- Can simplify complex type expressions


```
void f(const vector<int>& a, vector<float>& b) {
    typedef decltype(a[0]+b[0]) Tmp; // Type deduced by int + float
    for (int i=0; i<b.size(); ++i) {
        Tmp p = new Tmp(a[i]+b[i]);
        // ...
    }
    // ...
}
```

The slide is titled "decltype" and contains the following content:

- decltype yields the type of an expression without evaluating it


```
int x, *ptr;
decltype(x) it;           // it's type is int
decltype(ptr) p1;        // p1's type is int*
std::size_t sz = sizeof(decltype(ptr)[44]); // sz = sizeof(int); ptr[44] not evaluated
```
- Fairly intuitive, but some quirks, for example, parentheses can matter:


```
struct S { double d; };
const S* p;
decltype(p->d) x1;        // double
decltype((p->d)) x2;     // const double&
```

 - Quirks rarely relevant (and can be looked up when necessary)
- Can simplify complex type expressions


```
void f(const vector<int>& a, vector<float>&& b) {
    typedef decltype(a[0]*b[0]) Tmp; // Type denoted by int * float
    for (int i=0; i<a.size(); ++i) {
        Tmp* p = new Tmp(a[i]*b[i]);
        // ...
    }
    // ...
}
```

So, let us see what you can do? You can use decltype in this context. Decltype is the type of an expression, sorry, decltype is the type of an expression without actually evaluating it. So, it just, like something similar you will remember is available for sizeof, sizeof also does not takes a type or an expression. If you give it an expression it does not evaluate the expression, but gets the size of the type of value that the expression actually represents.

So, let us say in decltype what you can get? So, I have a declaration for an int x and a pointer to int ptr. Now, if I put decltype x it will take the alias type and make it int. If I take decltype ptr it will make the type of p1 as int*. Similarly, if I take say something like declktype ptr[44]. Now, what is ptr[44]? ptr is a pointer. So, when I write ptr[44] I am thinking of it as an array of element, so this is one element in that, so it has to be of the integer type.

Now, this ptr[44], I have neither allocated nor done anything so I cannot do an evaluation of this, but without evaluating the decltype will find out the type for this. So, it sees, I mean, it looks like it is something similar to auto, but we will soon see what the advantages are. There are certain specific differences in terms of how you parenthesize but these are rare exceptions and you can always look up the manual for that.

Now, we get back to the example of that multiplication kind that we were doing. So, I want to find out the type of a[i] times b[i], where a is a constant reference to a vector of type int, b is a vector of type float and so what will be their product type. You can, looking at this you can easily say that it is int times the type denoted by int times float, so whatever questions and all that will happen, but how do you write that.

You will be able to write that by decltype a[0] into decltype b[0]. Now, here I have taken the example of int and float just to make it easy for you to understand, but this could have been a type T and this could have been a type U. Even then the type a[0] will be of type T and type b[0] will be of type U, so their product has an expression and you can do a decltype on that to find out the type.

Mind you, we are not again doing any evaluation, so the difference with auto is, auto we will need to have a specific variable for which it is finding the type, but here you are just taking the expression and finding a type without associating it to with any other variable. And that comes pretty handy in terms of doing the expression. So, let us see how does auto and decltype kind of are similar as well as they are somewhat semantically different.

(Refer Slide Time: 16:34)

auto / decltype: Semantic Differences

- auto and decltype both infer types from expressions; but they semantically differ.

```
#include <iostream>

int main() {
    int a = 5;      // int
    int& b = a;     // int&
    const int c = 7; // const int
    const int& d = c; // const int&

    // auto never deduces arguments like cv-qualifier or reference
    auto a_auto = a; // int
    auto b_auto = b; // int&
    auto c_auto = c; // const int
    auto d_auto = d; // const int&

    // cv-qualifier or reference needs to be explicitly added
    auto& b_auto_ref = a; // int&
    const auto c_auto_const = c; // const int

    // decltype deduces the complete type of the expression
    decltype(a) a_dt; // int // [C++14] decltype(auto) a_dt_auto = a; // int
    decltype(b) b_dt = b; // int& // [C++14] decltype(auto) b_dt_auto = b; // int&
    decltype(c) c_dt = c; // const int // [C++14] decltype(auto) c_dt_auto = c; // const int
    decltype(d) d_dt = d; // const int& // [C++14] decltype(auto) d_dt_auto = d; // const int&
}
```

auto / decltype: Semantic Differences

- auto and decltype both infer types from expressions; but they semantically differ:

```

#include <iostream>

int main() {
    int a = 5; // int ✓
    int& b = a; // int& ✓
    const int c = 7; // const int
    const int& d = c; // const int&

    // auto never deduces subtypes like cv-qualifier or reference
    auto a_auto = a; // int ✓
    auto b_auto = b; // int ✓
    auto c_auto = c; // int
    auto d_auto = d; // int

    // cv-qualifier or reference needs to be explicitly added
    auto& b_auto_ref = a; // int&
    const auto c_auto_const = a; // const int

    // decltype deduces the complete type of the expression
    decltype(a) a_dt; // int // [C++14] decltype(auto) a_dt_auto = a; // int
    decltype(b) b_dt = b; // int& // [C++14] decltype(auto) b_dt_auto = b; // int&
    decltype(c) c_dt = c; // const int // [C++14] decltype(auto) c_dt_auto = c; // const int
    decltype(d) d_dt = d; // const int& // [C++14] decltype(auto) d_dt_auto = d; // const int&
}

```

auto / decltype: Semantic Differences

- auto and decltype both infer types from expressions; but they semantically differ:

```

#include <iostream>

int main() {
    int a = 5; // int
    int& b = a; // int&
    const int c = 7; // const int
    const int& d = c; // const int&

    // auto never deduces subtypes like cv-qualifier or reference
    auto a_auto = a; // int
    auto b_auto = b; // int
    auto c_auto = c; // int
    auto d_auto = d; // int

    // cv-qualifier or reference needs to be explicitly added
    auto& b_auto_ref = a; // int&
    const auto c_auto_const = a; // const int

    // decltype deduces the complete type of the expression
    decltype(a) a_dt; // int // [C++14] decltype(auto) a_dt_auto = a; // int ✓
    decltype(b) b_dt = b; // int& // [C++14] decltype(auto) b_dt_auto = b; // int& ✓
    decltype(c) c_dt = c; // const int // [C++14] decltype(auto) c_dt_auto = c; // const int ✓
    decltype(d) d_dt = d; // const int& // [C++14] decltype(auto) d_dt_auto = d; // const int& ✓
}

```

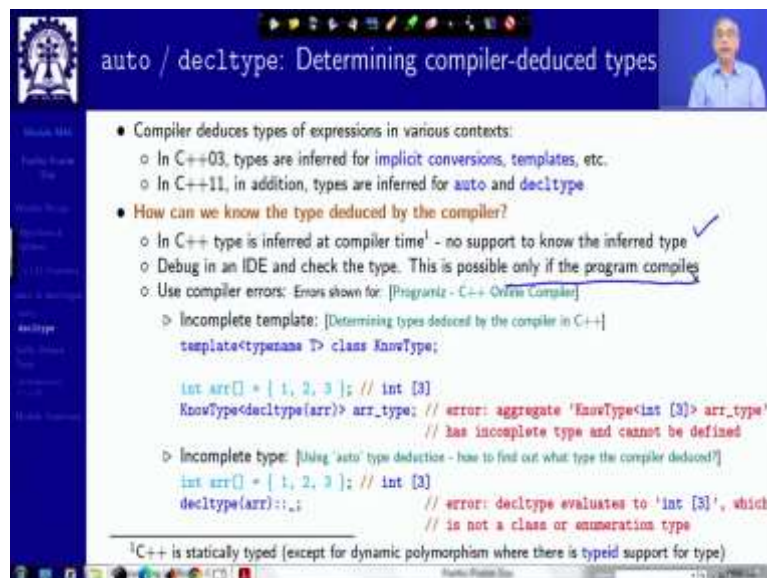
So, first consider a set of declarations there `int a` `int` reference, `const int` `const int` reference, so if you check the type of all these variables `a`, `b`, `c`, `d`, you will get the corresponding types as written. Now, if you do an `auto` using these respective variables, whether the variable is of type `int` or of `int` reference or `const int` or `const int` reference, `auto` will always, what it will do, it will strip off the `const`, it will strip off the reference, it will just give you the basic type. So, all of these will be `int`.

So, if you have to put the CV qualifier or reference you have to add that explicitly. So, if you do that `a` using a which is of type `int` and you say `auto&`, this will become `int&`. Similarly, if you do, take `a` and use `const auto`, then it will become `const int`, so you have to explicitly put that. In contrast if you use this, if you do this with `decltype`, then you can simply write `decltype a`.

What does decltype take? It takes an expression. So, a is an expression, it is a variable so it is an expression, so it will give you the type of a, which int, this part is same. But if you do decltype b, you will get int&, which was the actual type of b. Auto was giving you just the int, so decltype differs from the auto is that it does not do the strip off of reference pointer or const volatile, etcetera, it gives you the actual type that you have.

Now, in C++ 14 you have another version of decltype, we just uses the word auto in place of the expression. So, what this means is this auto relates to the initializing expression. So, when do decltype(auto) a_dt_auto, it initialized with a, it will take the type from a, similarly, here it will take the type from b and correspondingly it follows the decltype semantics and it will give you the corresponding types. So, this part is just that you need to learn and remember.

(Refer Slide Time: 19:05)



auto / decltype: Determining compiler-deduced types

- Compiler deduces types of expressions in various contexts:
 - In C++03, types are inferred for implicit conversions, templates, etc.
 - In C++11, in addition, types are inferred for auto and decltype
- How can we know the type deduced by the compiler?
 - In C++ type is inferred at compiler time¹ - no support to know the inferred type ✓
 - Debug in an IDE and check the type. This is possible only if the program compiles
 - Use compiler errors: Errors shown for: [Programiz - C++ Online Compiler]

▷ Incomplete template: [Determining types deduced by the compiler in C++]

```
template<typename T> class KnowType;
```

```
int arr[] = { 1, 2, 3 }; // int [3]
KnowType<decltype(arr)> arr_type; // error: aggregate 'KnowType<int [3]> arr_type'
// has incomplete type and cannot be defined
```

▷ Incomplete type: [Using 'auto' type deduction - how to find out what type the compiler deduced?]

```
int arr[] = { 1, 2, 3 }; // int [3]
decltype(arr)::; // error: decltype evaluates to 'int [3]', which
// is not a class or enumeration type
```

¹C++ is statically typed (except for dynamic polymorphism where there is typeid support for type)

auto / decltype: Determining compiler-deduced types

- Compiler deduces types of expressions in various contexts:
 - In C++03, types are inferred for implicit conversions, templates, etc.
 - In C++11, in addition, types are inferred for auto and decltype
- How can we know the type deduced by the compiler?
 - In C++ type is inferred at compiler time¹ - no support to know the inferred type
 - Debug in an IDE and check the type. This is possible only if the program compiles
 - Use compiler errors: Errors shown for: [Programiz - C++ Online Compiler]
 - > Incomplete template: [Determining types deduced by the compiler in C++]


```
template<typename T> class KnowType;
int arr[] = { 1, 2, 3 }; // int [3]
KnowType<decltype(arr)> arr_type; // error: aggregate 'KnowType<int [3]> arr_type'
// has incomplete type and cannot be defined
```
 - > Incomplete type: [Using 'auto' type deduction - how to find out what type the compiler deduced?]


```
int arr[] = { 1, 2, 3 }; // int [3]
decltype(arr)::; // error: decltype evaluates to 'int [3]', which
// is not a class or enumeration type
```

¹C++ is statically typed (except for dynamic polymorphism where there is typeid support for type)

auto / decltype: Determining compiler-deduced types

- Compiler deduces types of expressions in various contexts:
 - In C++03, types are inferred for implicit conversions, templates, etc.
 - In C++11, in addition, types are inferred for auto and decltype
- How can we know the type deduced by the compiler?
 - In C++ type is inferred at compiler time¹ - no support to know the inferred type
 - Debug in an IDE and check the type. This is possible only if the program compiles
 - Use compiler errors: Errors shown for: [Programiz - C++ Online Compiler]
 - > Incomplete template: [Determining types deduced by the compiler in C++]


```
template<typename T> class KnowType;
int arr[] = { 1, 2, 3 }; // int [3]
KnowType<decltype(arr)> arr_type; // error: aggregate 'KnowType<int [3]> arr_type'
// has incomplete type and cannot be defined
```
 - > Incomplete type: [Using 'auto' type deduction - how to find out what type the compiler deduced?]


```
int arr[] = { 1, 2, 3 }; // int [3]
decltype(arr)::; // error: decltype evaluates to 'int [3]', which
// is not a class or enumeration type
```

¹C++ is statically typed (except for dynamic polymorphism where there is typeid support for type)

Now, the question certainly you might come to is well, if you have, I mean, as a programmer how do you know using auto or decltype, what type actually the compiler is inferring. This is happening at the compile time, so at the run time you have no way of, you have no way of executing and knowing that, so C++ also does, C++ 03 also does lot of inferences in terms of implicit conversion, templates and so on.

C++ 11 in addition is inferring for auto and decltype, the question is how do you get to know that. Unfortunately, the compiler does do the inference but there is no support to let that infer type known to you, for the simple reason that the compiler's job is to translate it to a program which you will execute. Compiler is like a service provider, it is not supposed to give its internals to you. So, how do you actually know the type that has been inferred?

One way could be you can use your IDE, the debugger, particularly run the program and do a break point and at the debugger you can see what is the type, debuggers usually give that information. This is clumsy, as well as the difficulty is you will be able to do this only when your program compiles. Suppose you need to know the type that has been found by auto; that has been inferred by auto or decltype.

And because your program is not compiling, it is saying that there is some error, so how do you get to know that? So, there is no standard way, but I have given you here two possible hacks that you can use and I have given the references software, these are been discussed at greater length, one is you can just declare a template, but not define it. So, it is like a forward declaration.

You say knowledge type is a template of with the type parameter T, but you have not said what is that class is. Then suppose you have an array `int arr`, how do you get to know its type, its type is `int [3]`. So, what you can do is you try to instantiate the template with the decltype of `arr`. So, `arr` is an expression because it is a variable, it is an array of size 3 of int, you do try to do a decltype of that. This type you are trying to pass as T.

As you do that so you are trying to instantiate that template, but that template has not been defined, so the compiler will not be able to instantiate the template. So, what will do? The compiler will give a error. So, for example, this is, this error message is particularly from this online compiler, you can try this out.

So, in the error message you will see that the type inferred is embedded because compiler is trying to tell you that this is something which is wrong, so it cannot be, it is an incomplete type it cannot be defined. Other, even simpler way to do this is not even trying to do a template, but just do `decltype(arr)::`, say `underscore`, `underscore` is a name of a variable, any variable name can start with an underscore.

Now, what will this, what does this mean? This, you are doing `decltype(arr)::`, which means this is a type, so it is saying `type :: something`, so it is expected to give you a class or an enumeration type, `type :: syntax` basically means that, but there is nothing that actual type is `int [3]` and it is not a class, it is not an enumerated type, so the compiler will not be able to interpret this particular statement that you have written.

So, the compiler gives you an error, it says, 'decltype evaluates to' and that is what you wanted, to know what is the type that decltype is inferring, which is not a class or of enumeration type, which is a very quick way I always use it to know what is the type of an expression that is being inferred. You can use it in any context, it is a very nice one-line hack, which will give you the information through compiler messages.

(Refer Slide Time: 23:30)

```

auto / decltype: Determining compiler-deduced types

• We may also use typeid operator to know the type
  ◦ Not a good idea as typeid is meant for dynamic type
  ◦ The name of type returned by typeid is encoded

#include <bits/stdc++.h> // includes all standard library, but is not a standard header file of GNU C++
// DO NOT USE

using namespace std;

int main() {
    int x; // int ✓
    char y; // char ✓
    cout << typeid(x).name() << endl; // i ✓
    cout << typeid(y).name() << endl; // c ✓

    vector<int> vi; // std::vector<int> ✓
    vector<double> vd; // std::vector<double> ✓
    cout << typeid(vi).name() << endl; // St$vectorIiSalIHEE ✓
    cout << typeid(vd).name() << endl; // St$vectorIdSalIHEE ✓

    auto it = vi.begin(); // std::vector<int>::iterator
    decltype(vi.cbegin()) cit = vi.cbegin(); // std::vector<int>::const_iterator
    cout << typeid(it).name() << endl; // N9_gnu_cxx17_normal_iteratorIiSt$vectorIiSalIHEE ✓
    cout << typeid(cit).name() << endl; // N9_gnu_cxx17_normal_iteratorIPKSt$vectorIiSalIHEE ✓
  }
  
```

Of course, you can use typeid operators, operator also, we have talked about typeid operator during dynamic polymorphism, but it is not a good thing to do because it is meant for dynamic type, but you can still apply typeid on anything. So, if you do that on a, so this is your original variables, so what you need to do is typeid pass the expression variable name and then do dot name typeid returns your structure so dot name has the name.

So, for x which is an integer you get as i, c. Now, if you try to do this on vector which are basically having these types you get these kind of strings, which are compilers own internal representation of the type that the entire thing it means. So, by the typeid you can get the type, but it is in a format that is very, very difficult to decode and it is compiler dependent, use a different compiler you will get a different coding. So, typeid method to know, type is not actually useful, you have to use one of the two hacks that I have just discussed.

(Refer Slide Time: 24:52)

Suffix / Trailing Return Type

Sources:

- Function declaration, cppreference.com
- When to use `decltype(auto)` versus `auto`?, cplusplus.com

Programming in Modern C++ Parth Patel Dev 584/35

Now, let us using `auto` and `decltype`, let us look at a very interesting feature that C++ 11 has introduced called the suffix or trailing return type.

(Refer Slide Time: 25:00)

Suffix / Trailing Return Type

- We really need `decltype` if we need a type for something that is not a variable, such as a return type. Consider:

```
template<class T, class U>  
T* mul(T x, U y) { return x*y; }
```
- How to write the return type? It is the type of `x*y` - but how can we say that? Use `decltype`?

```
template<class T, class U>  
decltype(x*y) mul(T x, U y) { return x*y; } // scope problem! types of x and y not known
```
- That won't work because `x` and `y` are not in scope. So:

```
template<class T, class U>  
{decltype(T*(0)**(U*)(0))} mul(T x, U y) { return x*y; } // ugly! and error prone
```
- Put the return type where it belongs, after the arguments:

```
template<class T, class U>  
auto mul(T x, U y) -> decltype(x*y) { return x*y; }
```

We use the notation `auto` to mean return type to be deduced or specified later

Suffix / Trailing Return Type

- We really need `decltype` if we need a type for something that is not a variable, such as a return type. Consider:


```
template<class T, class U>
T?? mul(T x, U y) { return x*y; }
```
- How to write the return type? It is the type of `x*y` – but how can we say that? Use `decltype`?


```
template<class T, class U>
decltype(x*y) mul(T x, U y) { return x*y; } // scope problem! types of x and y not known
```
- That won't work because `x` and `y` are not in scope. So:


```
template<class T, class U>
decltype((T*)(0))*((U*)(0)) mul(T x, U y) { return x*y; } // ugly! and error prone
```
- Put the return type where it belongs, after the arguments:


```
template<class T, class U>
auto mul(T x, U y) -> decltype(x*y) { return x*y; }
```

We use the notation `auto` to mean return type to be deduced or specified later

Suffix / Trailing Return Type

- We really need `decltype` if we need a type for something that is not a variable, such as a return type. Consider:


```
template<class T, class U>
T?? mul(T x, U y) { return x*y; }
```
- How to write the return type? It is the type of `x*y` – but how can we say that? Use `decltype`?


```
template<class T, class U>
decltype(x*y) mul(T x, U y) { return x*y; } // scope problem! types of x and y not known
```
- That won't work because `x` and `y` are not in scope. So:


```
template<class T, class U>
decltype((T*)(0))*((U*)(0)) mul(T x, U y) { return x*y; } // ugly! and error prone
```
- Put the return type where it belongs, after the arguments:


```
template<class T, class U>
auto mul(T x, U y) -> decltype(x*y) { return x*y; }
```

We use the notation `auto` to mean return type to be deduced or specified later

Suffix / Trailing Return Type

- We really need `decltype` if we need a type for something that is not a variable, such as a return type. Consider:


```
template<class T, class U>
T?? mul(T x, U y) { return x*y; }
```
- How to write the return type? It is the type of `x*y` – but how can we say that? Use `decltype`?


```
template<class T, class U>
decltype(x*y) mul(T x, U y) { return x*y; } // scope problem! types of x and y not known
```
- That won't work because `x` and `y` are not in scope. So:


```
template<class T, class U>
decltype((T*)(0))*((U*)(0)) mul(T x, U y) { return x*y; } // ugly! and error prone
```
- Put the return type where it belongs, after the arguments:


```
template<class T, class U>
auto mul(T x, U y) -> decltype(x*y) { return x*y; }
```

We use the notation `auto` to mean return type to be deduced or specified later

The question is when do we need a decltype? When we have a variable we can do auto on for that, but when we do not have a variable, then I need to use decltype. I cannot do an auto. So, what is that, when we need a type where there is no variable, when that variable is temporary?

For example, in a return type. So, let us say I am trying to define a multiply function of template type parameters being T and U, the expression is x times y, the question is what do I write here. In terms of T and U I have to be able to write the return type, but I do not know what to write, so I know the type to be returned is the type of x times y, type of x times y.

So, I say, ok, not a problem, I will just use decltype on this. decltype should tell me the type, so I will do decltype x times y and I will write the function. This will not compile. Why will it not compile because the compiler takes the code from left to right, so it comes across x before the scope of x has actually started, scope of x starts, type is defined here and the scope starts here. But you need to know x at this point, similarly you need to know y at this point.

You cannot know that, there is no way to know this, so because of the scope problem you cannot use decltype directly there. So, if you are a very, very smart programmer you might want to do something smart to get there. What you said ok, I can always use 0, it is a constant, stands for the null pointer and cast it to, decltype cast it to T*, so I take 0, which is, which does not have a type or it is an integer type, I cast it to T*.

So, I get a pointer of T type, similarly, I get a pointer of U type. I dereference I will get a object of U type, I dereference T type I will get an, I dereference the second one I get a object of U type, so if I mark it well then this is an object of T type, this is an object of U type. I am doing the same thing as this, but I am not using the name x and y, and then I multiply, this is a multiplication, this star is a binary multiplication.

If I do that, certainly I will be able to get what is the type of x times y, but certainly as you can see this is a very, very clumsy and error prone way of doing it and in general it is difficult to write this kind of expression. So, what suffix return type, a trailing return type does, it pushes the return type to the place where it actually belongs, that is after the function.

So, what it does is in the usual traditional return type position you just write auto that it will be deduced, and then after the function header and before the function body with an arrow you put the return type. This is available in general. And in that you say that the return type is

decltype of x into y. Now, the advantage is x is already in the scope, y is already in the scope, so x and y are defined symbols their types, so you can deduce.

The compiler can deduce the decltype, the only thing is you are placing the return type after the header, not before the header, before the header you just have a placeholder to say that I want the compiler to deduce this. So, this is what is called the suffix or trailing return type mechanism by which you can let the compiler deduce that.

(Refer Slide Time: 29:29)

Suffix / Trailing Return Type

- The suffix syntax is not primarily about templates and type deduction, it is really about scope

```
struct List {  
    struct Link { /* ... */ };  
    Link* erase(Link* p); // remove p and return the link before p  
    // ...  
};  
List::Link* List::erase(Link* p) { /* ... */ } ✓
```

- The first `List::` is necessary only because the scope of `List` is not entered until the second `List::`. Better:
`auto List::erase(Link* p) -> Link* { /* ... */ } // No explicit qualification for Links`
- To declare objects, `decltype` can replace `auto`, but more verbosely.
`std::vector<std::string> vs;
auto i = vs.begin();
decltype(vs.begin()) i = vs.begin();`
- Only `decltype` solves the template-return-type problem in C++11 (by Perfect Forwarding)
- `auto` is for everybody. `decltype` is primarily for template authors

Suffix / Trailing Return Type

- The suffix syntax is not primarily about templates and type deduction, it is really about scope

```
struct List {  
    struct Link { /* ... */ };  
    Link* erase(Link* p); // remove p and return the link before p  
    // ...  
};  
List::Link* List::erase(Link* p) { /* ... */ }
```

- The first `List::` is necessary only because the scope of `List` is not entered until the second `List::`. Better:
`auto List::erase(Link* p) -> Link* { /* ... */ } // No explicit qualification for Links`
- To declare objects, `decltype` can replace `auto`, but more verbosely.
`std::vector<std::string> vs;
auto i = vs.begin();
decltype(vs.begin()) i = vs.begin();`
- Only `decltype` solves the template-return-type problem in C++11 (by Perfect Forwarding)
- `auto` is for everybody. `decltype` is primarily for template authors

And actually if you think of it this syntax, this suffix syntax is not necessarily about templates it is about scoping. It is type deduction in a scope rule, because scope is what you are not getting, for example, if you have a list of link elements and if you have a member function

erase that takes a link pointer and returns you a link pointer then outside of the class, if you outline the body of this function, then you have to write it like this.

You will have to write `List::Link*`, and then `List::erase`. Now, this `List::Link*` is required because you get to know that this is a member function of `erase`, the member function of `List` only after you have come to this point, but you need to know about `Link*` before that, so you need to write it twice.

Now, what you can do, you can just write an `auto` here that is you are not writing this, you are just directly writing the member function. So, you write `auto`, because this is necessarily the return type and then you put the return type as a suffix pointer `Link*`. So, there is no specific qualification for the `Link` that you need because you will be able to deduce, compiler will be able to deduce that from the suffix notation itself in the `auto`. So, this is how the `decltype` specifically help you in doing variety of different deduction.

(Refer Slide Time: 31:17)



The slide is titled "Suffix / Trailing Return Type: C++14" and features a small video inset of a speaker in the top right corner. The main content consists of three bullet points explaining the evolution of trailing return types in C++:

- In C++11, we use suffix return type to specify return type of templates to be inferred:**

```
template<class T, class U>  
auto mul(T x, U y) -> decltype(x*y) { return x*y; } ✓
```

This is unclear because the return expression has to be repeated within `decltype`.
- In C++14, suffix type can be skipped and the return type is deduced directly:**

```
template<class T, class U>  
auto mul(T x, U y) { return x*y; }
```

For compatibility, it still supports the suffix return type. Hence, the following is still valid:

```
template<class T, class U>  
auto mul(T x, U y) -> decltype(x*y) { return x*y; }
```
- C++14, further introduces `decltype(auto)` for deducing the return type by the semantics of `decltype` and not the semantics of `auto`. No suffix return type is allowed here**

```
template<class T, class U>  
decltype(auto) mul(T x, U y) { return x*y; }
```

A final bullet point states: "We present an example to highlight the differences between `auto` and `decltype(auto)`".

Suffix / Trailing Return Type: C++14

- In C++11, we use suffix return type to specify return type of templates to be inferred:


```
template<class T, class U>
auto mul(T x, U y) -> decltype(x*y) { return x*y; }
```

 This is unclear because the return expression has to be repeated within `decltype`
- In C++14, suffix type can be skipped and the return type is deduced directly:


```
template<class T, class U>
auto mul(T x, U y) { return x*y; }
```

 For compatibility, it still supports the suffix return type. Hence, the following is still valid:


```
template<class T, class U>
auto mul(T x, U y) -> decltype(x*y) { return x*y; }
```
- C++14, further introduces `decltype(auto)` for deducing the return type by the semantics of `decltype` and not the semantics of `auto`. *No suffix return type is allowed here*

```
template<class T, class U>
decltype(auto) mul(T x, U y) { return x*y; }
```
- We present an example to highlight the differences between `auto` and `decltype(auto)`

And you can, as you can see we can use it in C++ 11 as we have seen this, we can use it in this form. When you come to C++ 14, you get something which is even more interesting. You can, the suffix type, this suffix type specification if you see, the only problem is it has to write the return expression twice, but then the compiler knows the return expression. Why do I have to write it twice? The compiler can generate this entire part itself.

That is a simple common sense. So, C++11 just allows you to write `auto`, it is like any other function that you write, just you can write `auto`, of course, for backward compatibility with C++ 11 it will also allow you to use the suffix return type. C++ 11 introduces another, which is `decltype(auto)`. So, you can write `decltype(auto)` also in place of just `auto`. But if you use `decltype(auto)` then you will not be able to write the suffix return type because it does not need a backward compatibility, `decltype(auto)` was not there in C++ 11.

(Refer Slide Time: 32:36)

Suffix / Trailing Return Type: C++14: auto / decltype

```
#include <iostream>

// returns prvalue: plain auto never deduces to a reference. prvalue is a pure rvalue - THE later
template<typename T> auto foo(T t) { return t.value(); } ✓

// return lvalue: auto always deduces to a reference
template<typename T> auto& bar(T t) { return t.value(); } ✓

// return prvalue if t.value() is an rvalue
// return lvalue if t.value() is an lvalue
// decltype(auto) has decltype semantics (without having to repeat the expression)
template<typename T> decltype(auto) foobar(T t) { return t.value(); } ✓

int main() {
    struct A { int i = 0; int& value() { return i; } } a; ✓
    struct B { int i = 0; int value() { return i; } } b; ✓

    foo(a) = 20; // *** error: expression evaluates to prvalue of type int
    foo(b);     // fine: expression evaluates to prvalue of type int

    bar(a) = 20; // fine: expression evaluates to lvalue of type int
    bar(b);     // *** error: auto& always deduces to a reference

    foobar(a) = 20; // fine: expression evaluates to lvalue of type int&
    foobar(b);     // fine: expression evaluates to prvalue of type int
}
```

Suffix / Trailing Return Type: C++14: auto / decltype

```
#include <iostream>

// returns prvalue: plain auto never deduces to a reference. prvalue is a pure rvalue - THE later
template<typename T> auto foo(T t) { return t.value(); }

// return lvalue: auto always deduces to a reference
template<typename T> auto& bar(T t) { return t.value(); }

// return prvalue if t.value() is an rvalue
// return lvalue if t.value() is an lvalue
// decltype(auto) has decltype semantics (without having to repeat the expression)
template<typename T> decltype(auto) foobar(T t) { return t.value(); }

int main() {
    struct A { int i = 0; int& value() { return i; } } a;
    struct B { int i = 0; int value() { return i; } } b;

    foo(a) = 20; // *** error: expression evaluates to prvalue of type int
    foo(b);     // fine: expression evaluates to prvalue of type int

    bar(a) = 20; // fine: expression evaluates to lvalue of type int
    bar(b);     // *** error: auto& always deduces to a reference

    foobar(a) = 20; // fine: expression evaluates to lvalue of type int&
    foobar(b);     // fine: expression evaluates to prvalue of type int
}
```

Suffix / Trailing Return Type: C++14: auto / decltype

```

#include <iostream>

// returns prvalue: plain auto never deduces to a reference. prvalue is a pure rvalue - THE later
template<typename T> auto foo(T t) { return t.value(); }

// return lvalue: auto always deduces to a reference
template<typename T> auto& bar(T t) { return t.value(); }

// return prvalue if t.value() is an rvalue
// return lvalue if t.value() is an lvalue
// decltype(auto) has decltype semantics (without having to repeat the expression)
template<typename T> decltype(auto) foobar(T t) { return t.value(); }

int main() {
    struct A { int i = 0; int& value() { return i; } } a;
    struct B { int i = 0; int value() { return i; } } b;

    foo(a) = 20; // *** error: expression evaluates to prvalue of type int
    foo(b);     // fine: expression evaluates to prvalue of type int

    bar(a) = 20; // fine: expression evaluates to lvalue of type int
    bar(b);     // *** error: auto& always deduces to a reference

    foobar(a) = 20; // fine: expression evaluates to lvalue of type int&
    foobar(b);     // fine: expression evaluates to prvalue of type int
  
```

Suffix / Trailing Return Type: C++14: auto / decltype

```

#include <iostream>

// returns prvalue: plain auto never deduces to a reference. prvalue is a pure rvalue - THE later
template<typename T> auto foo(T t) { return t.value(); }

// return lvalue: auto always deduces to a reference
template<typename T> auto& bar(T t) { return t.value(); }

// return prvalue if t.value() is an rvalue
// return lvalue if t.value() is an lvalue
// decltype(auto) has decltype semantics (without having to repeat the expression)
template<typename T> decltype(auto) foobar(T t) { return t.value(); }

int main() {
    struct A { int i = 0; int& value() { return i; } } a;
    struct B { int i = 0; int value() { return i; } } b;

    foo(a) = 20; // *** error: expression evaluates to prvalue of type int
    foo(b);     // fine: expression evaluates to prvalue of type int

    bar(a) = 20; // fine: expression evaluates to lvalue of type int
    bar(b);     // *** error: auto& always deduces to a reference

    foobar(a) = 20; // fine: expression evaluates to lvalue of type int&
    foobar(b);     // fine: expression evaluates to prvalue of type int
  
```

Suffix / Trailing Return Type: C++14: auto / decltype

```

#include <iostream>

// returns prvalue: plain auto never deduces to a reference. prvalue is a pure rvalue - THE later
template<typename T> auto foo(T t) { return t.value(); }

// return lvalue: auto always deduces to a reference
template<typename T> auto& bar(T t) { return t.value(); }

// return prvalue if t.value() is an rvalue
// return lvalue if t.value() is an lvalue
// decltype(auto) has decltype semantics (without having to repeat the expression)
template<typename T> decltype(auto) foobar(T t) { return t.value(); }

int main() {
    struct A { int i = 0; int& value() { return i; } } a;
    struct B { int i = 0; int value() { return i; } } b;

    foo(a) = 20; // *** error: expression evaluates to prvalue of type int
    foo(b);     // fine: expression evaluates to prvalue of type int

    bar(a) = 20; // fine: expression evaluates to lvalue of type int
    bar(b);     // *** error: auto& always deduces to a reference

    foobar(a) = 20; // fine: expression evaluates to lvalue of type int&
    foobar(b);     // fine: expression evaluates to prvalue of type int
  
```

So, we will present a simple example to show what is the difference, here you have, I have defined three functions foo, bar and foobar. Now, if you see here, then, first let us look at the parameters, parameters are going to be struct a or struct b, both of which have a member function value, this returns by reference and this returns by value. Now, the question is so in each of this function I take t as a reference of type T and return this.

So, basically the body is the same. Now, how does the semantic differ? If I write auto& and if I write decltype(auto), if I write auto what will happen, auto strips off all reference and all that. So, if I try to do foo(a), foo(a) should return you a reference and therefore, you will think that since it is a reference I can make an assignment to it, so I can say foo(a) assign 20.

But this is, this will be an error because auto will strip off the ampersand, will strip of the reference it will just return a type int which is an rvalue to which you cannot make an assignment, whereas if you do foo(b) which is doing, not trying to assign, foo(b) simply has an int type, it applies that and forgets the temporary result, it is fine.

Similarly, if you try to do it on bar where auto has an ampersand, then auto will necessarily put the reference in every case. So, now if I do foo bar(a) I will get a reference to int, l value will be type of int& and it will be fine. But if I try to do foo bar(b), then I have a reference and a reference without an initialization is not allowed, so I will have an error. In terms of decl auto, decltype(auto), it will figure out what is the actual type. So, if you call it with a it knows that the type is int reference, so it will allow it int reference, if you call it with b it will figure out that the type is int, it will give you a type int and both of these will be correct. So, that is the difference between using auto and using decltype(auto).

(Refer Slide Time: 35:26)

Suffix / Trailing Return Type: auto / declspec(auto)

- Recommendations
 - auto
 - ▷ Use auto to return a *prvalue* with *type deduction*
 - ▷ Use `auto&` or `const auto&` to return an *lvalue* with *type deduction*
 - `decltype(auto)`
 - ▷ Use `decltype(auto)` to write *forwarding templates*
 - ▷ Using `decltype(auto)`, the return type is as what would be obtained if the expression used in the *return statement* were *wrapped in decltype*
 - ▷ Without `decltype(auto)`, the deduction follows rules of *template argument deduction*
- Summary

```
auto foo() { // rt = int
  int x = 0; return (x); // returning local variable by value. foo
}

decltype(auto) bar() { // rt = int&
  int x = 0; return (x); // returning local variable by reference. Trouble
}
```

Source: *When to use decltype(auto) versus auto?*
Programming in Modern C++ Part 6: Primitives 184/21

So, here are some common recommendations that I have put, you can go through that.

(Refer Slide Time: 35:32)

Module Summary

- Introduced following C++11 general features:
 - auto
 - decltype
 - suffix return type

Programming in Modern C++ Partha Pratim Das 38/46/22

And, so this brings us to the end of this module and we have introduced three general C++ features of auto, decltype and suffix return type. Thank you very much for your attention, we meet in the next module.